20-EECE-4029 Operating Systems Spring, 2015 John Franco

Final Exam name: ______________________________

Question 1: Device Driver (20)

a. Data that is sent from an application on a local machine to a remote computer through a NIC is handled by the device driver for that NIC. How does the Linux OS present this data to the device driver?

Through a socket buffer (there are other acceptable answers)

b. Among other things, a NIC device driver must contain an initialization section, a section that allows manipulation of a packet header, and a section that transmits packets to the NIC. What other sections does the device driver developer have to worry about implementing?

Section to handle receive interrupts
Section to handle transmit interrupts
Section to return status (others are possible)

c. Communication with a device from the device driver may be through a mapped page of memory. In doing this, what is the role of virtual memory management and what minor problem that is presented by virtual memory has to be worked around?

Virtual memory management eventually transfers the contents of the physical memory address space, that is used for communication between the kernel and the device, to and from the device.

In setting up a memory map, a pointer to the space is handed to the driver. Since the pointer is fixed, the virtual memory manager cannot repage that space.

d. The device driver must ask the OS for resources and must tie the device and those resources together so the kernel knows how to respond to incoming and outgoing requests. Name some resources that are needed.

An interrupt line
PCI bus registration
Ethernet device allocation
Bus mastering
Memory mapped region
Work queue or Tasklet
Major and Minor numbers

Question 2: Memory Management (20)

(a) Why is there a distinction between paging and demand paging?

Paging: segmentation of memory used by a process into equal size blocks.
Demand paging: pages are brought into physical memory when needed.

(b) What is so good about demand paging anyway?

More processes can be running due to more available memory.

(c) Name a page allocator

Buddy allocator (what else?)

(d) When does a page fault occur?

When a process attempts to access a page that is not in memory, so the OS has to load it.

(e) What happens when a page fault occurs?

1. an exception is raised - the OS handles it
2. the state of the process causing it is saved
3. if its a page fault, it is determined that the address reference is valid
4. the location of the page on disk is found
5. the page is read from disk
6. the process that raised the exception is restarted
7. the state of the current I/O process is saved
8. the page table is updated
9. the state of process causing the fault is restored & process is resumed

(f) What does a page replacement algorithm do?

When a page fault occurs, and there are no more free frames available, a page replacement algorithm must choose a page to be replaced in memory.

(g) Name and describe a real fancy but practical page replacement algorithm

Second chance least recently used: a circular queue keeps track of pages and stores a bit that keeps track of recently used pages by making the bit as a 1. When a page needs to be replaced, the first that contains a 0 will be replaced and, while looking for that 0 entry, all visited page entries (their bits have value 1) will have their bits cleared giving them a second chance (to remain in memory).

Question 3: File Systems (25)

(a) Windows uses a variant of linked allocation to store files so that possibly scattered blocks can be easily located. Describe what linked allocation is using the example of FAT16.

Directory entry: 14 bytes for a name and attributes, 12 bytes for metadata, 2 bytes for first block (cluster) occupied by the file, 4 bytes for the size of the file in bytes. Example: `0x3E7F` byte file beginning at cluster `0x42`:

HOWDY	0x00	..	..	..	0x0AAD	..	..	0x0042	0x0003E7F

File Allocation Table: Contains a list of entries that map to each block (4 sectors or 2KB) on the volume. Each entry records one of the following: the block number of the next block in a linked list of block numbers indicating file position; a special end of block-chain (EOC) entry that indicates the end of the linked list; a special entry to mark a bad block; a zero to note that the block is unused. Each entry in a FAT corresponds to a block in the filesystem. A linked list corresponding to the above file whose data is found in blocks `0x42`, `0x44`, `0x46`, `0x47` (in that order) might look like this where entry `0x44` represents block `0x42`:

...	0x44	0x81	0x46	0x05	0x47	0x00	0x67	...

(b) If blocks are 4096 bytes, what percentage of disk space is used for the meta data that supports FAT16?

FAT: 2 bytes for every 4096 bytes, Directory: 32 bytes for each file. Number of files is no greater than number of blocks since cannot have more than one file per block. Hence Directory overhead is at worst 32 bytes for 4096 bytes. Total overhead is 34 bytes for 4096 bytes, worst case, or 0.83% assuming 1 FAT. For 2 FATs it's 0.88%.

(c) If blocks are stored consecutively on a disk then it is possible for random accesses to be fast for FAT16. But if additions to the file requiring more blocks occur, this property is lost. What can be done to mitigate the effects of this problem?

defragmentation
allocate more blocks than is needed initially

(d) The linked structures of FAT16 are stored at one location on a disk. A second structure is also stored elsewhere as a backup in case one of the two becomes corrupted. But, `inodes` in unix-like systems are scattered over the entire disk. Why is that?

If inodes are placed near the blocks that they reference, there will be less hard disk arm movement needed to retrieve and store data on the disk.

(e) What kind of disk scheduling algorithm is suitable for SSDs?

FIFO

(f) The original implementation of indexed allocation for Linux was great for locating blocks in small files but not so great for large files. What was the problem and how was it fixed (or at least made more practical)?

In EXT2 an unbalanced tree structure was used to locate the blocks of a file. After EXT2 a balanced tree structure was used to achieve a uniform log depth regardless of block. In addition, a single leaf could represent many blocks if they represent contiguous data in a file.

Question 4: ACPI (15)

(a) ACPI is an open standard for configuring devices and managing power by the OS kernel. It is supported, at least in part, by all the major OSes. But the hardware must also comply with the standard. What does hardware (processors) provide that allows for direct power management?

The ability to control voltage and processor speed, either by direct input or instruction, to various sections of the processor.

(b) There is something potentially wrong with the code below? What is it? The ... are not the problem - they were used to signify use of additional variables and to save space.

```
int init_module () {
    void battery_check(struct work_struct *work) {
        acpi_status status;
        acpi_handle handle;
        union acpi_object *result;
        struct acpi_buffer buffer = { ACPI_ALLOCATE_BUFFER, NULL };
        int v1, ...;
        acpi_string s1, ...;
        status = acpi_get_handle(NULL, "\\_SB_.BAT0", &handle);
        status = acpi_evaluate_object(handle, "\_BIF", NULL, &buffer);
        result = buffer.pointer;
        if (result) {
            v1 = result->package.elements[0].integer.value;
            ...
            s1 = result->package.elements[9].string.pointer;
            ...
            kfree(result);
        }
    }
    return 0;
}
```

The code is run in `init_module` so it will lock up soon after the driver is loaded.

(c) What can be done to raise confidence in the reliability of the code?

The code should be run in a kernel thread or workqueue.

Question 5: Threads (20)

Distinguish the mechanisms below from each other by indicating some important property that they have and how they are used.

(a) spinlock

A test is made to see whether a thread may acquire the lock. If the test fails, it will be tried again and again until it succeeds. In a well designed system a `spinlock` will usually not fail to acquire the lock and therefore will operate with low overhead. But the code it protects must run atomically or else there is a risk that the system will hang.

(b) trylock

A test is made to see whether a thread may acquire the lock. If the test succeeds the lock is acquired but if it fails, the trylock is exited immediately with a return value that indicates failure. By allowing pre-emption, a trylock may be used to avoid deadlock. However, livelock is possible.

(c) semaphore

Checks a counter to determine whether a thread can continue past the semaphore. If the count is 0 the thread is suspended. If the count is greater than 0, the thread continues past the semaphore and the count is reduced by 1. The count may be incremented by 1 from anywhere using a function developed for that purpose. Since ownership of the section of code the semaphore protects is irrelevant, overhead is generally less than that of mutexes. Semaphores can simulate mutexes but also can easily handle cases where some number of threads may be operating in a critical section.

(d) mutex

A thread cannot get a mutex lock on a critical section that is "owned" by another thread until either that thread gives up the lock or is suspended due to a condition variable setting. Support re-entrant code.

(e) barrier

Stops a thread until all reads or writes or dependent reads, depending on the barrier, are completed. Used to undo the effects of compiler or hardware optimizations which may reorder execution of code or, in the case of compilers, combine lines of code, when inconsistent results are possible.