

Important Issues in the Experiment

181240020 Hu Junhao

July 2, 2020

1 L1

1.1 Details in the alloc and free design

1.1.1 Agreement

- The header is inserted within the heap and “right” before the beginning of the allocated memory. There might be some space between the header and the allocated memory due to the alignment requirement. And the address of the header is inserted right (without quotation mark) before the beginning of the allocated memory.
- Whenever we got an allocation request with n bytes, we always round it up to the nearest number of bytes which can be divided by 8. In this case, although we might waste tons of memory when small amount like 1 byte of memory is requested, we are assured that the address of each header is appropriate for data alignment needs.
- We did not use all of the heap memory. We deliberately cut out the last peice of memory for a void header.
- When we call `alloc(0)`, we return `NULL`.

1.1.2 alloc

- Whenever we got an allocation request with n bytes, we always round it up to the nearest number of bytes which can be divided by 8. And at the same time, we store the address of the header right before the beginning of the allocated memory. This altogether takes $O(1)$ time. But we have to spend extra time sweeping through the list tp find out which block is suitable for the allocated memory, which takes $O(n)$, and n is the length of the list, which might grow very large as the call to free function increases.
- We have `bigalloc` and `kalloc`. Since we know that the biggest alloc call is 4096, we can `bigalloc` a big “page” beforehand and alloc the comparably small memory in the “page”.

1.1.3 free

- When we want to free one block, we first get an address of the to-be-freed memory block and take a step forward, get the address of the header, free this block and return it to the free list, which only takes $O(1)$.
- Notice that when we free a certain address, we simply set the sptr of the header to 0 without returning it to the free list. After certain amount of time, for example, after certain amount of frees are called, we go through the whole allocated lists and find out those big “pages” whose members are all freed. And we return this whole “page” back to the free list.

1.2 Lock

1.2.1 First try

- Originally, we only put a large, heavy lock whenever we need to access the heap and the corresponding header — a list structure. This is helpful in the very beginning, in that we can fully focus on the correctness of the alloc and free function.

1.2.2 Modification

- Then we only add the heavy big lock when we call bigalloc. And when we allocate small memories we do not need any locks, which boosts the level of parallelism.

1.3 Problems encountered

- It is really hard to find the correct parameter, which optimize the whole allocating process. For example, how large a big “page” should be? Should it be 4096, or $4096 \cdot 2^n$, or $4096 \cdot 2^n + b$? We take the last type of parameter since we need to keep space for the headers inserted within the pages. When I set the page size to be $4096 \cdot 2^n$, the program fails on online judge. Nevertheless, $4096 \cdot 2^n + b$ is much more faster than that and the program passes.
- There is a challenging request — the restriction on the starting address of the allocated area. Considering the fact that we have incorporated headers right before the allocated space, we should be really careful about the size of the header. If the size of the header is 2^n , it should be much easier for us to implement the program and we are also able to save a lot of space which might become internal fragment if the size of the header is, for example, an odd number.

2 L2

2.1 Simplification and assumption

In order to avoid the potential trouble brought by L1, we replace the delicate and complicated realization of alloc and free with the simplest version. where there is only malloc and we do not need to return the memory back. This greatly reduces our concerns in the experiment, and allow us to focus on the possible bugs appeared in L2.

2.2 Design for locks

Since we need to close the and open the interrupt while we are playing with locks, it is necessary for us to record the state of interrupt before we acquire the lock, so that we can see the interrupt properly after we return the lock.

We used a global variable `lk_intr` in order to record the previous state of interrupt. The correctness is easily arguable. We can simply add possible interrupt points between each code lines in `spin_lock` and `spin_unlock` and see what happens. Here we omit the proof.

2.3 Design for semaphores

In order to keep the “First arrived first served” policy, we use an FIFO data structure — a queue to record the order in which tasks ask for semaphores.

And we met an interesting bug when designing semaphores. Here in the code we used a Flag to denote whether the acquirement is successful, which proves to be right on online judge. However, originally, we simply write “if(`Current_stat` == HUNG)_yield()”. And this leads to every tasks asking for this semaphore to sleep forever. We are still researching on the possible reasons for this phenomenon.

2.4 Design to avoid CPU starvation

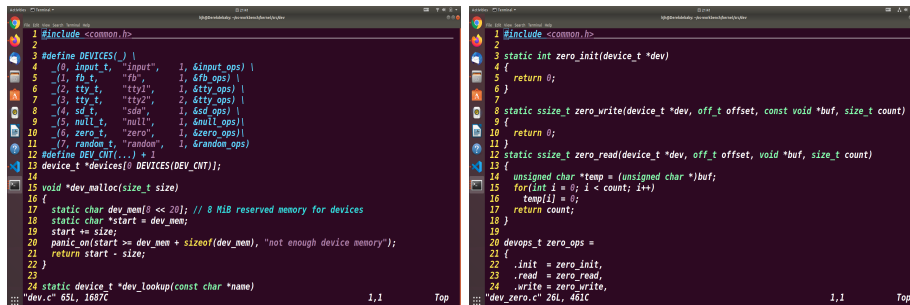
The general idea cannot be simpler, but we have to avoid stack race. The basic idea is also simple — since we do not want a task, that has just been dragged down from a CPU to be scheduled immediately, we put a marker on him and schedule this task the second time we meet him.

2.5 Some thoughts

Bugs on semaphore (mentioned above) really drove me crazy those days. And back there, for several times, I wanted to give up, and found tons of excuses for myself — Debugging for this long is time-consuming while I really gained nothing during this whole time, and I have finished other experiments so I do not need to worry about my scores if I just give this one up.

However, thanks to my best friend Hung Tao, he kind of saved me from the pit fall. I have always considered him my friend as well as respectable rival. He

Figure 1: Abstractions on three virtual devices



has finished this lab early and with ease. I just felt jealous and uncomfortable. Besides, I believe he also considers me as his best friend, respects me, and wants to study with this “equally excellent student”. I felt sorry to fail his trust and feelings. And that’s the reason why I moved on. I picked up the lab half month after I quit it, and run tests, search for bugs. Finally, here I am, with four greeny and lovely “accept” on the screen.

Never give up, no matter it is for the rest of the labs or for the rest of my life challenges! And million thanks and bless to my best friend Huang Tao.

3 L3

3.1 Design Trick

Since xv6 is literally the best learning materials, I spent half a month studying the part of code about file system, and make sense of every line of it.

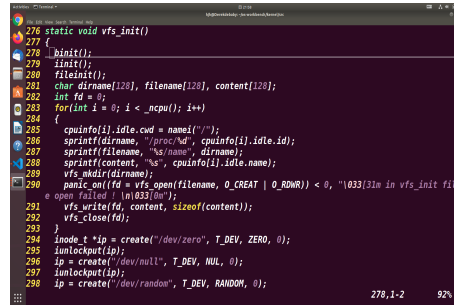
As a result, I figured out that, the file system we are about to construct, is nothing more than a simpler version of that of xv6 — simply a filesystem without logging mechanism. However, there is something more about L3. All the extra work we will have to do, is to record the information of each thread and build an abstraction on those three virtual devices. As for all the other part, we can simply type along the xv6.

3.2 Abstraction of Devices

Imitating the way we construct abstraction on the input, tty... devices, I incorporate three more virtual devices into dev.c. If we want to read from or write to those devices, we simply call the corresponding read function and write function. More details are shown as in Figure 1.

In vfs_init, we manually add in /dev, dev/null, dev/zero, dev/random, with correct file type. When we encounter T_DEV type as dev/null. dev/zero, dev/random, instead of read from or write to the file, we call the corresponding read and write functions.

Figure 2: Store information about thread in /proc



```
276 static void vfs_init()
277 {
278     kinit();
279     fileinit();
280     char dirname[128], filename[128], content[128];
281     int fd = 0;
282     for(int i = 0; i < _ncpu(); i++)
283     {
284         cpuinfo[i].idle.cwd = name("/");
285         sprintf(dirname, "/proc/%d", cpuinfo[i].idle.id);
286         sprintf(filename, "%s/name", dirname);
287         sprintf(content, "%d", cpuinfo[i].idle.name);
288         vfs_mkdir(dirname);
289         panic_on((fd = vfs_open(filename, O_CREAT | O_RDWR)) < 0, "03331m in vfs_init fil
290         if open failed: %s\n", strerror(errno));
291         vfs_write(fd, content, sizeof(content));
292         vfs_close(fd);
293     }
294     inode_t *ip = create("/dev/zero", T_DEV, ZERO, 0);
295     iunlockput(ip);
296     ip = create("/dev/null", T_DEV, NULL, 0);
297     iunlockput(ip);
298     ip = create("/dev/random", T_DEV, RANDOM, 0);
299 }
```

3.3 Record of Threads

Similarly, we manually create /proc in vfs_init, and whenever we call kmt_create to create a new thread, we build corresponding directories and files storing information about the thread. More details are shown in Figure 2.

3.4 Brief Notes on Other Part

When we are writing codes for L3, the main learning material is code in xv6. Therefore, as in xv6, we also have buffers for each data block.

3.5 Rearrangement of Files

Originally, the file is in a mess — Header files embedding each other. Therefore, we modified files like devices.h. we guarantee all the definitions happen only once, and provide all the interfaces in common.h, making sure all the source files contain the common.h header.

Thus, whenever we want to call whatever function that has been defined in anywhere of our project, we just call it!

More details are shown in Figure 3.

3.6 Debugging

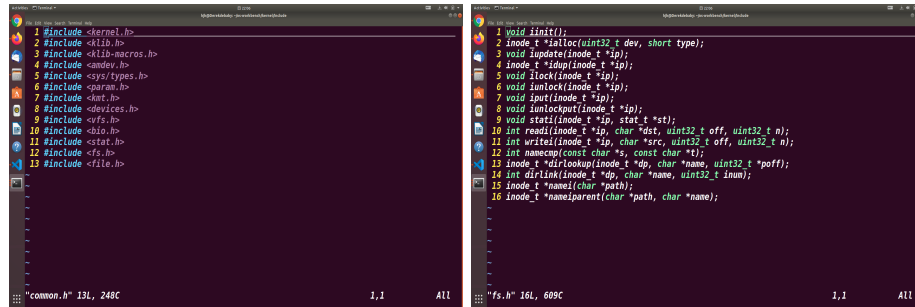
I have gained useful experience in debugging when after I finished L3 — half finished.

Since this is a huge project, and we actually build up several layers of abstraction on the disk — buffers, file systems, the actual file..., we should make sure each layer is working correctly in a down-top approach.

The lower layer is much simpler and once we are assured that it can work without errors, we are less anxious to use it in order to build the upper layers.

Also, I learned how to deal with the “dead end”. Sometimes, we might get stuck in the middle of our process. And we can insert some printf codes to locate exactly where we get stuck.

Figure 3: common.h and some of the interfaces in fs.h



4 Conclusion

It is really a fresh and challenging experience to literally build a mini, ultra-simple but complete operating system from scratch. I have been so familiar with both the general view and most of the details of a operating system right now. Quoting from my os teacher jyy, “I survived!”.

However, I still have a long way to go. There are still a lot more things to learn, a lot more problems to solve. Surviving the os class is nothing more than a small step forward towards the “eventual success”.

I will keep up with good work and do better in the future class.

In the end, thanks, os class, thank you, dear jyy.