**Report of Project 2**

**卞思沅 518021910656**

**1. Introduction**

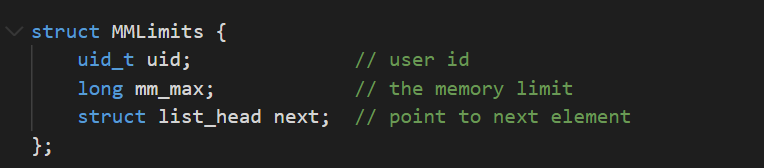
In this project, I have implemented a new system call to set the memory limit, and then devise a new oom killer to kill the largest process whose user has exceeded its memory limit. In the report, I will

1. Explain how your system call works in detail.
2. Explain how the original OOM killer is triggered.
3. Explain how you design and implement your new OOM killer.
4. Show the results of my implementation
5. Explain my implementation of some extra parts.

**2. System call**

Generally speaking, I add system call according to the blog[[1]](#footnote-1). Here I will give a detailed explanation of my code.

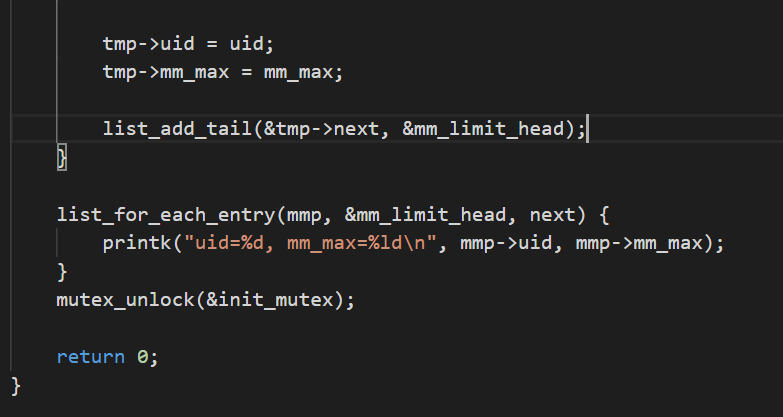
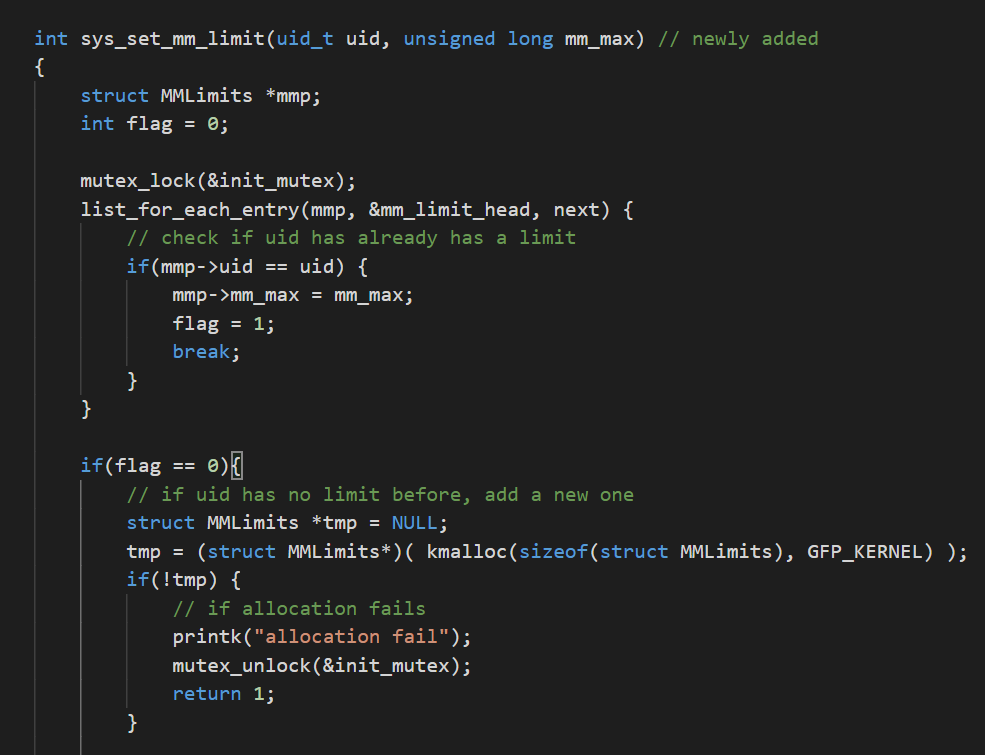
In the sys\_arm.c file, I define a new structure to help record the memory limit of each user. The struct is defined as below:



The notable point is that I use list\_head to help us simplify the code. With the help of *list\_head*, we do not need to use *kmalloc* to allocate kernel memory at first call, and we do not need to initiallize the whole array as well. We can add struct to the tail of the list one by one whenever necessary. In this way, we have saved a lot of memory space. And there is no need to worry about whether the array is long enough or whether the space is enough to store the array. So the number of possible problems drastically decreases.

We define a global list\_head variable *mm\_limit\_head*. In later part, we will refer to memory limits in my new *oom\_killer* through this variable.

In the system call, we first check whether the current user has already has a limit. If so, we just update the limit. Otherwise, we create a new *MMLimits* struct and add the struct to the tail to the *list\_head*. Here is the code:



**3. Original oom killer**

The original oom\_killer is triggered by *\_\_alloc\_pages\_may\_oom()* in *page\_alloc.c*.

*\_\_alloc\_pages\_may\_oom()* will call *out\_of\_memory()* when the memory is exhausted. Then *out\_of\_memory()* will then kill one process according to its designed algorithm.

In *out\_of\_memory()*, it first do some checks to make sure that we have no other options but kill one process. Then it will call *select\_bad\_process()* to find a victim process. If the selected process is killable, it will use *oom\_kill\_process()* to kill it, and return to normal routine.

The *select\_bad\_process()* function chooses the victim process according to point of each process scored by *oom\_badness()*. Specifically, it will call *oom\_badness()* to rate a point from 1 to 1000, and the process with the highest score will be selected as the victim process. Function *oom\_badness()* rates the score mostly by the amount of memory one process has used, but it also takes other factors into account. For example, if the process belongs to a root user, it will get lower score. And *p->signal->oom\_score\_adj* is also used to avoid kill important processes.

**4. New oom killer**

To check and kill a process when the user memory limit has been exceeded, we devise a new oom killer. Here are detailed explanations.

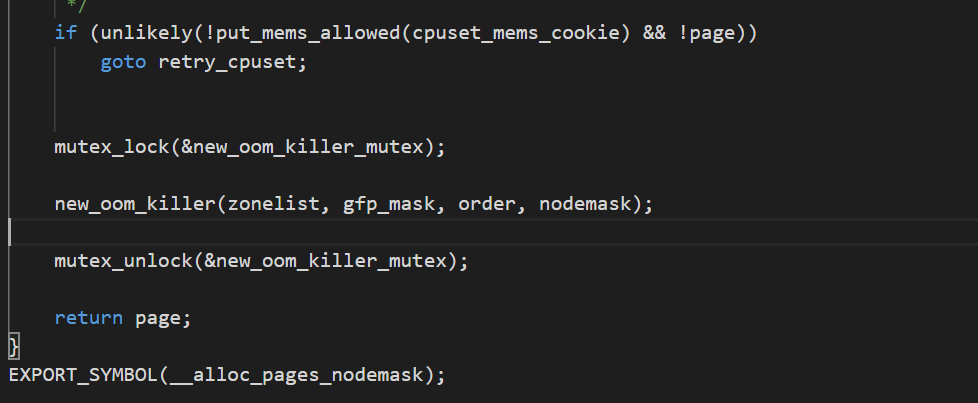
In *\_\_aloc\_pages\_nodemask()*, we add a new line, in which we call the new function: *new\_oom\_killer()*. And we add a mutex to protect it.

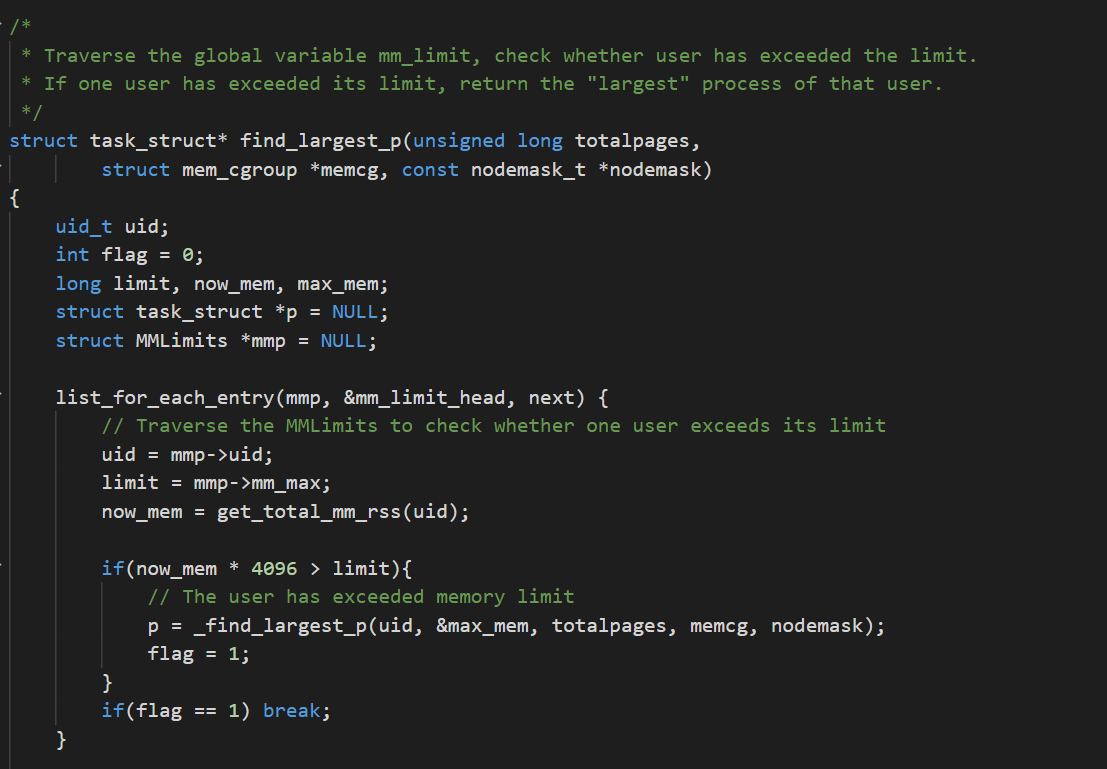
In *new\_oom\_killer()*, we first use *find\_largest\_p()* to check whether there exists a user whose memory usage exceeds its limit. If so, *find\_largest\_p()* will just return the largest process of that user, and we kill that process by *new\_oom\_kill\_process()*.

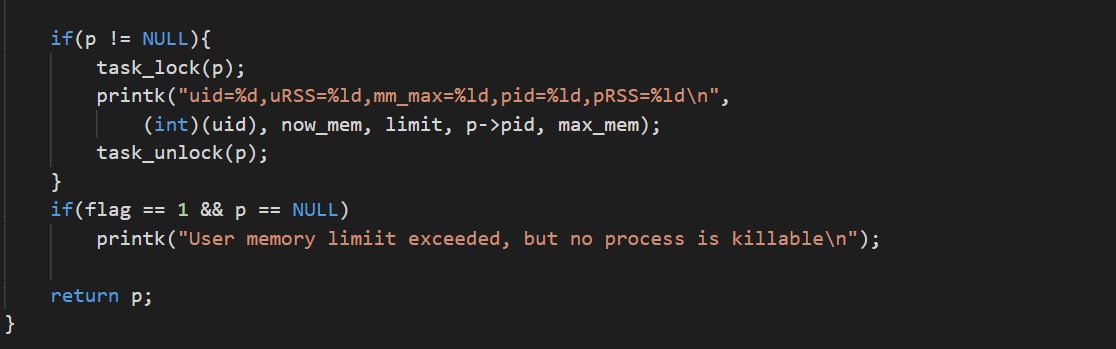
*new\_oom\_killer()* is quite similar to the function *out\_of\_memory()* from oom\_kill.c, except that I discard some unnecessary checks, because in our new oom killer, we only want to kill the exact out-of-memory process, and there is no need to check whether other processes are able to be killed and so on. And *new\_oom\_kill\_process()* is also quite similar to *oom\_kill\_process()*. I just also discard some unnecessary parts. The details can be refered in the code.

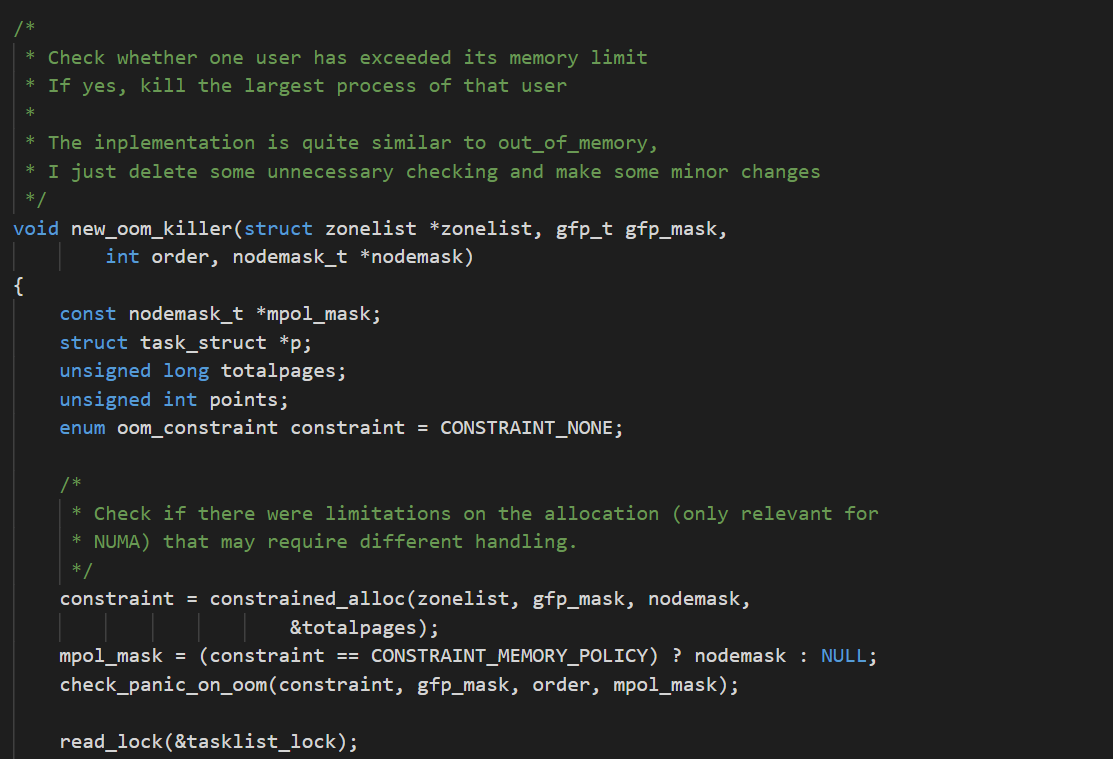
We use *find\_largest\_p()* to help us retrive the largest process whose user exceeds memory limit. If no user exceeds it memory limit, it will just return NULL, otherwise, it returns the needed process described above. In the function, we just traverse the MMLimits list, and check the users one by one.

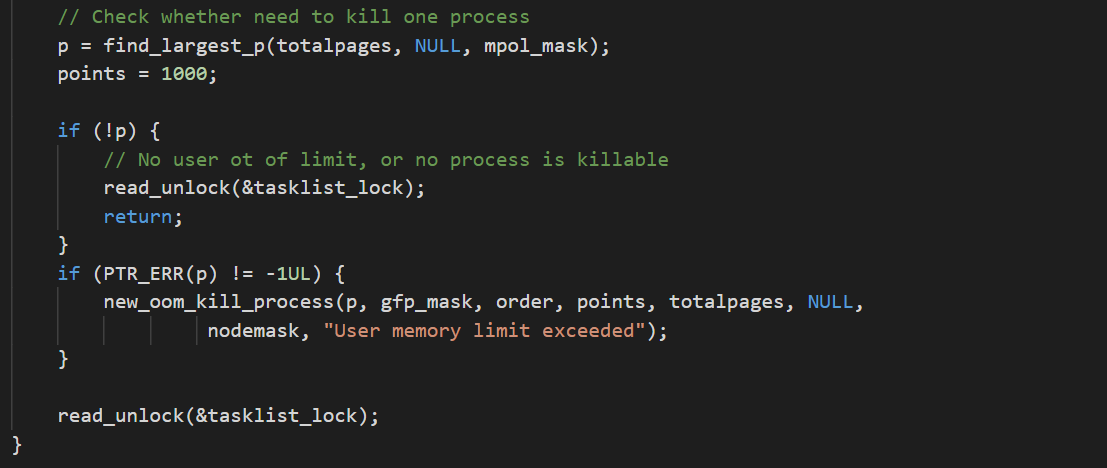
In the following part, I will just show part of important codes.







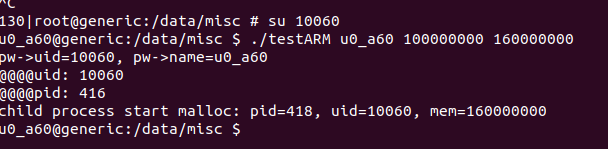




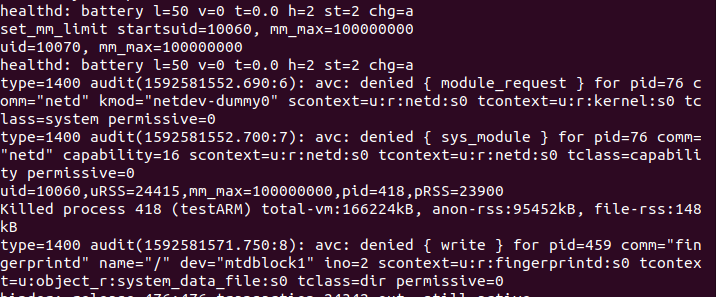
**5. Results**

I have set the memory limit for many users, and the new oom killer still makes good result. Here are the screen shots.

AVD shell:



Kernel:



The kernel also prints much debug information below, that is because I just call the new\_oom\_killer, and keeps many debug outputs. But they are not important. And between the lines are some other debug information, they have nothing to do with our project either, and it seems that the netd process whose pid=76 always has some problems, no matter whether we implement the new oom killer or not. And we should note that the unit of mm\_max is byte, and the unit for RSS is page, which may cause some misunderstanding.

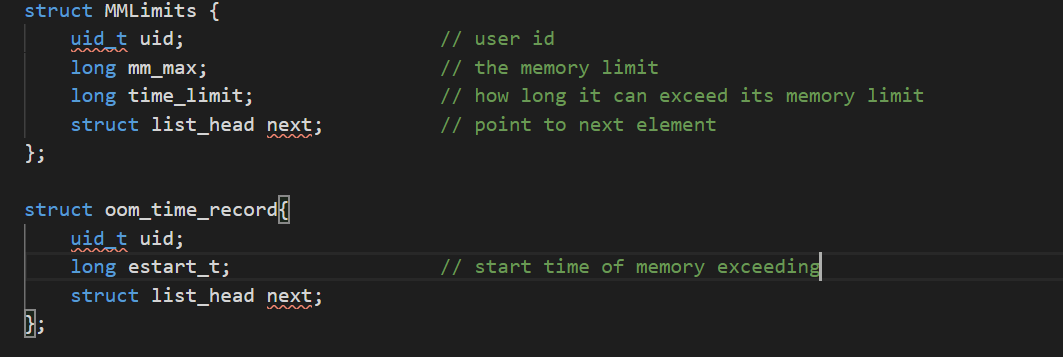
Note that the string *“child process finish malloc:”* is not printed in AVD shell, that is because before the finish of memory allocation of that child process, it has been killed by our now oom killer.

But in fact, there are still some unsatisfying parts in my implementation. I have found that the new oom killer will be called every time a new page is allocated. That influences the speed of memory allocation. I think that should be improved with further effort.

**6. Extra parts**

Because time limit, I fail to complete bonus part 1(trigger new OOM killer periodically). I successfully finished **bonus 2 and 3**. I have changed my code to take into account the time limit of exceeding memory limit. And I also add a new\_oom\_badness function to design a new reasonable rule to choose a victim process.

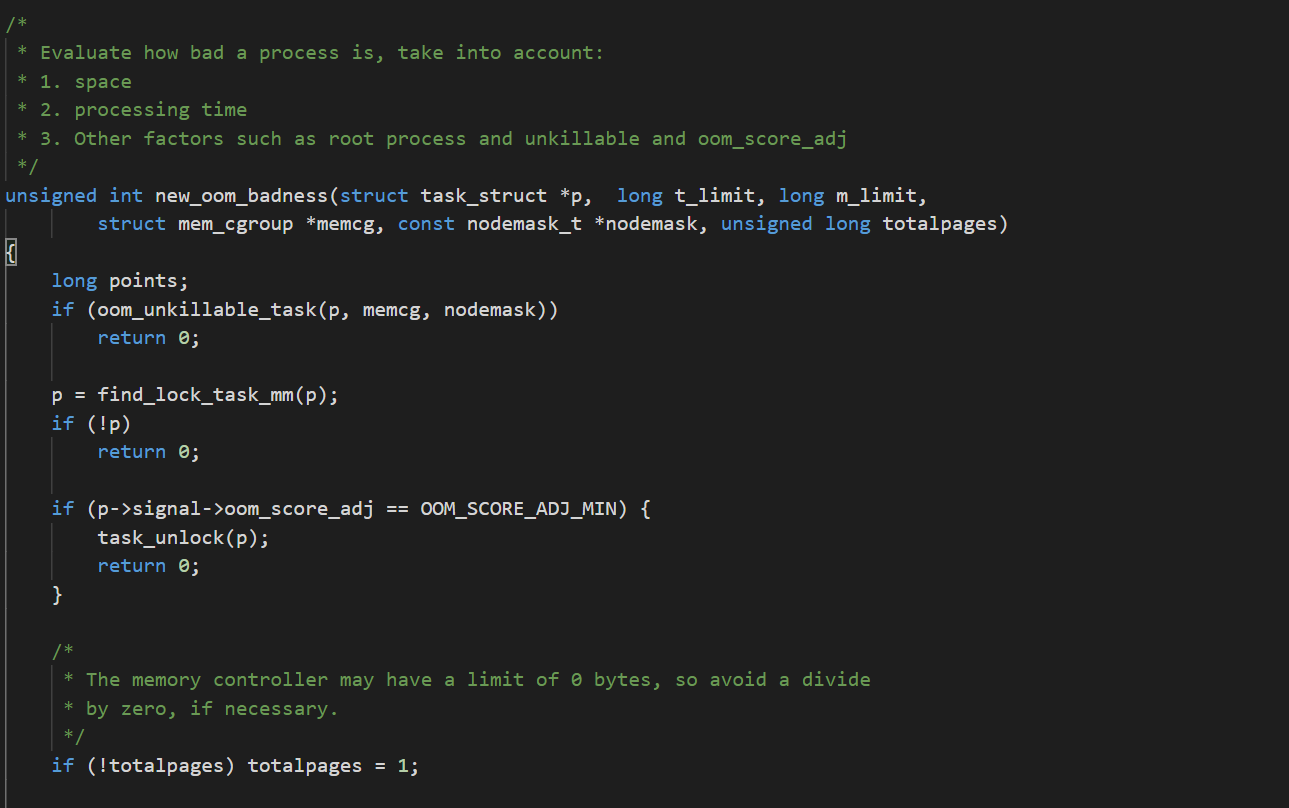
First, to record the time limit, I change the MMLimits struct and add a new oom\_time\_record struct to record the start time of memory exceeding as below:

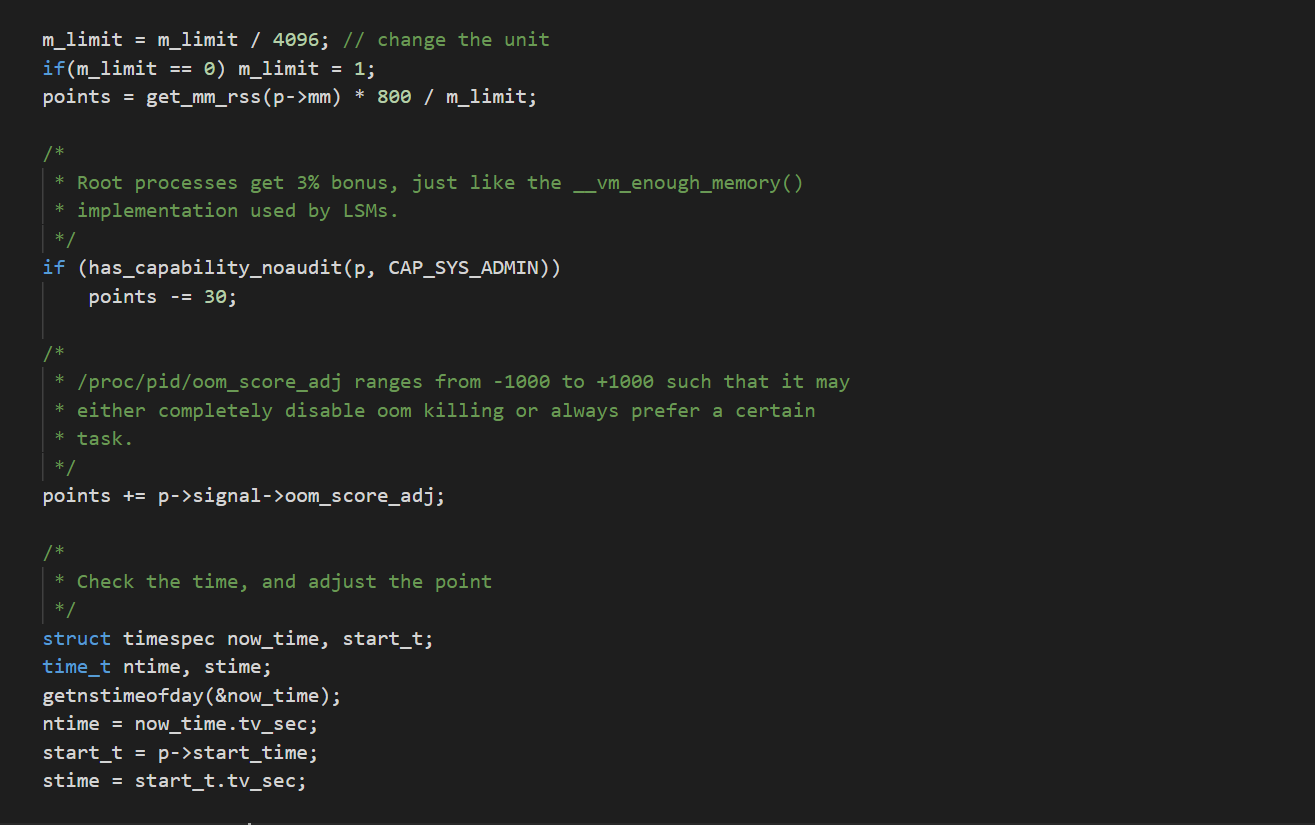


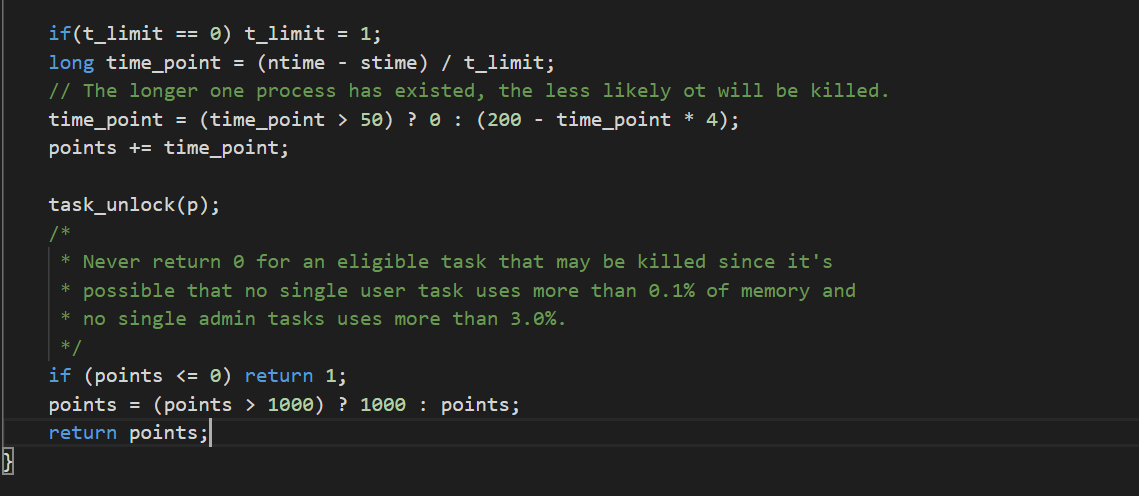
When we start a new oom killer, if the memory limit has been exceeded for the first time, we will record the start time of memory exceeding. Then when a second time a new oom killer is triggered, if the time limit for the user is still exceeded, we will check the time limit. But if the memory gets back to within the limit, then we will just remove the start time of memory exceeding of that user. If one user has exceeded the memory limit longer than the time limit, then one process will be killed.

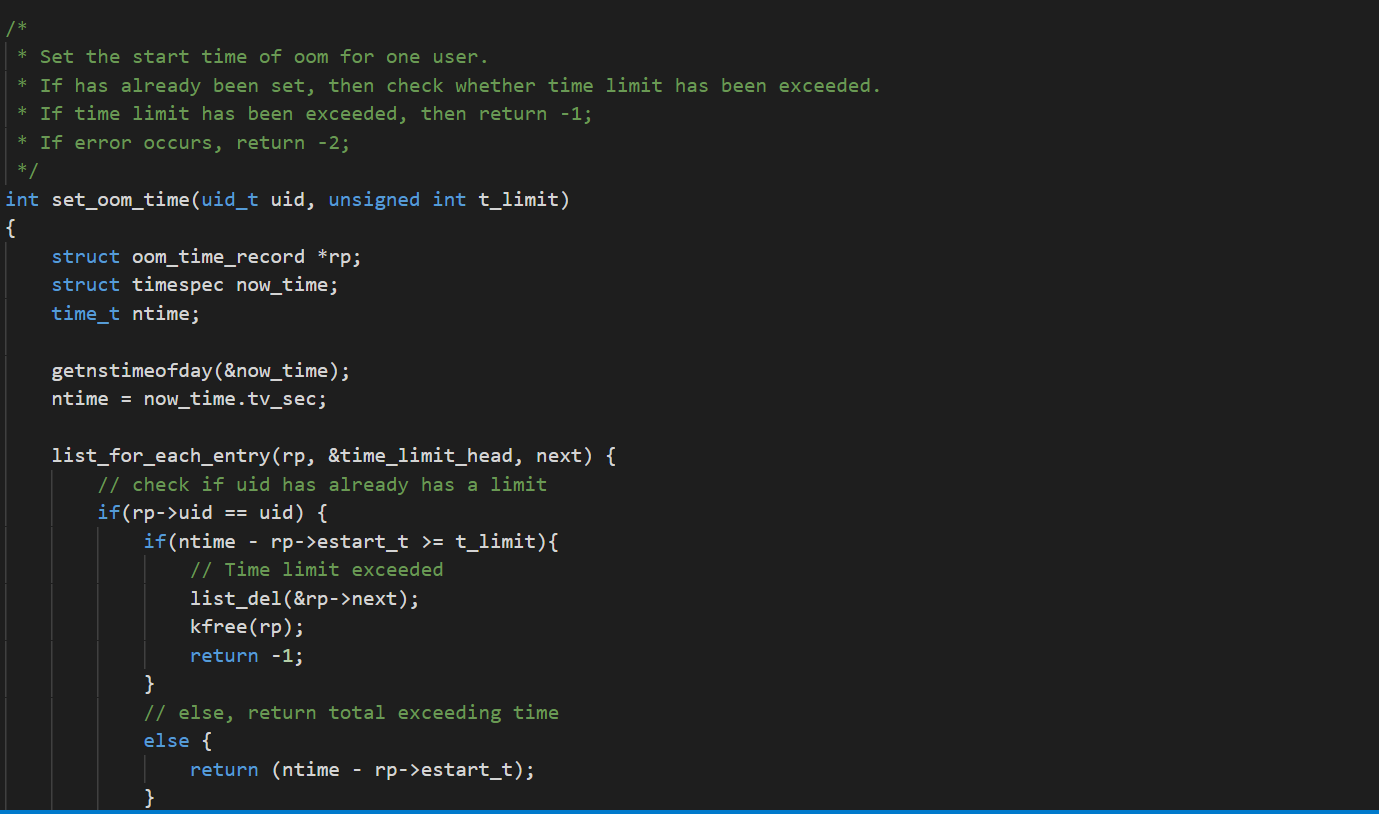
The new\_oom\_badness function follows the old oom\_badness function in that they all account whether the process is a root process and how large the oom\_score\_adj of a process is. But new\_oom\_badness evaluates more factor. It accounts both the start time of a process and the space one process takes. The earlier one process starts, the less likely it will be killed. That definitely decreases the chance of killing some important tasks. The processing time of a user is obtained by p->start\_time.

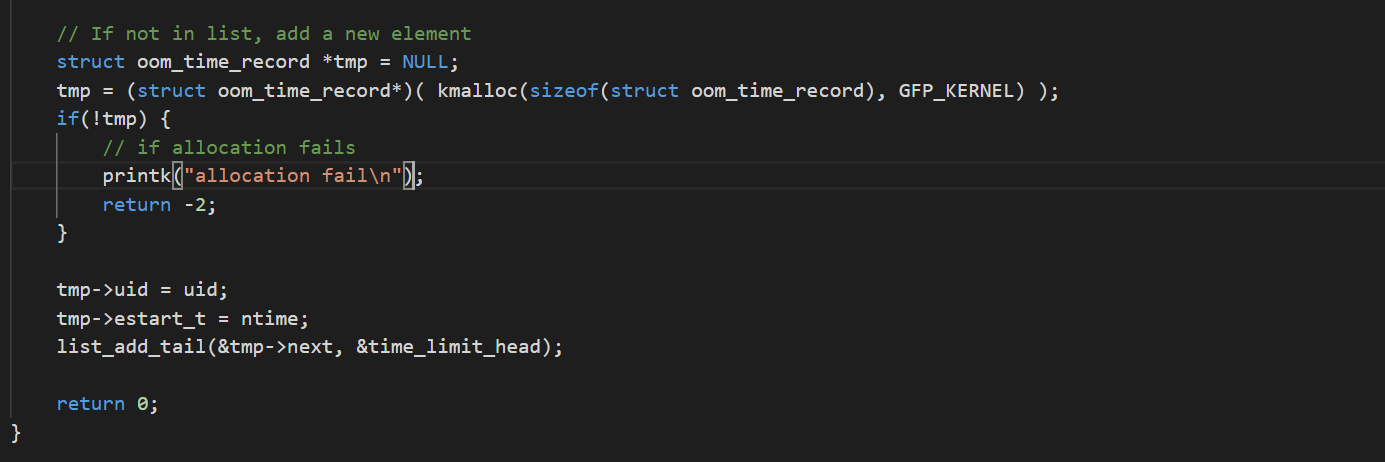
Here are the key parts.



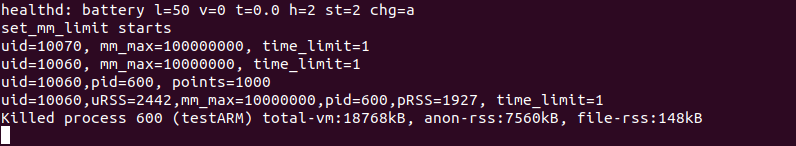


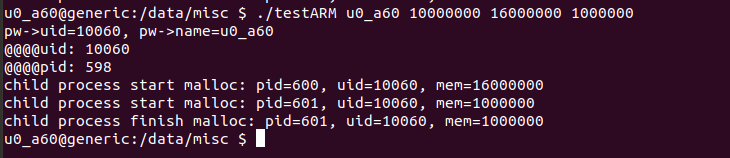






**Result of bonus part:** (I have set the default time limit to be one second)





The time limit is 1s. Because the allocation takes quite much time (1s or nearly 1s), so the total badness points get to 1000 for the first process, and thus the first child process is killed.

1. http://blogsmayan.blogspot.com/p/adding-simple-system-call.html [↑](#footnote-ref-1)