### Project 2: Memory Limit for Applications

518030990025 Eduardo Wang Zheng

1. Introduction

This project is divided into three steps.

Step 1 is to recompile the kernel. Step 2 is to add a new system call, whose basic function is to set the memory limit for a specific user. Step 3 is to design and implement a new oom killer.

1. Process of completing this project
   1. Recompile the Kernel
      1. Evaluation

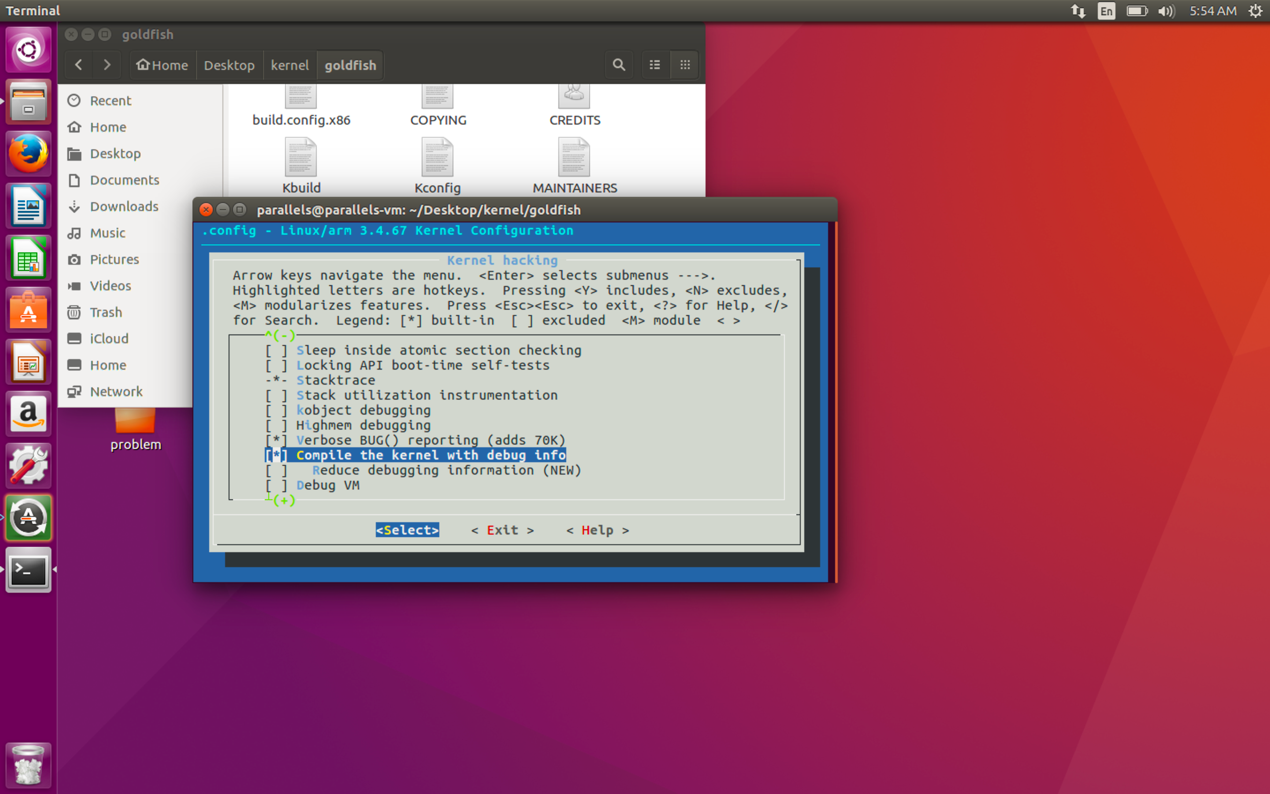


Figure 1. Open the **Compile the kernel with debug info**

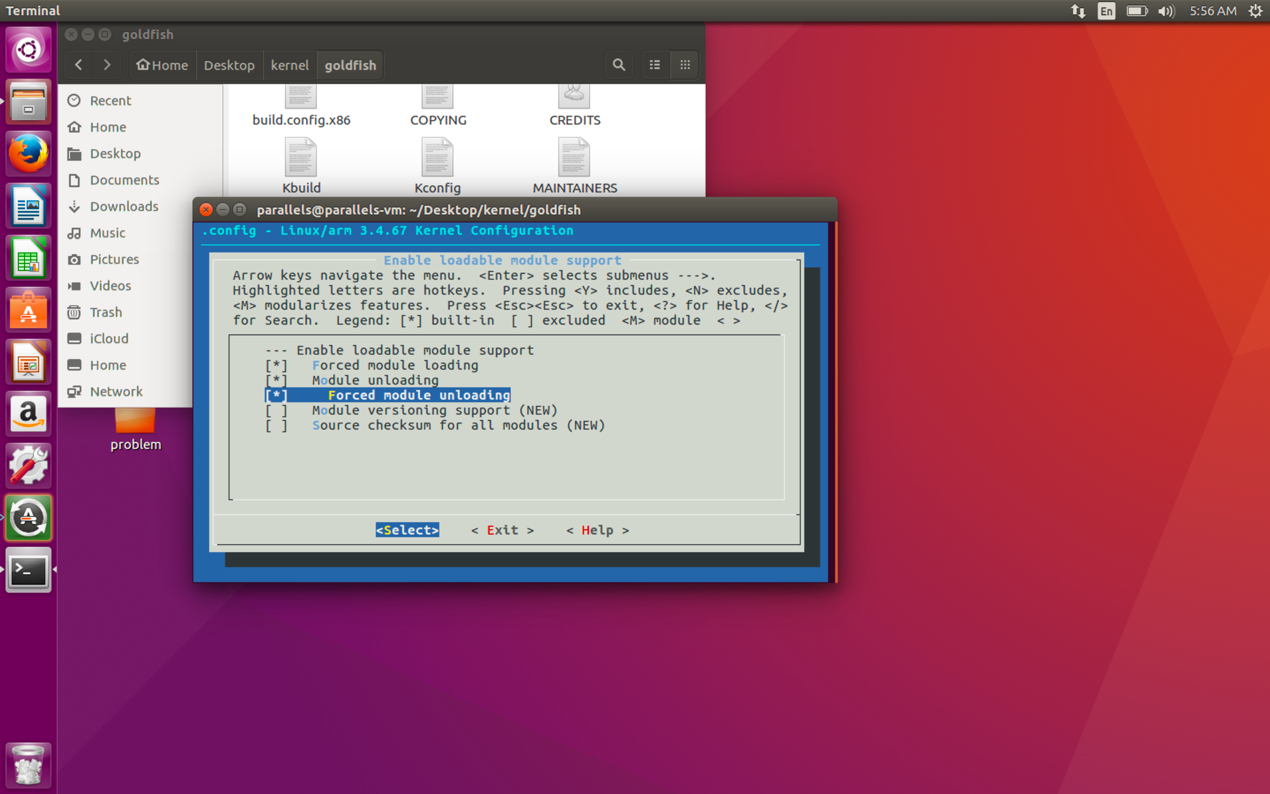


Figure 2. Enable loadable module support with **Forced module loading, Module unloading** and **Forced module unloading**

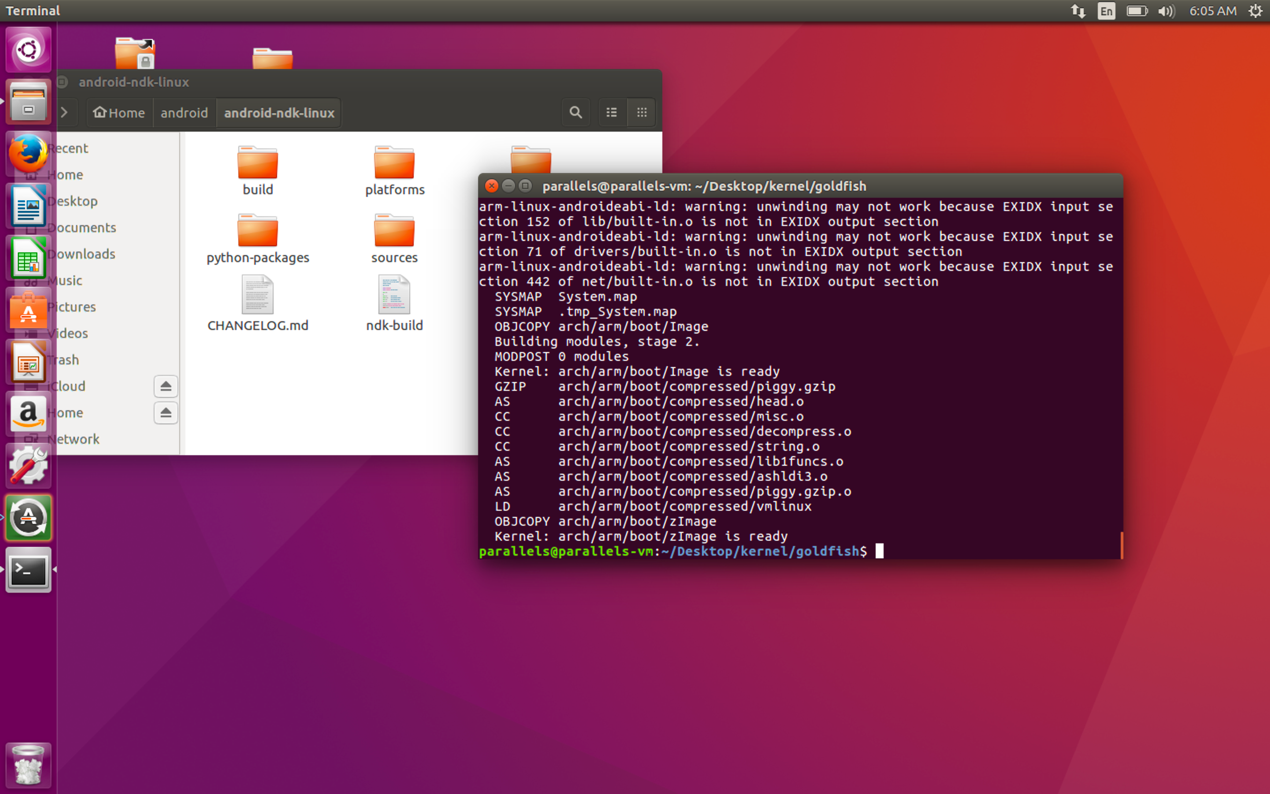
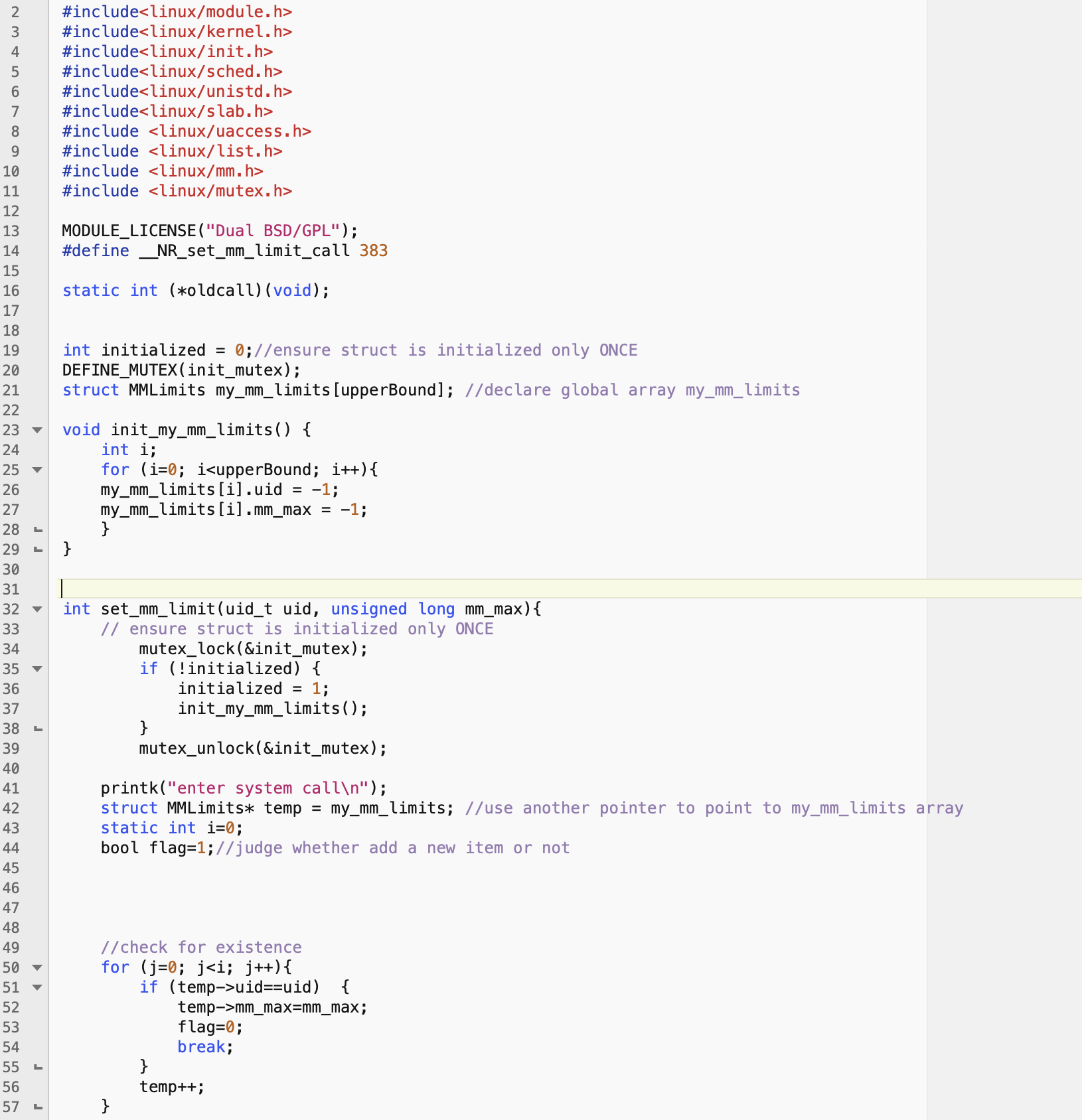


Figure 3. Recompile

* 1. Add a new system call
     1. Design of my system call

Procedure:

1. Check if there already exists a memory limit for that uid. If yes, it would update the old memory limit entry (the global variable) and set flag to 0, which means there is no need to add a new memory limit entry. Else, do nothing.
2. If flag equals to 1, add a new memory limit entry. Else, do nothing.
3. Traverse and output all existing memory limit entries using printk().
4. Return 0.
   * 1. Implementation of my system call
        1. Rewrite system call through kernel module



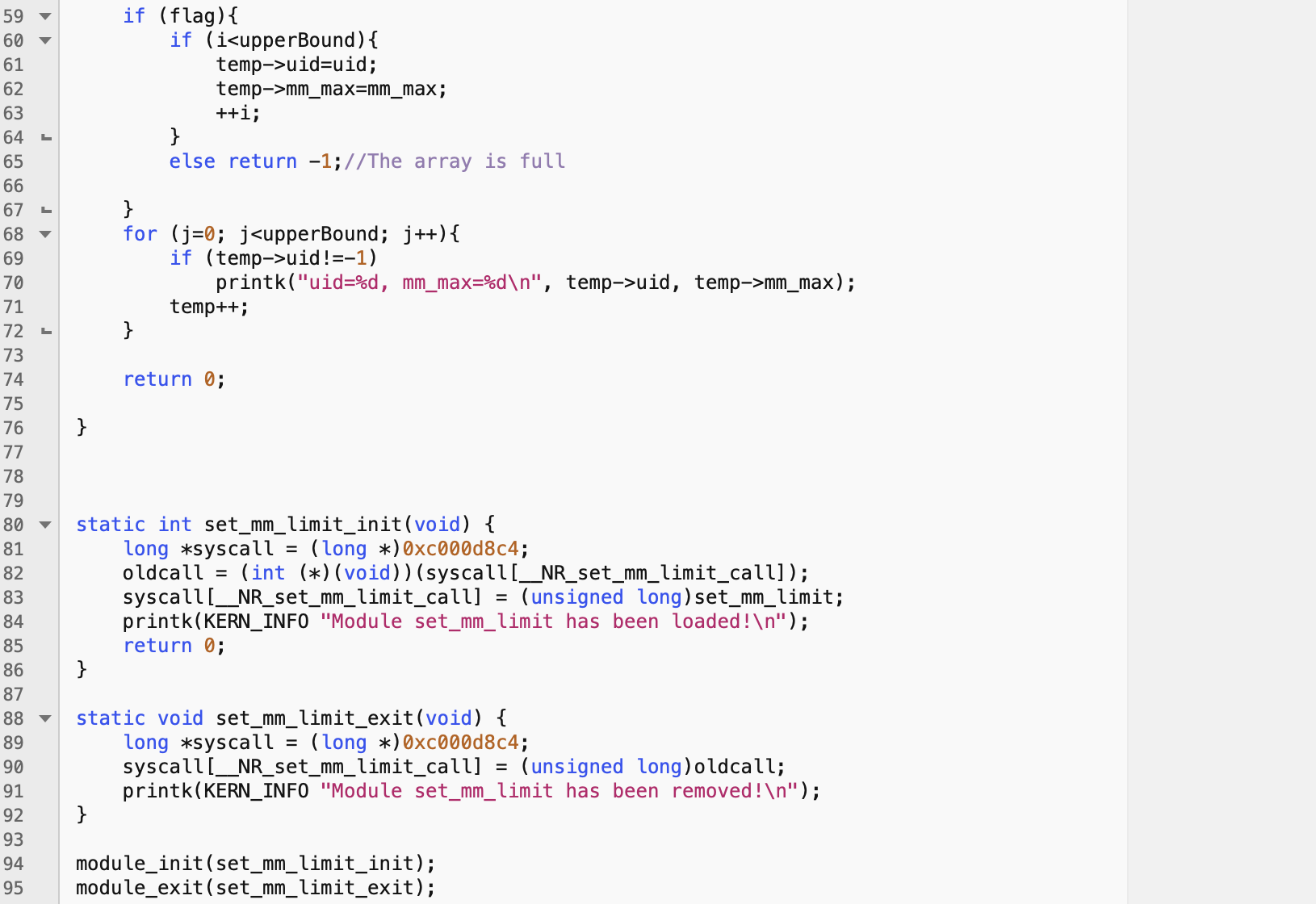


Figure 4 set\_mm\_limit.c (first try)

However, after testing I find there is something wrong with my method. I can only successfully insert the kernel module once. Later I find out this is because there are some errors happening when I add the global variable. I will explain it with detail in **section 3.1.1.**

With **Chan Young Lee’s help,** I try to add the new system call directly to linux kernel.

* + - 1. Add the new system call directly to linux kernel



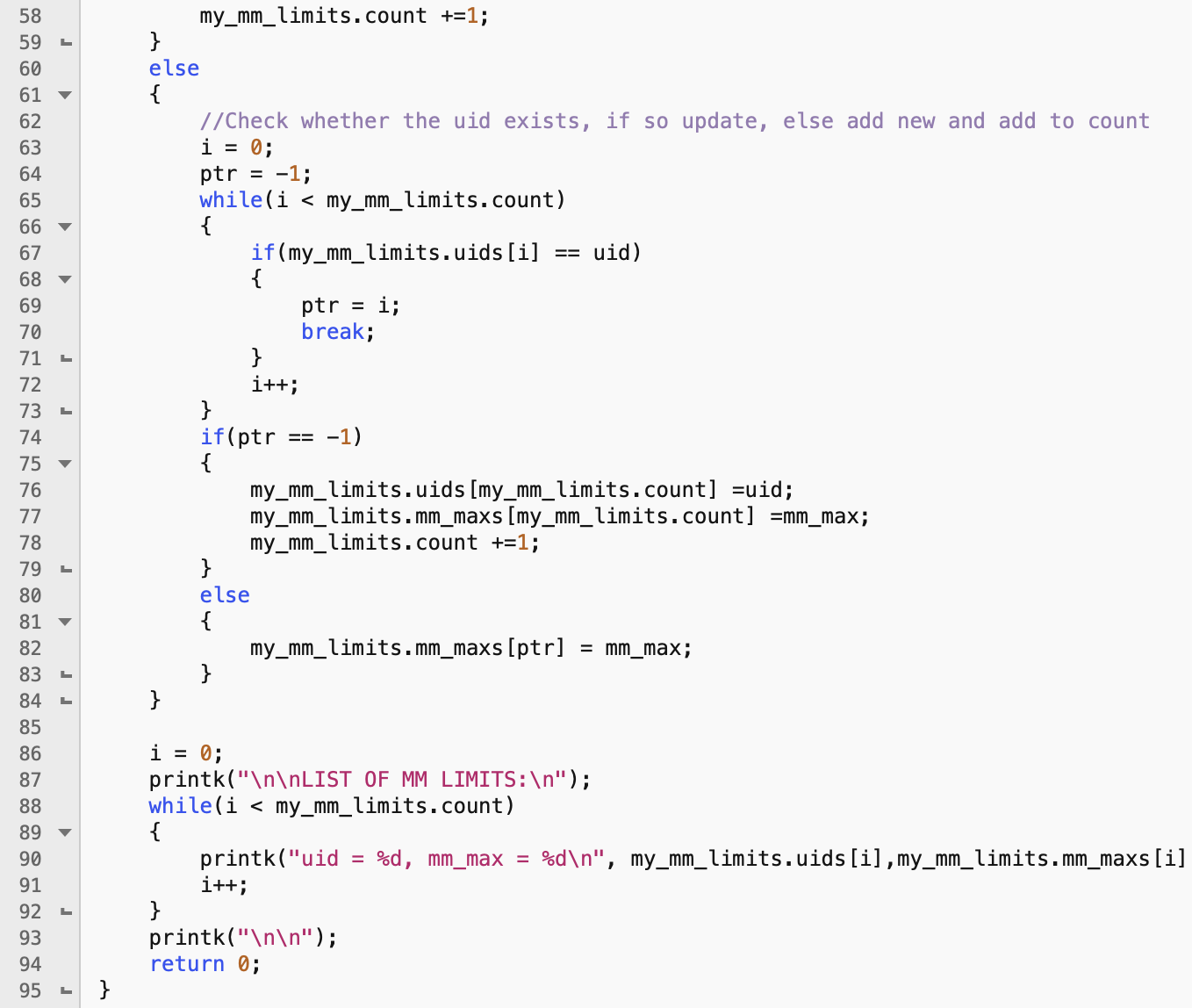


Figure 5 Add the new system call in **sys\_arm.c**

**Reference to Chan Young Lee’s code**



Figure 6 Update the system call table in **calls.S**

**Reference to Chan Young Lee’s code**

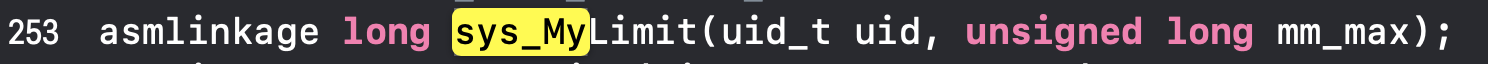


Figure 7 Register the system call in **syscalls.h**

**Reference to Chan Young Lee’s code**

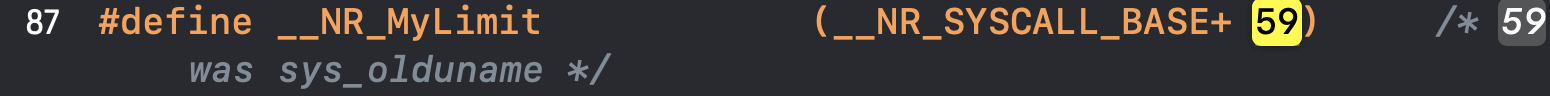


Figure 8 Set the system call number to 59 in **unistd.h**

**Reference to Chan Young Lee’s code**

Notice that the implementation of the system call is different from my previous one:

1. The **mmLimit** struct is different, which makes the **my\_mm\_limits** a structural variable instead of a structural array.
2. The initialization of **my\_mm\_limits** is different. Instead of using special number to mark the empty entries, this implementation initializes the two pointer to point to two arrays independently and then directly insert the first entry.
3. This implementation does not need to “count” from the begin of the arrays to the end each time we call the system call.
   1. Triggering mechanism of the original OOM killer

The oom mechanism is triggered when a process applies for memory resources but the system has no extra memory resources available.

When the kernel triggers the oom mechanism, it will call the ***out\_of\_memory*** function, which is called in the following order:

\_\_alloc\_pages

|-->\_\_alloc\_pages\_nodemask

|--> \_\_alloc\_pages\_slowpath

|--> \_\_alloc\_pages\_may\_oom

| --> out\_of\_memory

Let us have a brief look at each function.

1. The function ***\_\_alloc\_pages*** would directly call the function ***\_\_alloc\_pages\_nodemask***
2. The function***\_\_alloc\_pages\_nodemask*** mainly consists of two steps:

1. Direct distribution

2. If failed, try another way to continue processing, which is ***slowpath***

The first attempt to allocate is to call the ***get\_page\_from\_freelist*** function, the main functions of which are:

1. Find a suitable **zone** in **zonelist**;

2. Allocate pages from the **zone.**

Before allocation, the **watermark** is generally specified to allow allocation, or directly allocate, when the corresponding identifier is **ALLOC\_NO\_WATERMARKS**.

In the **zone** structure, there is a **vm\_stat** field, which is an array that records the number of pages in each state, including the free pages. And for the allocation with **watermark** it is necessary to verify whether the number of free pages **(NR\_FREE\_PAGES)** are greater than the corresponding **watermark**, and only when it is greater than the **watermark** can pages be allocated. Otherwise, it is necessary to reclaim the pages according to the situation. If they cannot be reclaimed or if the conditions are still not met after reclaiming, it will return directly.

3） The function***\_\_alloc\_pages\_slowpath.*** The algorithm is as follows:

i. Determines whether the caller forbids to wake up **kswapd** thread. If not, the wake-up thread performs memory recycling.

ii. Adjust the memory allocation identifier through ***gfp\_to\_alloc\_flags()***

iii. Call ***get\_page\_from\_freelist()*** again to try allocation. If succeed, return. Otherwise, continue to try memory allocation.

iv. Determine whether the **ALLOC\_NO\_WATERMARKS** flag is set. If it is set, **watermark** will be ignored and call ***\_\_alloc\_pages\_high\_priority()*** for allocation.

v. When return from ***\_\_alloc\_pages\_high\_priority(),*** judge whether the **\_\_GFP\_WAIT** flag is set from the ***\_\_alloc\_pages\_high\_priority().*** If it is set, it means that the memory allocation sleeps. Otherwise, exit directly due to memory allocation failure.

vi. Call ***\_\_alloc\_pages\_direct\_compact()*** and ***\_\_alloc\_pages\_direct\_reclaim()***to try to reclaim memory and allocate it.

If the above multiple attempts to allocate memory still fail, the **OOM killer** mechanism will be triggered by calling ***\_\_alloc\_pages\_may\_oom.***

4) The function ***\_\_alloc\_pages\_may\_oom***. The algorithm is as follows:

i. Determines whether the **oom killer** has been killing in other cores through ***try\_set\_zonelist\_oom()***. If not, it will lock in ***try\_set\_zonelist\_oom()*** , ensuring that only one core is killing.

ii. Call ***get\_page\_from\_freelist()*** to try to get memory again in the case of high **watermark**, but it is doomed to fail here.

iii. Then the key function ***out\_of\_memory ()*** is called. When the function exit, ***clear\_zonelist\_oom()*** will be called to clear the lock operation in ***try\_set\_zonelist\_oom()***.

5) The function ***out\_of\_memory.*** The algorithm is as follows:

i. First, call ***blocking\_notifier\_call\_chain()*** for OOM kernel notification chain callback processing;.

ii. Check whether **SIGKILL** signal is suspended or in signal processing, and if so, exit;

iii. Call ***constrained\_alloc()*** to check memory allocation limit and call ***check\_panic\_on\_oom()*** to check whether to report to Linux kernel panic;

iv. Check **sysctl\_oom\_kill\_allocating\_task** variable and do process check. If the conditions are met, kill the currently allocated memory process. Otherwise, we will select the best process through ***select\_bad\_process()*** and call ***oom\_kill\_process()*** to kill it.

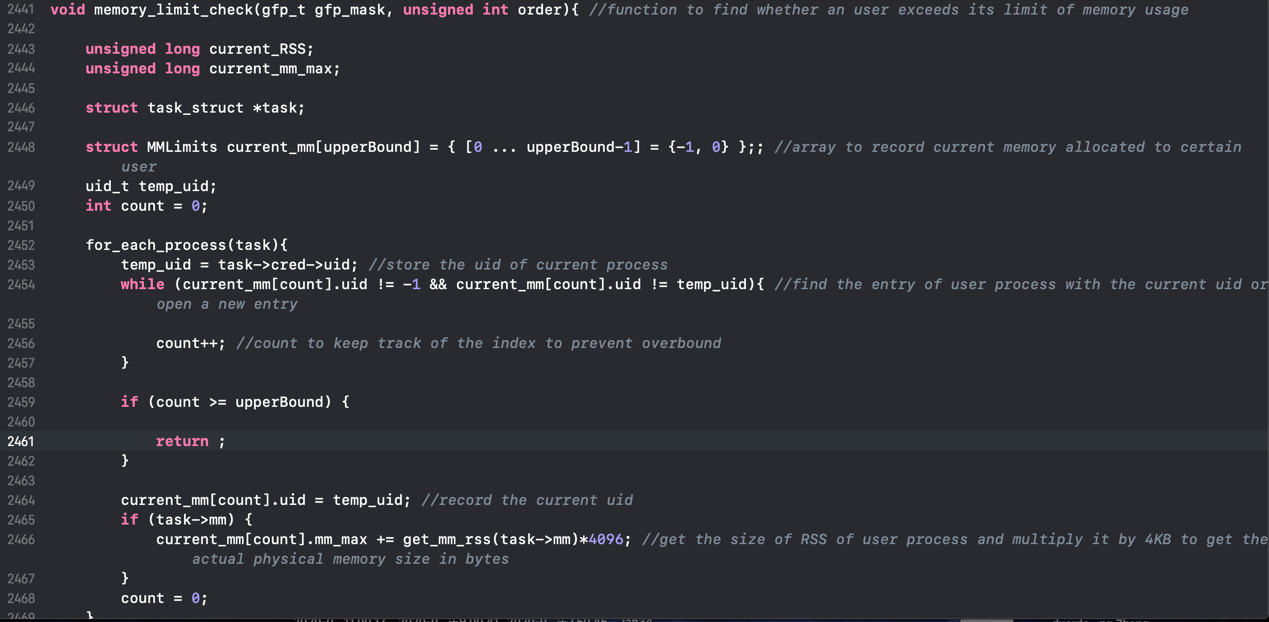
* 1. Design and implementation of my OOM killer
     1. Initial thought

The key idea is to insert the ***memory\_limit\_check*** procedure in the appropriate position in the above function chain.

I think it is appropriate for me to do ***memory\_limit\_check*** in the functions: ***\_\_alloc\_pages\_nodemask.***

Let us have a look at the algorithm of ***memory\_limit\_check*** procedure:

1. Traverse all the processes. In the traversing process, record the **uid** and memory usage(by getting the size of **RSS** of user process and multiply it by 4KB to get the actual physical memory size in bytes) in an array.
2. Check the memory limit entries in **my\_mm\_limits** one by one. If any user’s memory usage exceeds limit,
3. Find process with the **uid** by traversing all processes. In the traversing process, record the highest **RSS** and the correspondin**g tast\_struct**
4. Kill the process with highest **RSS.** The “killing” procedure is as follows:
5. Output the essential messages.
6. If the task is already exiting, don't alarm the sysadmin or kill its children or threads, just set TIF\_MEMDIE so it can die quickly.
7. Kill p
8. If any of p's children has a different mm and is eligible for kill, the one with the highest ***oom\_badness()*** score is sacrificed for its parent. (This attempts to lose the minimal amount of work done while still freeing memory)
9. Kill all user processes sharing **target->mm** in other thread groups, if any. They don't get access to memory reserves, though, to avoid depletion of all memory. This prevents **mm->mmap\_sem** livelock when an oom killed thread cannot exit because it requires the semaphore and its contended by another thread trying to allocate memory itself. That thread will now get access to memory reserves since it has a pending fatal signal.



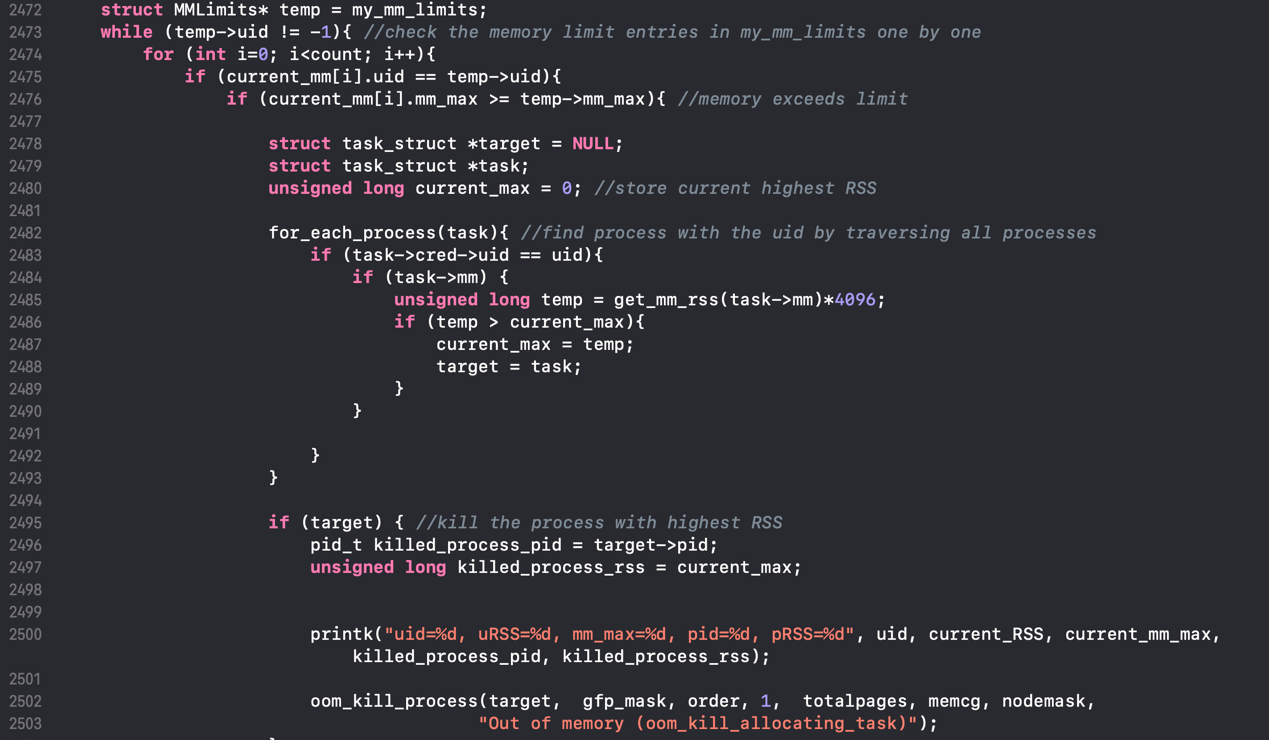


Figure 9 Implementation of my oom killer in **page\_alloc.c** (first try)

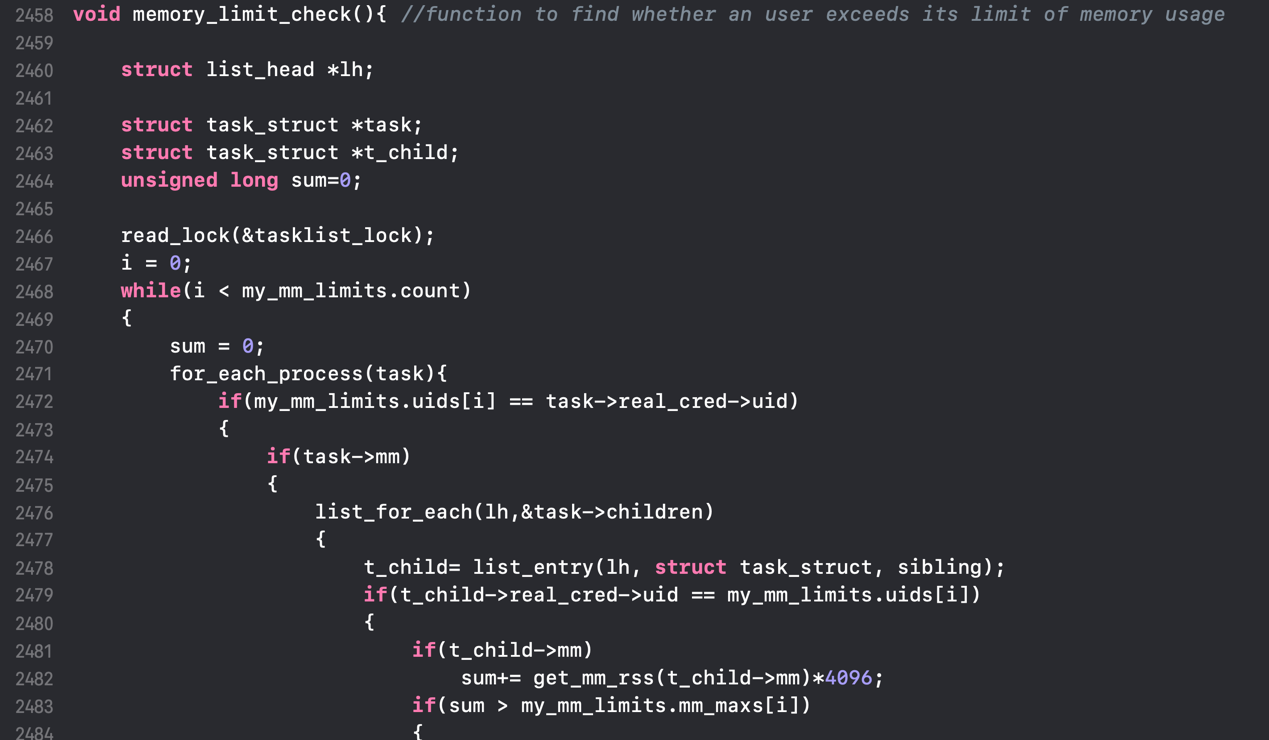
However, after testing, I find there are some problems with this method. It seems that when oom mechanism is triggered, ***\_\_alloc\_pages\_nodemask*** is not called.

* + 1. Find the correct way

**After studying Chan Young Lee’s code:**

I correct my algorithm:

1. Traverse all the existing entries in **my\_mm\_limits.** In the traversing process, traverse all the processes to find if there is a process **P** whose **uid** equals to **my\_mm\_limits.uids[i].** If so, traverse all the child processes of **P,** and sum thememory usage(by getting the size of **RSS** of user process and multiply it by 4KB to get the actual physical memory size in bytes) when meeting the process with the same **uid** as **P.** If this sum is greater than **my\_mm\_limits.mm\_maxs[i],** call ***mod\_oom*** to kill he process with highest **RSS.** The “killing” procedure is as follows:
2. Find process with the **uid** by traversing all processes. In the traversing process, record the **tast\_struct** of the process (**victim**) that is ready to kill.
3. Kill the **victim.** The whole process is similar to the one in **section 2.4.1**



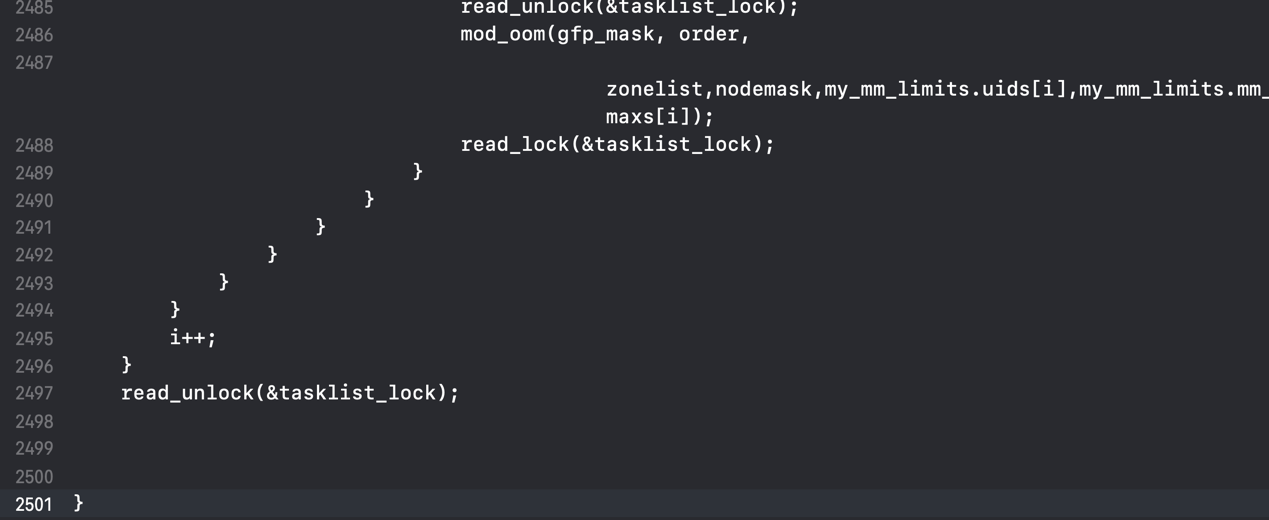
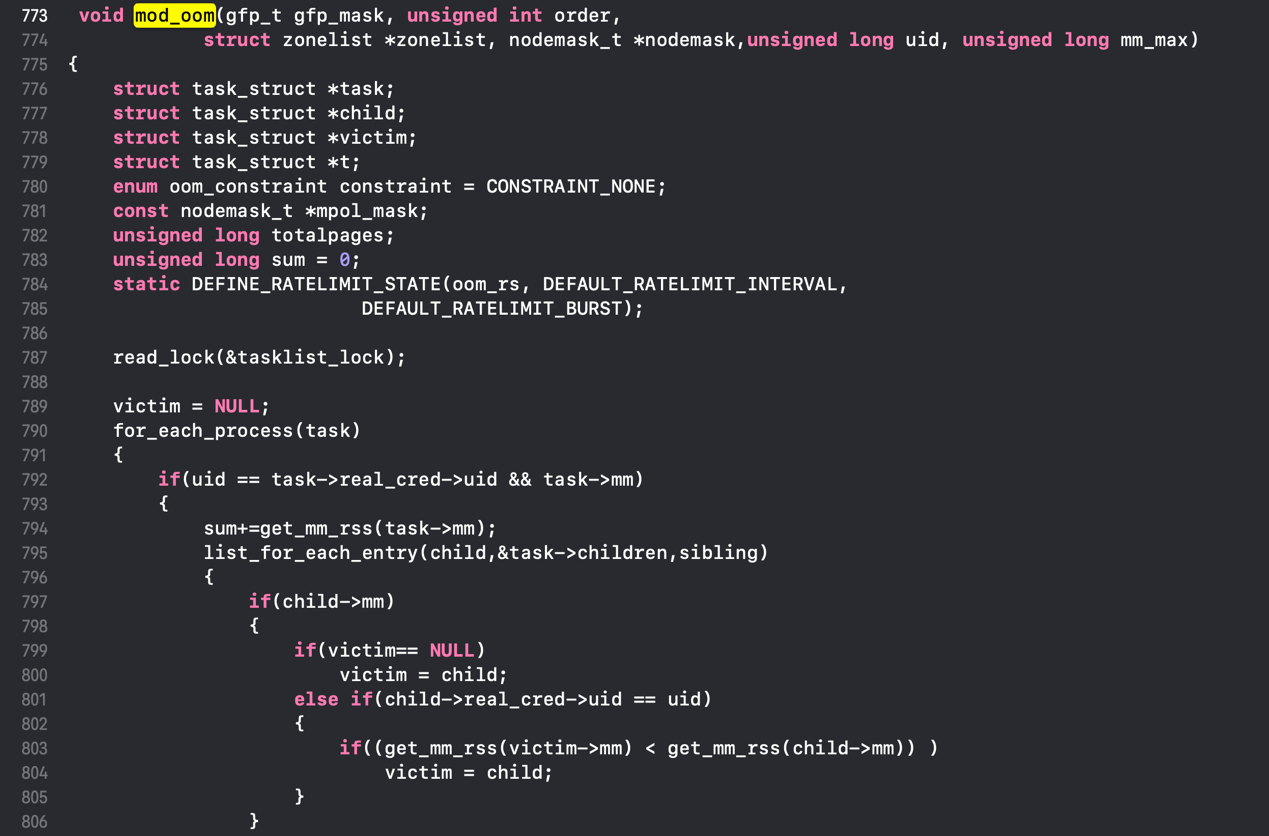
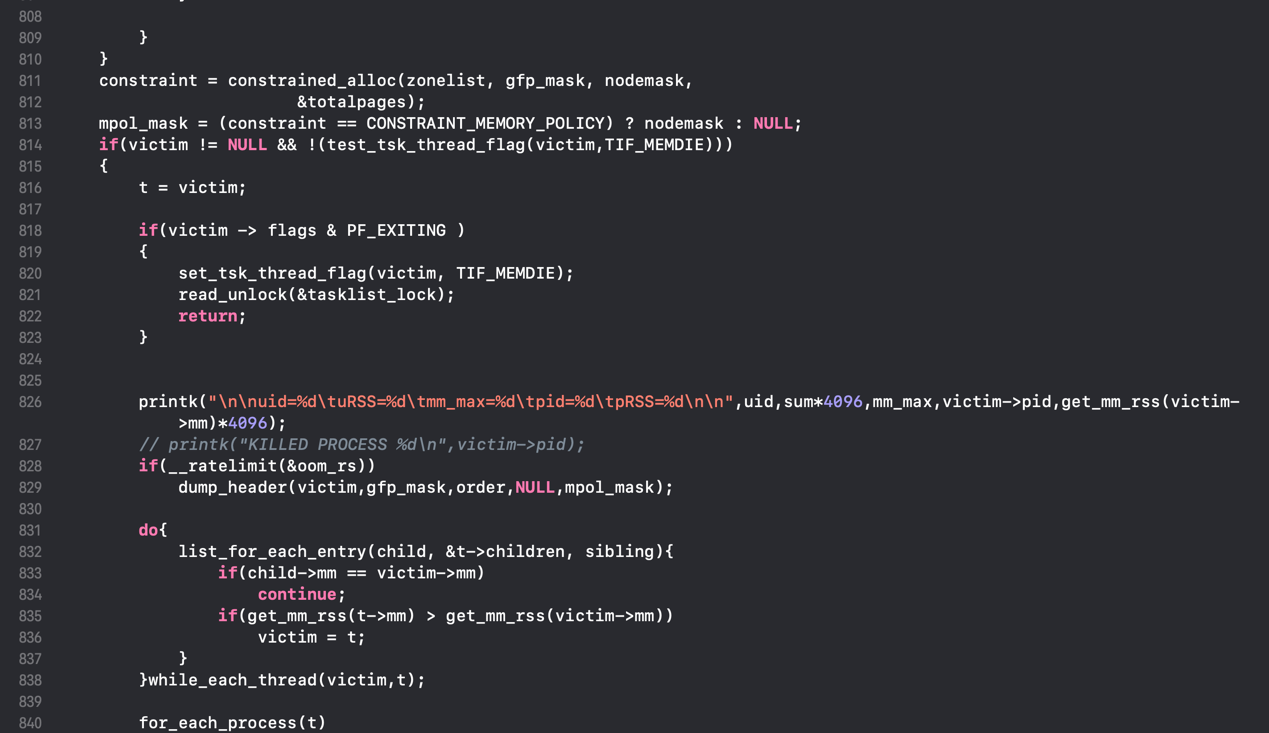


Figure 10 Implementation of my oom killer in **page\_alloc.c**

**Reference to Chan Young Lee’s code**





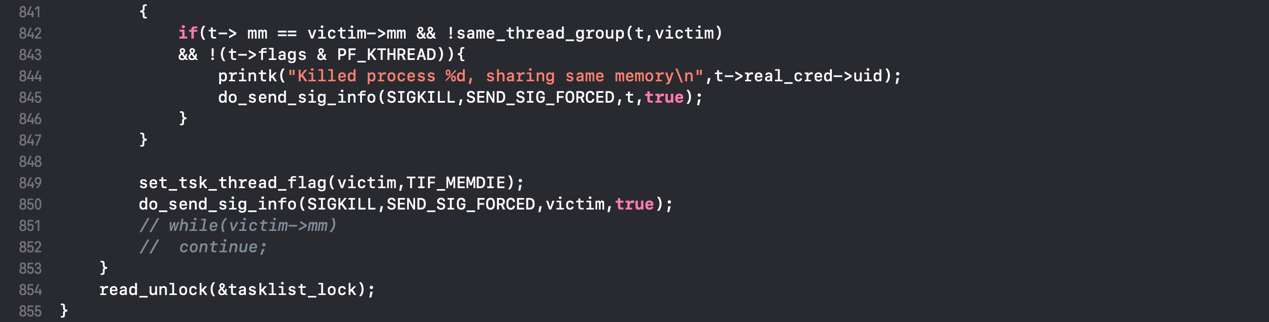




Figure 11 Implementation of my oom killer in **oom\_kill.c**

**Reference to Chan Young Lee’s code**

1. Conclusion
   1. Problems and solutions
      1. The definition, initialization and usage of global variables

I first tried to define the global variable by imitating the definition of **init\_task** and **task\_struct** as figure 12 and figure 13 show:

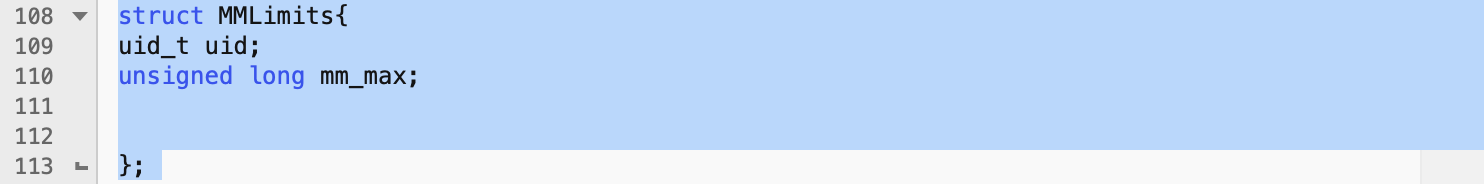


Figure 12. Definition of **MMLimits** struct (in **sched.h)**

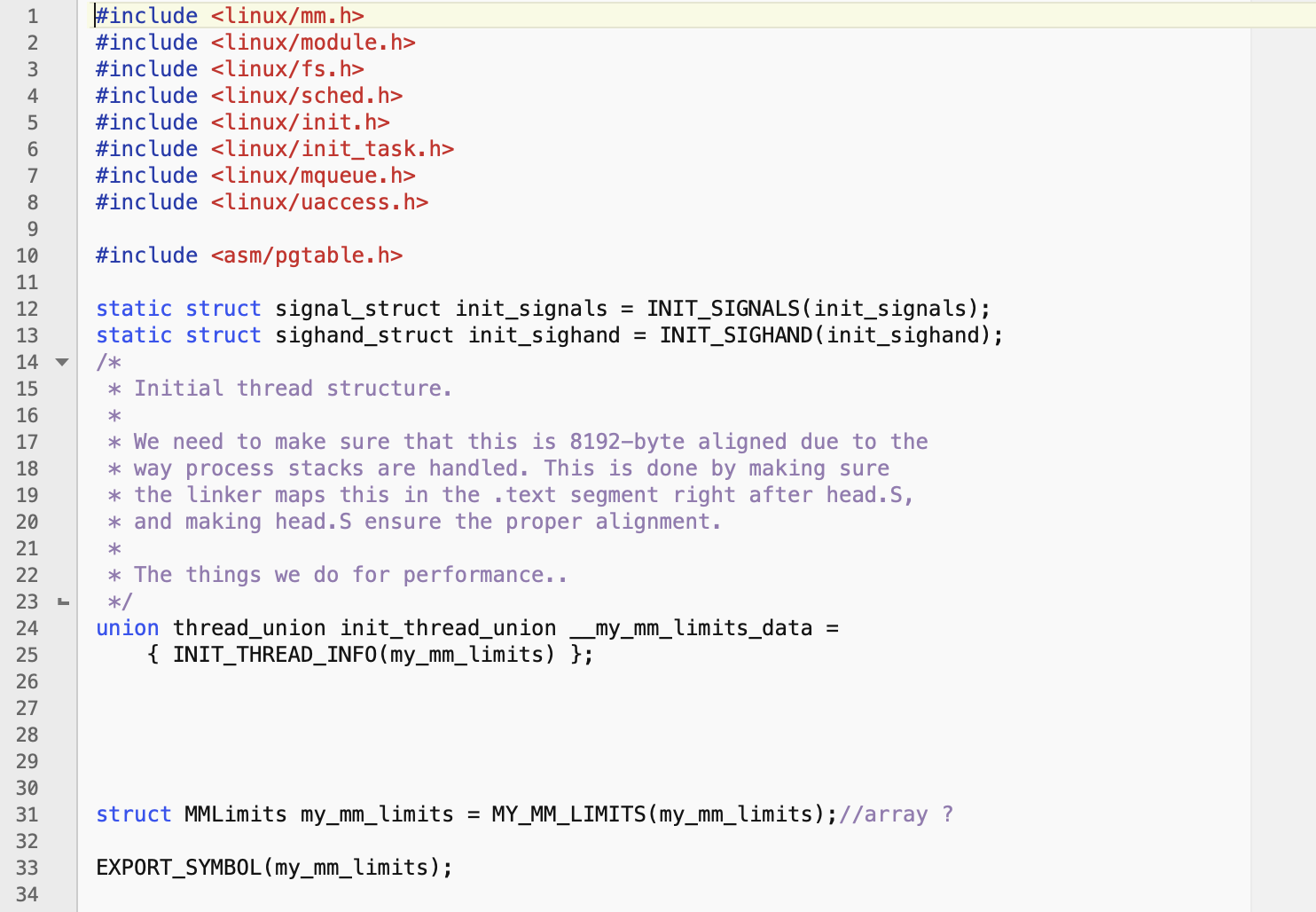
****

Figure 13. Definition of **my\_mm\_limits** (in **my\_mm\_limits.c)**

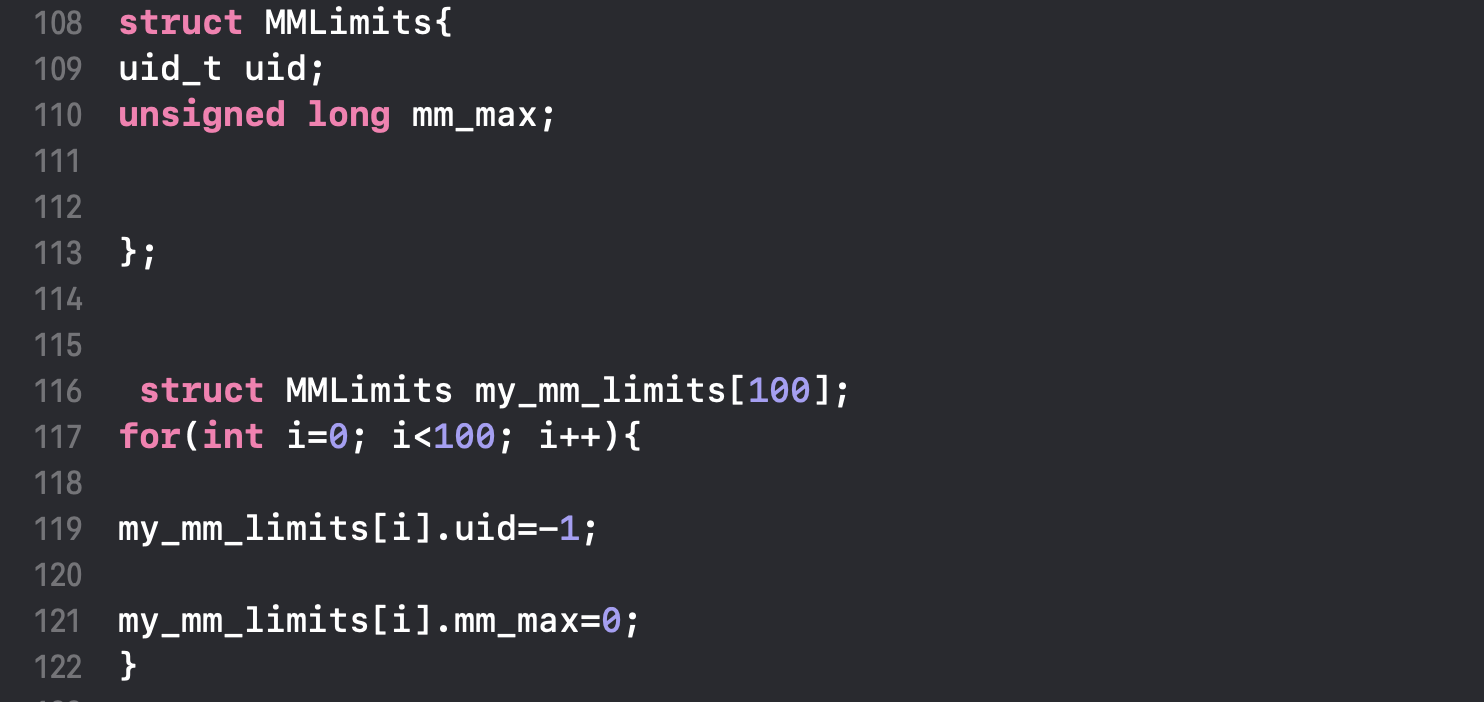
Unfortunately, the compiler cannot find the global variable. Because when you add new source files, you need to update the makefile correctly. Otherwise, the compiler cannot compile your new source file and find your new variable.

As the modification of the makefile is rather complex, I explore another way to add the code to the existing kernel source files to do the change, instead of creating a new source file.

Referring to the document provided by Zhang Chi, I find there are two ways to add the global variable:

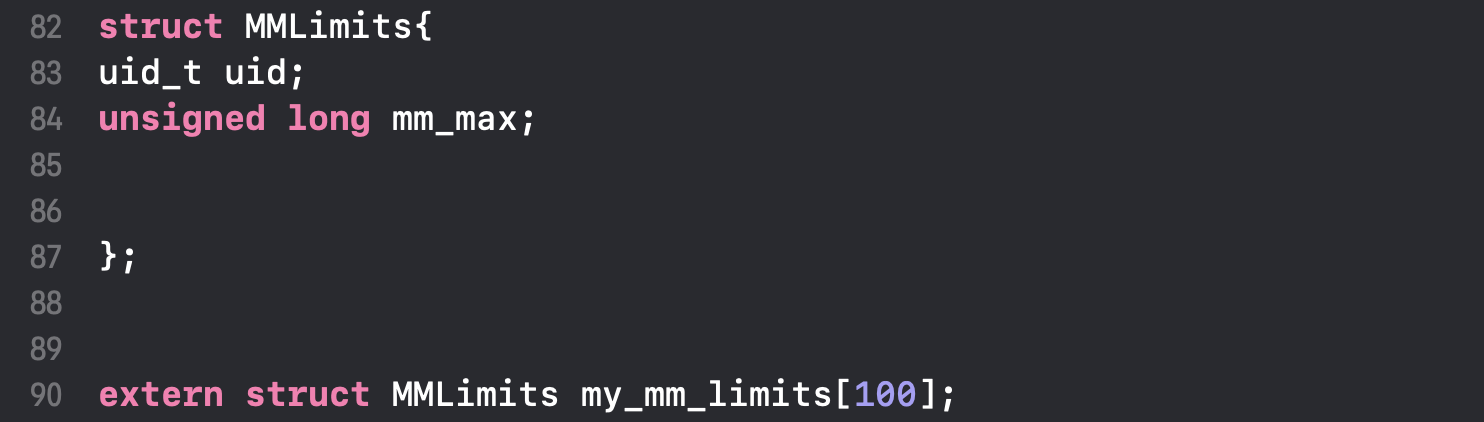
1. Add the global variable to the existing header file. By including the header file, we can use the global variable in other places.
2. Declare all global variables and structs when using them.

I first tried the first way as figure 14 shows:

****

**Figure 14** Definition of **MMLimits** struct and **my\_mm\_limits** (in **sched.h)**

**However, after modifying the header file, compiler will report an error. So I tried the second way** as figure 15 shows:

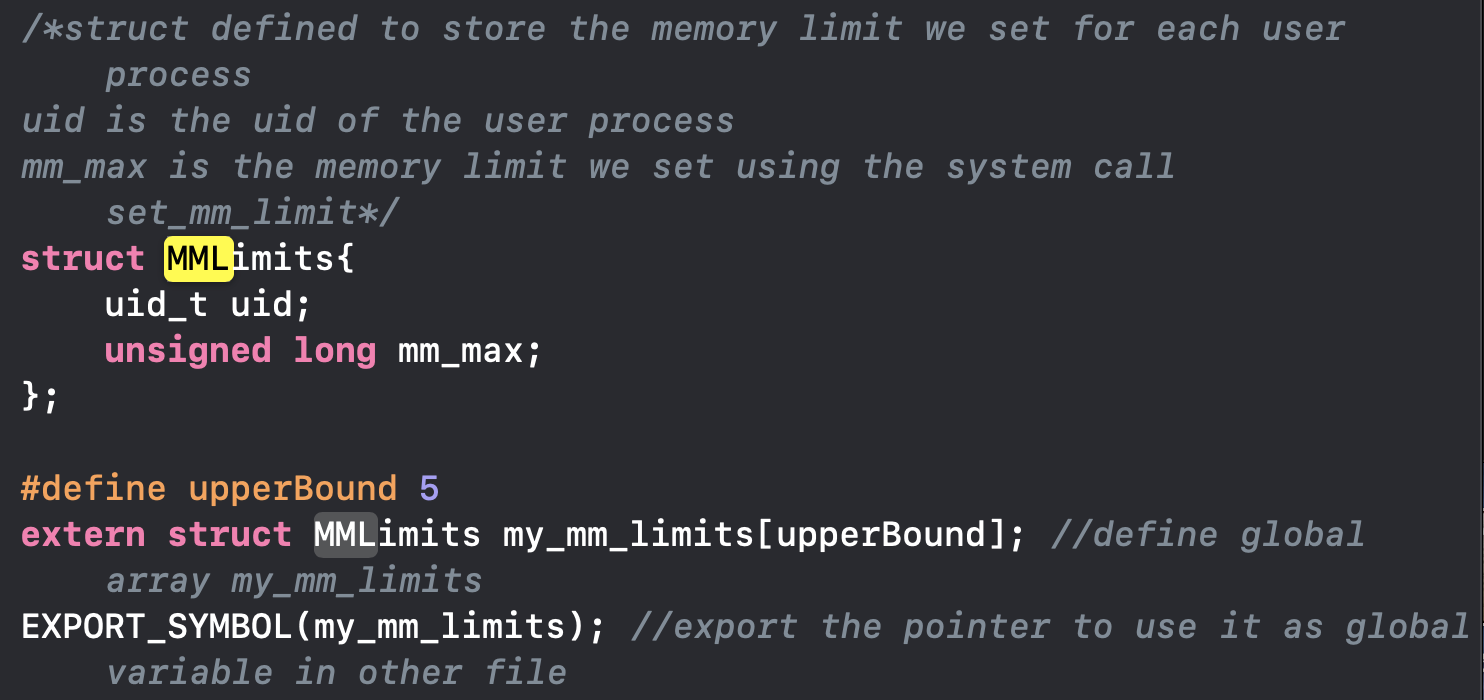


**Figure 15** Definition of **MMLimits** struct and **my\_mm\_limits** (in **memory\_check.c** and **set\_mm\_limit.c)**

**This time the compiler did not report any errors.. However, after a discussion with** Kar Chun Teong, I find there are some problems with my method:

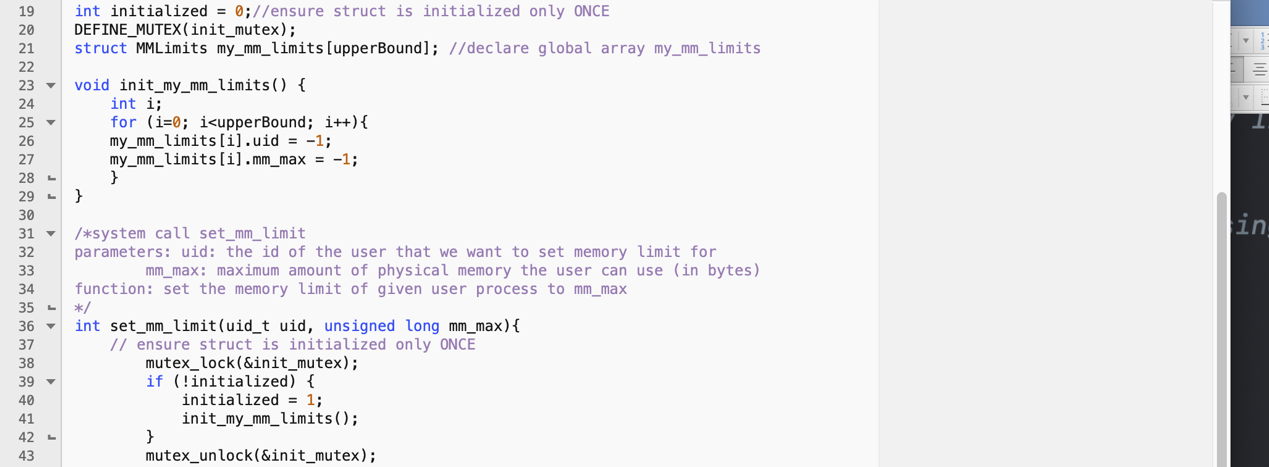
1. The variables may not truly be global
2. Can not ensure struct is initialized only ONCE.

So I correct my code as figure 16, figure 17 and figure 18 show:



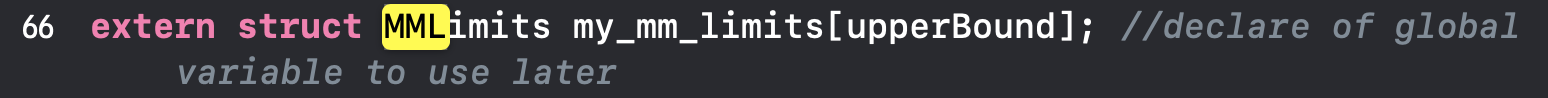
**Figure 16** Definition of **MMLimits** struct and **my\_mm\_limits** (in **mm.h)**

**Reference to Kar Chun Teong’s code**



**Figure 17** Initialization of **my\_mm\_limits** (in **set\_mm\_limit.c)**

**Reference to Kar Chun Teong’s code**



**Figure 18** Declare of **my\_mm\_limits** (in **page\_alloc.c)**

**Reference to Kar Chun Teong’s code**

Compared with the original code, the main differences are:

1. Using ***EXPORT\_SYMBOL(my\_mm\_limits)*** toexport the pointer to use it as global variable in other file.
2. The initialization is packaged in a function ***init\_my\_mm\_limits()***. Actually, the reason why **compiler reports an error is that I use *for* statement outside functions.**
3. Using an **init\_mutex** and a variable called **initialized** to ensure struct is initialized only ONCE.
4. Using a pointer ***(struct MMLimits\* temp = my\_mm\_limits)*** when using the global variables. This is related to 1).

However, after testing I find there is something wrong with my method. I can only successfully insert the kernel module once as figure 19 shows:

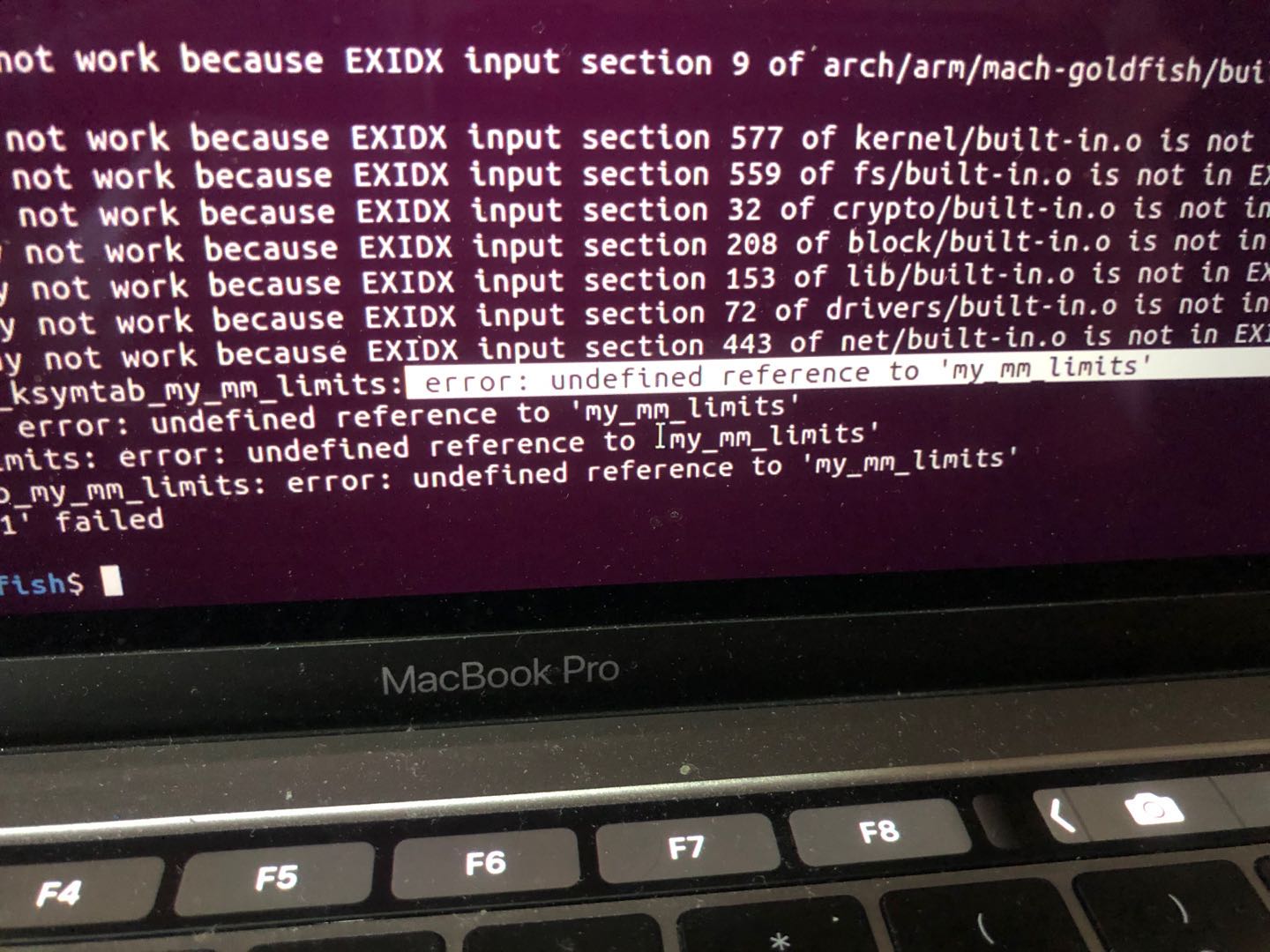
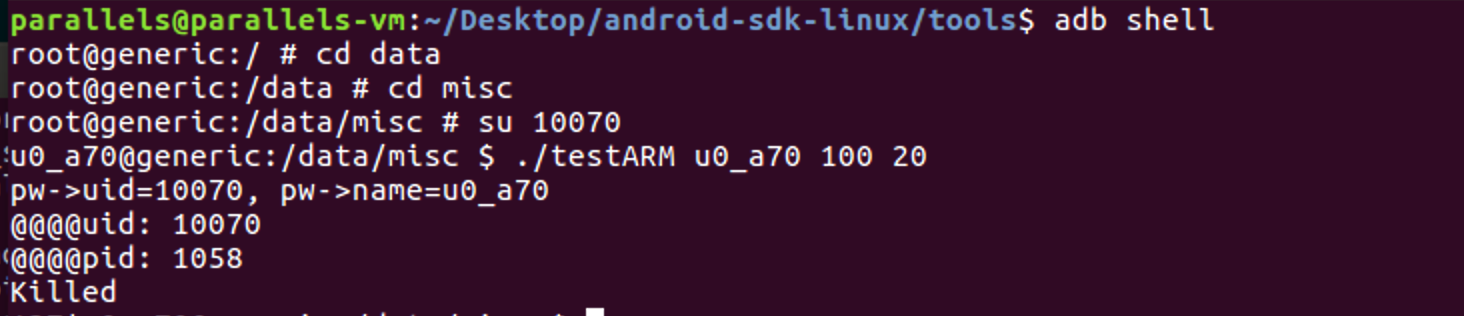


Figure 20 An error

Luckily, with **Chan Young Lee’s help,** I successfully define, initialize and use the global variable. The concrete description is in **section 2.2.2.2**

* 1. Test result analyze

Figure 21, figure 22 and figure 23 show the test result.



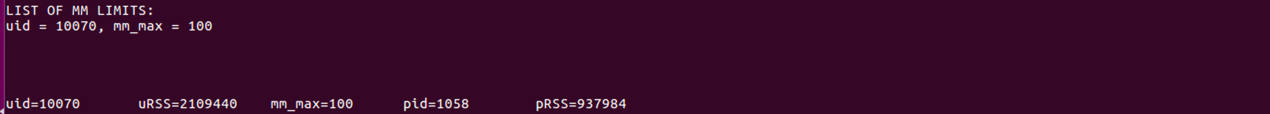
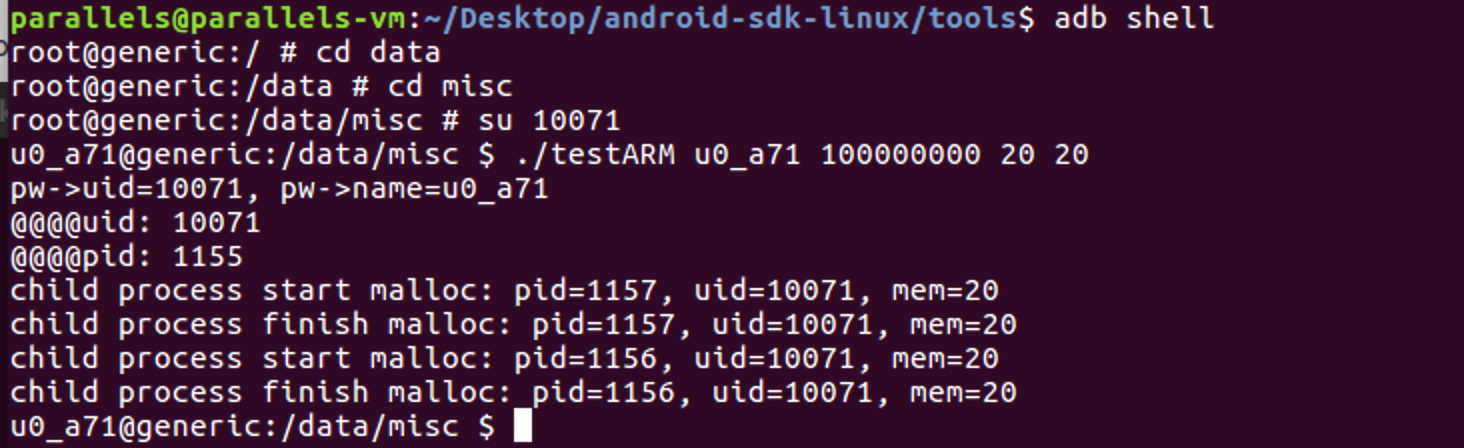


Figure 21



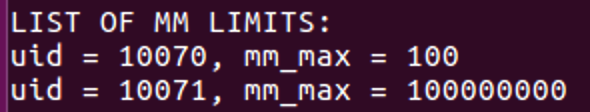


Figure 22



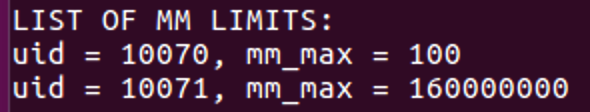


Figure 23

From the above figures, we can see:

1. No matter a user exceeds its memory limit or not, **LIST OF MMLIMITS** will be updated. And if we have not set the memory limit for the user, we will create a new entry in the **LIST OF MMLIMITS.** If we have set the memory limit for the user, we will update the old entry in the **LIST OF MMLIMITS.**
2. If a user exceeds its memory limit, the child process cannot be forked. The process with highest **RSS** and same **uid** is killed.
3. If a user does not exceed its memory limit, the system will allocate the memory for the child processes as normal.
   1. Achievements

From this project, I mainly achieve the following things:

1. Compile the Android kernel.
2. Familiarize with how information can be shared in the

kernel.

1. Familiarize with Android OOM killer.
2. Implement a new OOM killer that support memory limit for applications.
3. Add the new system call directly to linux kernel.

4、Reference

1、**Kar Chun Teong’s code in section 3.1.1**

2、**Chan Young Lee’s code in section 2.2.2.2 and 2.4.2**