Implement RCU in xv6

S.Maccarrone, N.Gallo

September 23, 2022

Contents

1 Project Data **2**

2 Project Description **3**

2.1 Design Implementation . . . 3

3 Project outcomes **4**

3.1 Concrete outcomes . . . 4

3.2 Learning Outcomes . . . 4

3.3 Existing knowledge . . . 4

3.4 Problems encountered . . . 5

4 Honor Pledge **7**

1 Project Data

- **Project supervisor**: Vittorio Zaccaria
- **Authors**:

Last and first name	Person code	Email address
Maccarrone Salvatore	10740276	salvatore.maccarrone@mail.polimi.it
Gallo Nicolas	10603572	nicolas.gallo@mail.polimi.it

- **Tasks organization**: For both of us this project was a learning experience and in order to move forward with the work we needed to both be at the same level. We did not split the work between us, but instead **we worked together in every part of the project**. This helped us to have a better understanding of how xv6 behaves.

2 Project Description

The project is about the implementation of RCU locking for some kernel data in xv6. During the initial phase of the project we decided, together with the advisor, that we would be implementing **RCU locking on the processes list managed by the xv6's Kernel**. It is important for the AOS course because we want to implement an advanced lock technique that we have seen from a theoretical prospective in a, tough not super complex, real operating system

2.1 Design Implementation

In xv6 the list of process is statically allocated as an array of size 64, meaning that only 64 processes can be active at the same time. The main idea of the RCU scheduler is to keep at thee same time two copies of the same item, in this case a process, a the same time. The old and the new version of an item coexist for a certain period, but with the constraint that anyone that access the list can see only one of them, either the first or the one, not both.

To fulfill these requirement we decided to convert the array of all processes into a **linked list**. With this new data structure xv6 can now, at least theoretically, manage an infinite number of active processes, since there are no constraints on the dimension of the list. We should nevertheless specify that even if the list can now grow indefinitely, the previous constraint of a maximum of 64 process does not disappear due to how xv6 manages the memory assigned to each one of them (this does not imply things cannot be changed but a bigger effort would be needed and we would have exited the scope of the project).

The list_proc.c file manages the behavior of the linked list and contains all the function needed to interact with it, to add, delete, update a node of the list. Note that these list was not written for a generic purpose, but was instead specifically implemented to manage a list of process. We decide to not go on a generic implementation for a practical and a debugging reason.

Note that the list_update_rcu() and list_del_rcu(), that respectively update and delete an element from the list, use a double pointer on the inputs variables. We decided to go through this implementation because we chose to bring outside list_proc.c the deletion phase. In fact, deletion time is critical for RCU (in RCU terms, **reclamation phase**) since an item can be finally deleted and its memory freed after a grace period has elapsed.

In the sched() we disabled the panic traps since the where all related to locks no more used with the implementation of RCU. We implemented all the functions that, according to the literature we found, allow a basic implementation of an RCU scheduler.

3 Project outcomes

3.1 Concrete outcomes

We managed to have a full working xv6 with a list of process scheduled with RCU. The working OS can be found here. However this branch contains more a beta version than a full working one. This branches allows to work normally with **xv6**, however it runs with only one CPU. There is no multi-threading, it is easier then to check for the grace period. In this case The synhronize_rcu() does nothing at all. The reason is because when a writer is scheduled and modifies the list, it can immediately delete an items, without waiting for any grace period, since it is sure no readers are actually reading the old version of the modified process and no of course no other writer are active are the moment. The **readers** are not interruptible during their critical section, so when a **writer** is running we are sure all the reader have already read what they needed.

Other thing to notice is what happens in the function rcu_assign_pointer() at line 101. A writer should not be interrupted while is updating a list's pointer and to avoid so this operation should be done **atomically**. The reason for this is to avoid a reader to access something in that is in the middle between an old and a new version. To force these behavior we used the **__atomic_store()** function. However due to other problems we encountered we did not test it and we do not know if the functions behave as we expect and if it actually works.

The final version we were working at is this one but we did not manage to make it work. The kernel boots, but it breaks. We discuss the reasons in the section 3.4.

3.2 Learning Outcomes

- **Nicolas**

  It was the first time I have seen kernel code. I have studied many times how a system call works and what it is used for. Now I have seen how it could really works in practice and I also learned to write it. I deepened my knowledge on the heap memory management. Now I understand what there really is behind the idea of allocating and freeing memory. I have learned to use a powerful debugging tool such as **gdb** and also to use git. I discovered that I prefer to program at low-level and that I might pursue my studies in the OS field.

- **Salvatore**

  I am delighted to say that I learned how several part of xv6 works in detail. Dealing with its kernel exposed me to several programming concepts that I have never seen in a detailed way previously. Among these concepts there are: use of macros to implement specific functionality (they can work as templates) or just for debug purposes; **compiler directives** to emit specific code (fences or spinlock related), this in turns shed a light on RISC-V assembly; **K&R heap allocator**, a small but powerful tool; **sleep/wakeup** mechanism based on channels.

3.3 Existing knowledge

We both did the bachelor on a different universities. However it helped us, except for a basic operative system course:

- **Nicolas**

  A course on C and shell language, where I learned to interact with the OS through a terminal and write basic shell scripts where fundamental to work on this project. It also helped me to follow a crash course on git hel by the POul association. I would also say that the course on computer security helped me on the debugging phase.

- **Salvatore**

  The crash course held by POuL on **git** was fundamental, but also hints on how **docker** works were super useful especially during the first phase of the project. Courses about computer security (Computer Security, Offensive and Defensive Cybersecurity) really eased my time with **gdb** and they also provided me a basic understanding of how an **heap allocator** should work.

It was for both of us the first time we got the opportunity to really modify an OS and it was not very easy especially at the beginning. We found very useful to get confident with xv6 the lab course of the washington.edu that we know you also cited on the lectures. We think it might be interesting for the course to allow all the student to get their hands dirty with a real operative system. So we would suggest that making some practical example with a 'simple' os like xv6, as they do in Washington might help a lot. It was easier for us to understand and remember what was done on class after working with xv6.

3.4 Problems encountered

First thing we noticed was that there was not a way to allocate memory dynamically at kernel level. Everything in xv6 at kernel level was allocated at compile time so we had to be very careful when we had to modify memory addresses. The first problem we encountered was that we needed to dynamically allocate memory to implement a linked list. The only way to to so was to use the kalloc() function, that allows to allocate a 4KB page. However, it was not practical to use, since assign a a page for each process would have brought to a complete and quick waste of memory space. To solve the problem we used the malloc writted by Kernighan & Ritchie that we found in the book The C Programming language at page 187. However we had to make some modifications since this malloc was designed for user space. It was enough to replace the sbrk() function with a kalloc() to make the code compatible with the kernel space. The code written can be found here.

A big difficulty we encountered was related to synchronize_rcu() at line 104 of list_proc.c. We needed to find a way to calculate the grace period. All the solution we found in the literature were not feasible with xv6, then we came out with a solution that works with the following assumptions:

- there can not be more than one writer inside synchronize_rcu();
- readers do not acquire lock in their critical section.

We can then check then for each CPU if it has done a context switch or if it is in idle (the are trying to acquire a lock). Checking for these allows us to be sure that all the readers have exited their critical section.

The other problems are related to **RCU** itself. For the algorithm purpose it is strictly important to separate readers from writers, since the firsts can access the shared resource without locking it while the seconds one con do it only once at time and once they modify an element the must wait for a grace period to elapse before deleting the old version, ensuring every reader have at least a pointer to **either the new or the old version of the resource**. Our approach was to see as a reader anyone that tried to access the shared list of processes and whenever a process needed to modify the list, see it as a writer. However that lead us to some consistency problems. The first one was related to the **sleep() and wakeup()**. A wakeup() always follows the sleep(), however the wakeup() is quicker than the sleep, then it happened that even if a sleep() was called, the OS did not change the state of the process in time, that the wakeup() was called and did not find any process on a sleeping state. These is caused by the fact that two threads that are calling a sleep and a wakeup() are both reader, because they at first access the list to find the process, then they become writers when they have to change its state. To solve it we force the wakeup() to wait the sleep() to complete, before looking for a process. To force this we needed to implement a lock. We did not like the solution we implement, because the idea of RCU is not to use locks, however we did not find a better one.

The main problem we encountered is the fact that at some point the init process start becoming a zombie process and this is not something we want to happen. It was happening that when exec() was called by init it did not find everything it needed returning -1 causing init to exit and making the os cash with a kerneltrap or with a panic(ilock no type). Debugging with gdb we realized that for some reason at some point the inode of the init code starts to have type=0 and that cause the the kernel to panic. Unfortunately we got stuck in here and we did not find a solution.

We want to point out that we realized that applying RCU on the list of processes might lead to some inconsistency. We show here an example. The **scheduler is a reader**, since it access the list of processes looking for one in a RUNNABLE state. Notice that in xv6 every CPU has its own scheduler that concurrently access the list. However, once the scheduler finds a process to run, it has to change its state from RUNNABLE to RUNNING, so it **becomes a writer**. For how RCU works, once a scheduler chooses a process, it to wait for the **grace period** to elapse before updating the list. Meanwhile another CPU could choose the same process of the first CPU, since this one would see the old version of the list. Once the grace period of both CPUs elapses and since the write operation are scheduled in a FIFO policy, they both update the list and there then will be two copy of the same process available and they also will be both running. This scenario will bring to big inconsistencies. Think if two CPUs pick up the init process, we will have two copy of it and this breaks the logic of the kernel.

We discussed a lot about possible solution. We thought to see the scheduler has writer from the moment it access the list, however we realized that all the functions that allow to access the process list behave as the scheduler does, so we would ended up on having only writers and it would have been like going back to the original behavior of xv6. We also did not go for this solution because the all thing of **implementing RCU is about eliminating locks**, at least most of them, and applying this solution was not really in harmony with the RCU's goal.

4 Honor Pledge

We pledge that this work was fully and wholly completed within the criteria established for academic integrity by Politecnico di Milano (Code of Ethics and Conduct) and represents our original production, unless otherwise cited.

We also understand that this project, if successfully graded, will fulfill part B requirement of the Advanced Operating System course and that it will be considered valid up until the AOS exam of Sept. 2022.

Group Students' signatures