**ASSIGNMENT-3**

**Group 11**

**KARTIKEYA SAXENA 180101034**

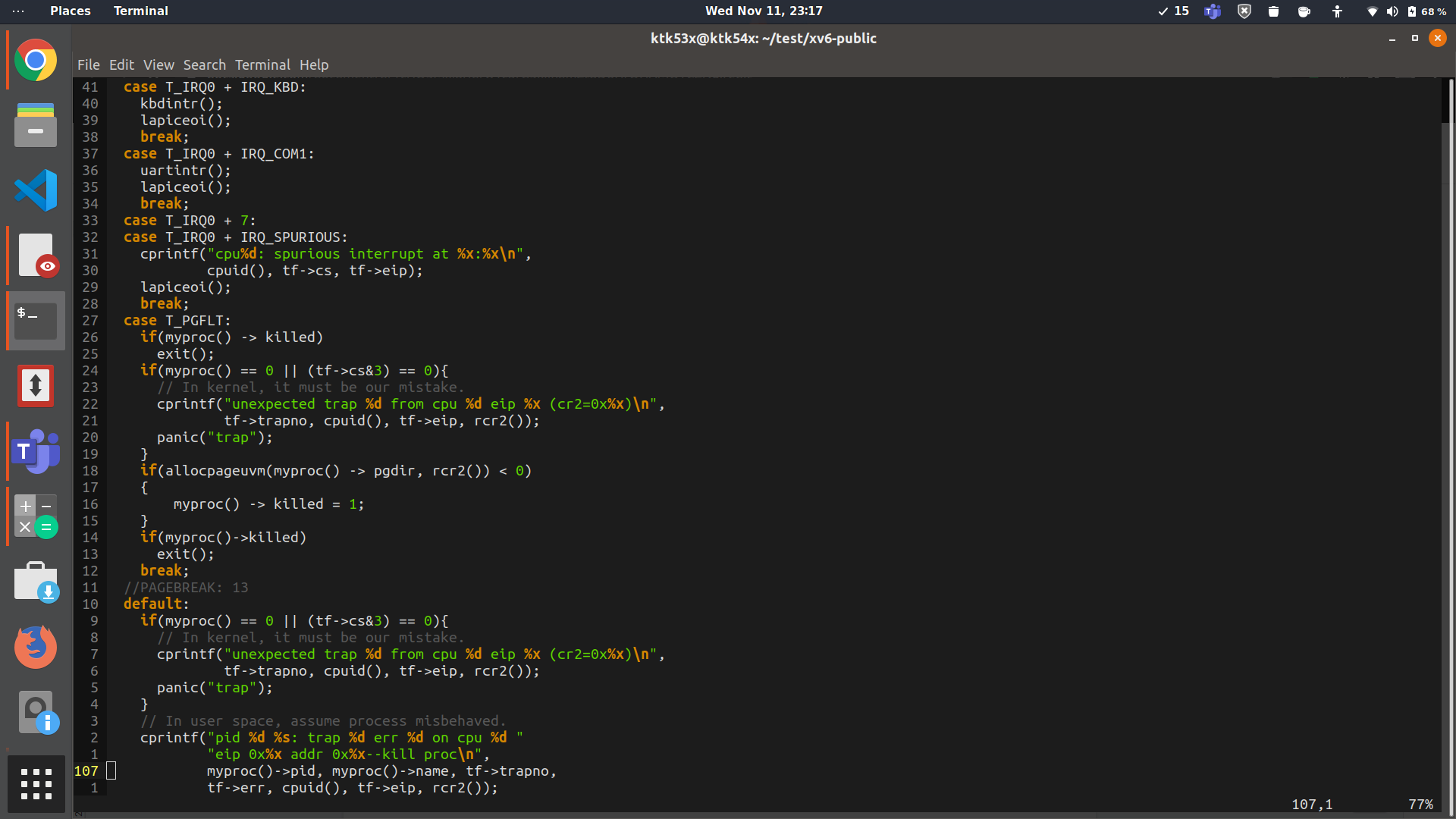
**KUSHAL SANGWAN 180101096**

**MILIND PRABHU 180101091**

**ADITYA PATIL 180101004**

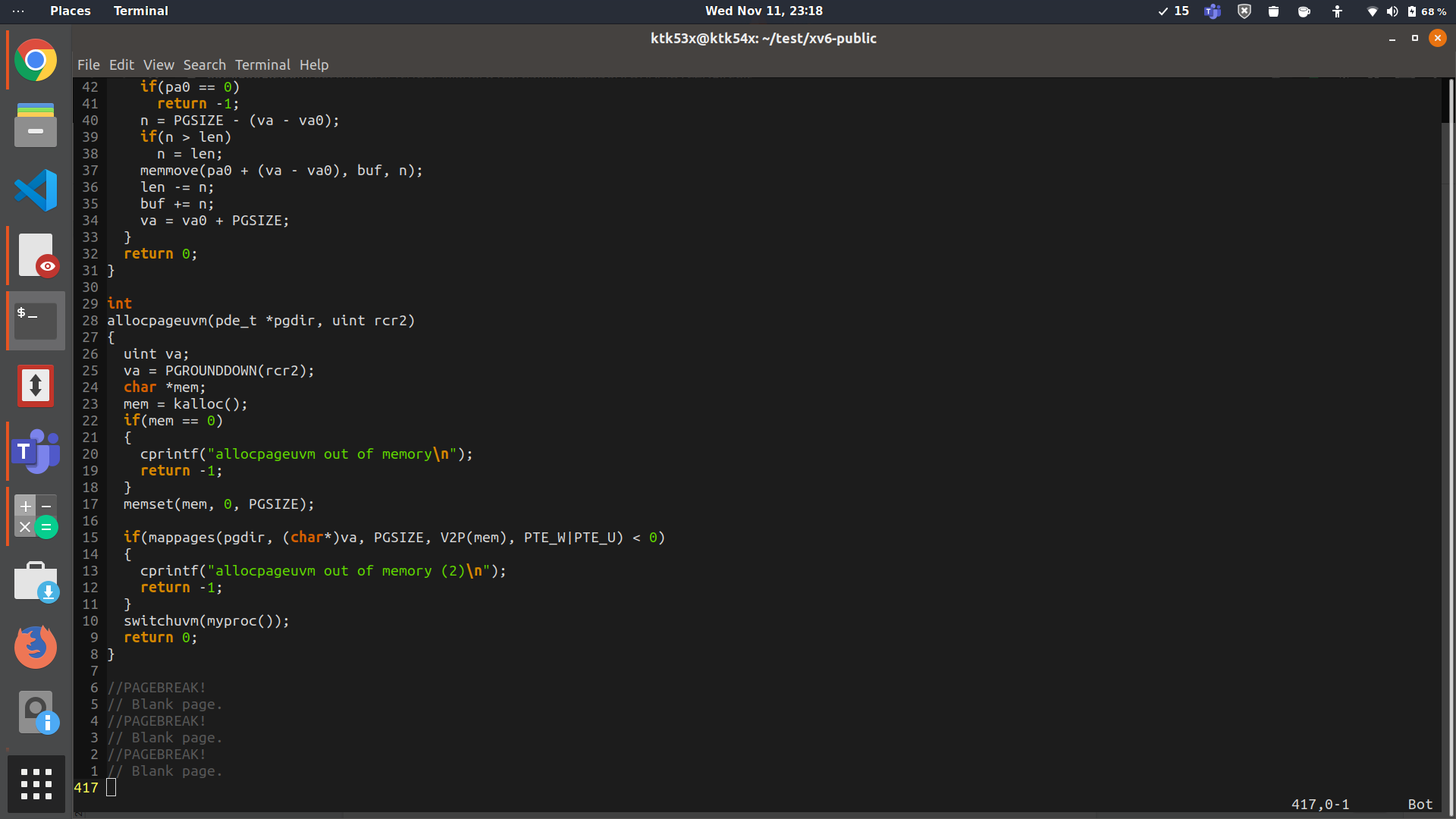
**Part A - Lazy Memory Allocation**

**i) trap.c**



1. Added a case in the switch block of Trap handler to determine what action to take when Page Fault Occurs - “case T\_PGFLT” (T\_PGFLT is Trap 14, and the corresponding term was found in trap.h.
2. If the process has already been killed, or had been called by the Kernel, we do not call the Lazy Allocator function from vm.c.
3. If none of the above two happen, the Lazy Allocation function, “**allocpageuvm”**  is called which will then, assign memory to the process at runtime.
4. If this returns an error, the process is killed; else, controlled is returned to where the page fault had last occurred.

**ii) vm.c**



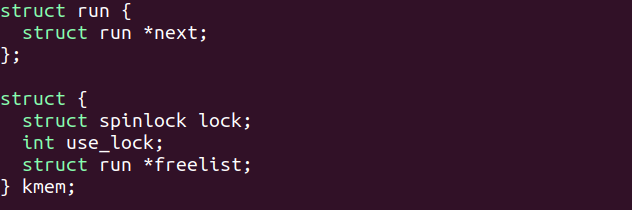
1. The pointer to the page table is passed to the function along with the result of the assembly function rcr2(). rcr2() returns the virtual address at which the page fault occurred.
2. This Virtual address is Rounded Down so that the unsigned integer variable **va** now stores the address of the beginning of the page.
3. kalloc() is called, this finds a free frame in the physical memory and then the pointer to the beginning of this frame is returned and stored in char\* mem. kalloc() returns 0 if no free memory is found and the corresponding error is checked for, and printed if it occurred.
4. For the next PGSIZE (which is a xv6 constant for Page Size) locations, we make these 0 as a good initialization practice, using memset(mem, 0, PGSIZE)
5. Finally, this new free memory at \*mem is mapped to the address at va, and the entry is added into the Page Table that was passed initially; permissions for this being a user and writable memory are passed accordingly. Finally, the error corresponding to mappages was checked for.

**Q/A:**

**Q-1**- How does the kernel know which physical pages are used and unused?

**Ans:**

The kernel tracks free pages available using the struct run data structure which is defined in kalloc.c . The structure stores a pointer to the next free page thus forming a linked list of free pages. The pointer to the next free page is stored within the page. The head of the linked list, freelist, is maintained by the kernel in struct kmem. All physical pages are appended to this linked list whenever they are initialized or are freed. Whenever the kernel wants to allocate a free page it traverses the linked list, finds a suitable page to allocate and returns the virtual address of the page.

****

Q-2- What data structures are used to answer this question?

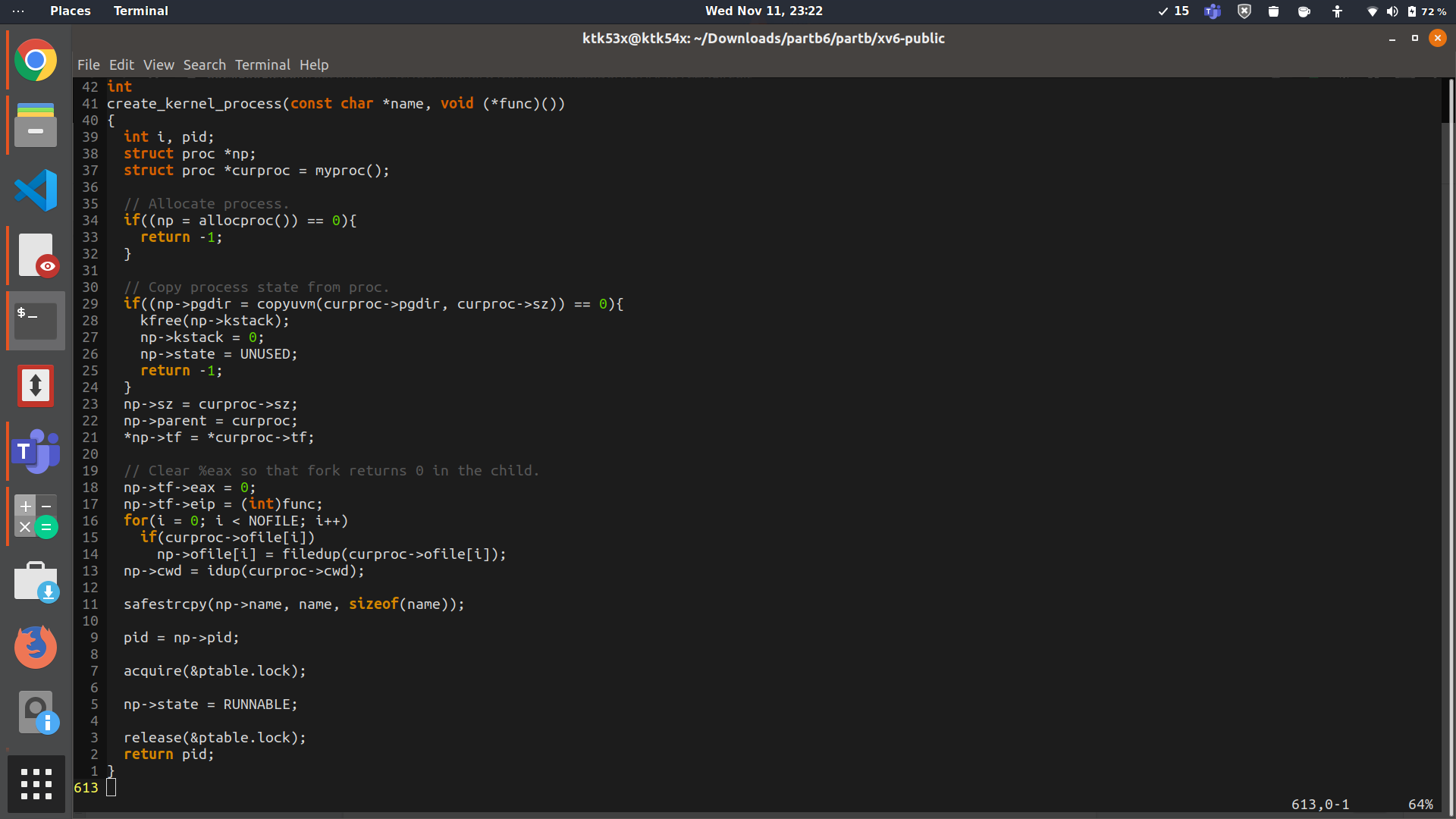
**Ans:** A linked list data structure is used to store the list of free pages as explained above.

Q-3 - Where do these reside?

**Ans:** The code for these structures reside in kalloc.c. The structures themselves reside in the kernel memory.

**Part B - Paging (Swap In / Swap Out)**

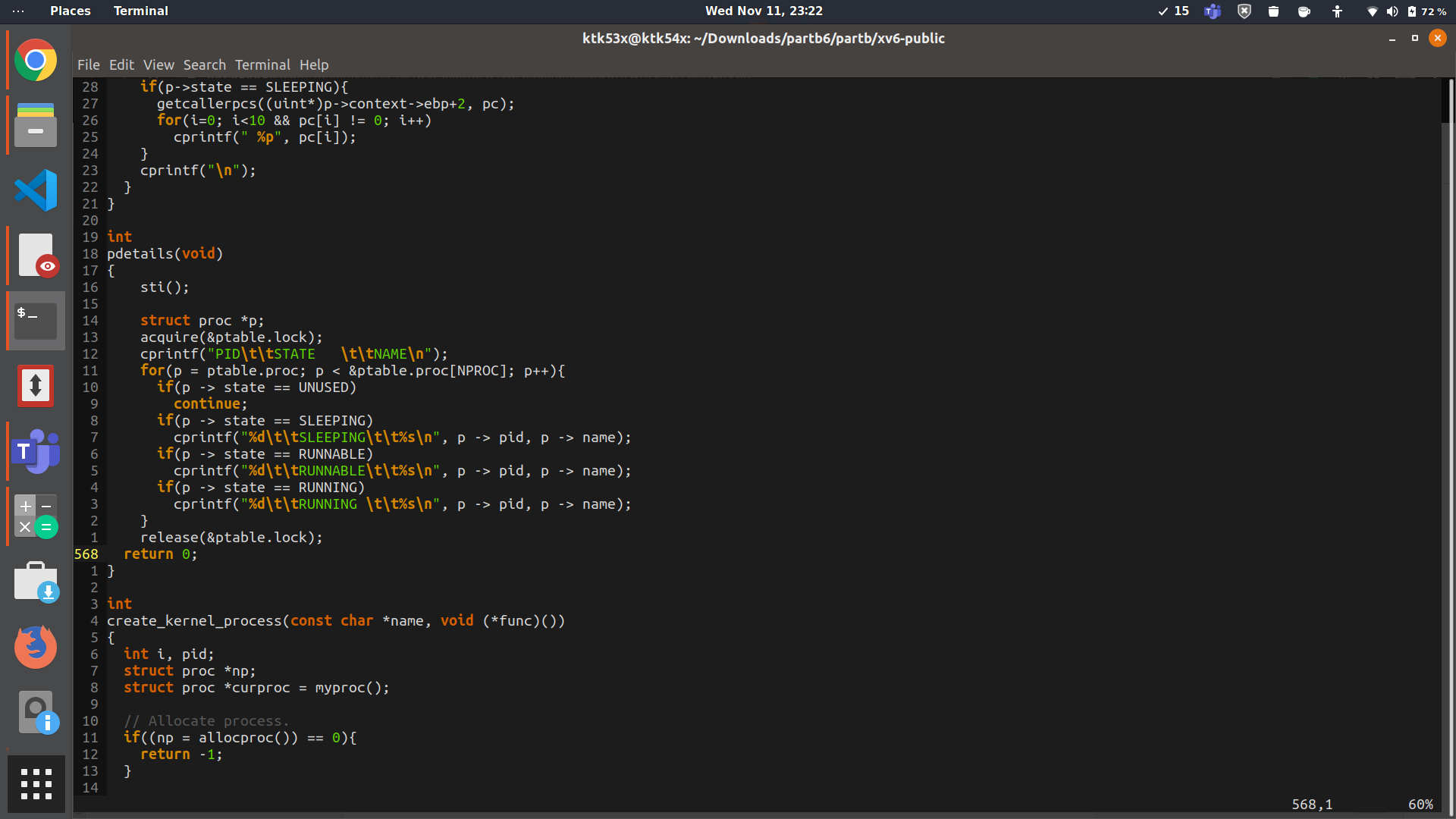
Task1-Making kernel process



Create kernel process is a system call in which is somewhat like fork() call except that we point the eip in trapframe to the function pointer passed as argument of this call instead of keeping it like in initial case(where it points to instruction after fork()) , and also instead of copying the same in name as parent process in safe strcpy, we now pass it to the new name like “sin” or “sout”.

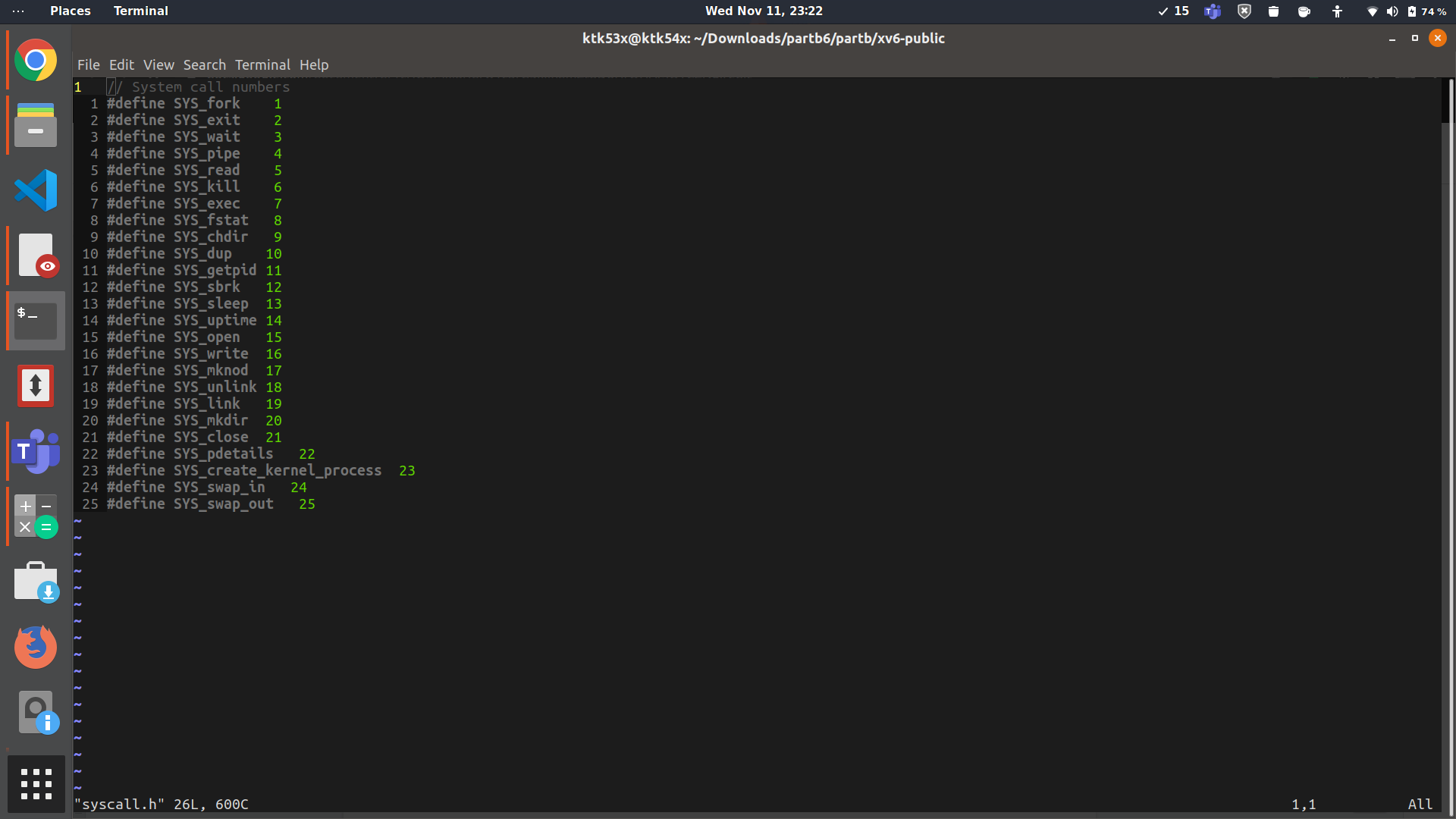
me to the function pointer passed as argument of this call instead of keeping it like in initial case(where it points to instruction after fork()) , and also instead of copying the same in name as parent procenter passed as argument of this call instead of keeping it like in initial case(where it points toss in safe strcpy, we now pass it to the new name like “sin” or “sout”.

me to the function pointer passed as argument of this call instead of keeping it like in initial case(where it points to instruction after fork()) , and also instead of copying the same in name as parent process in safe strcpy, we now pass it to the new name like “sin” or “sout”.

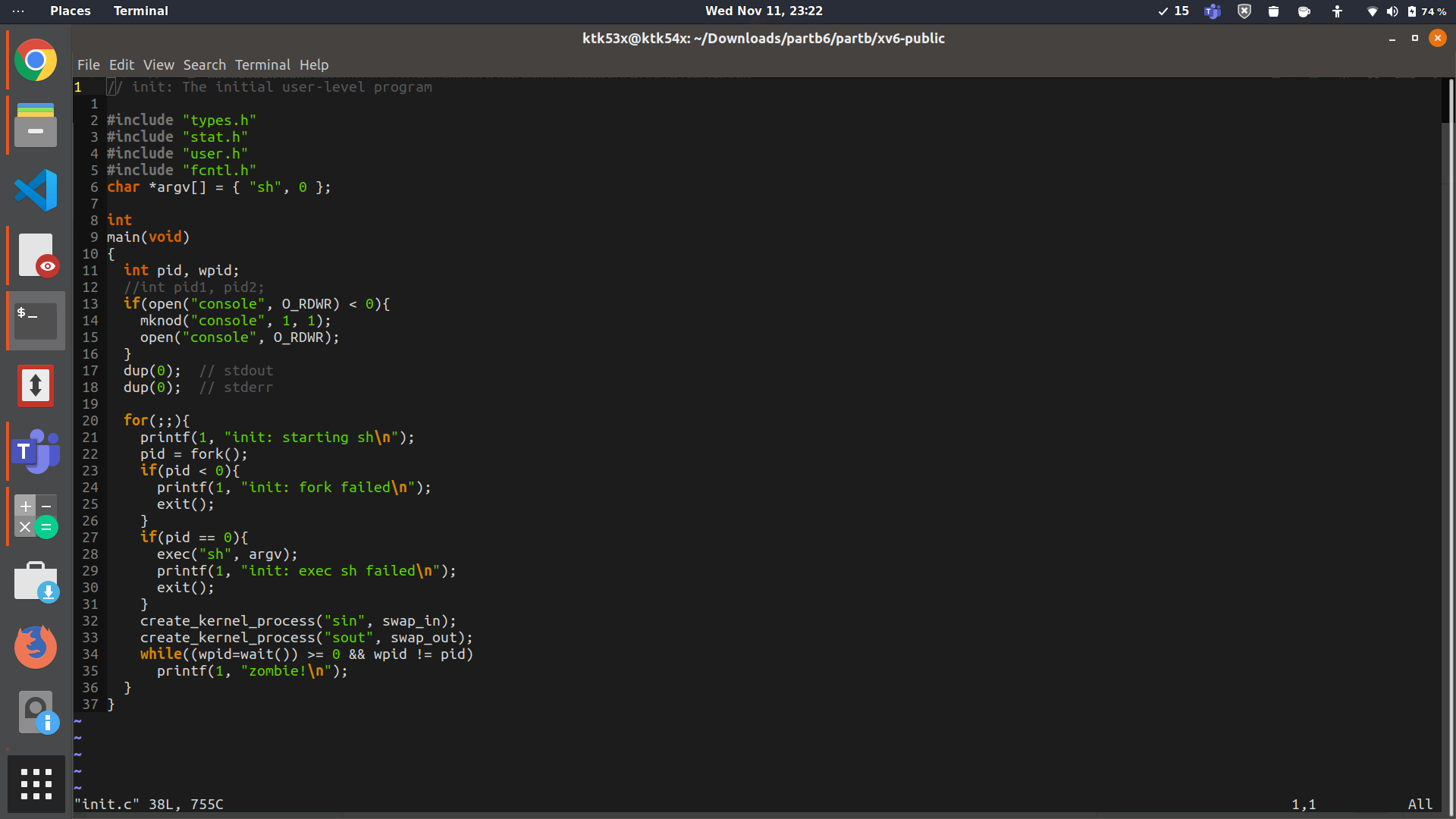


Also as we are creating the a nice.c user program is created to check out whether the processes are being added to the process queue or not . In nice.c , details system call is done to get the process details from ptable and then print them.

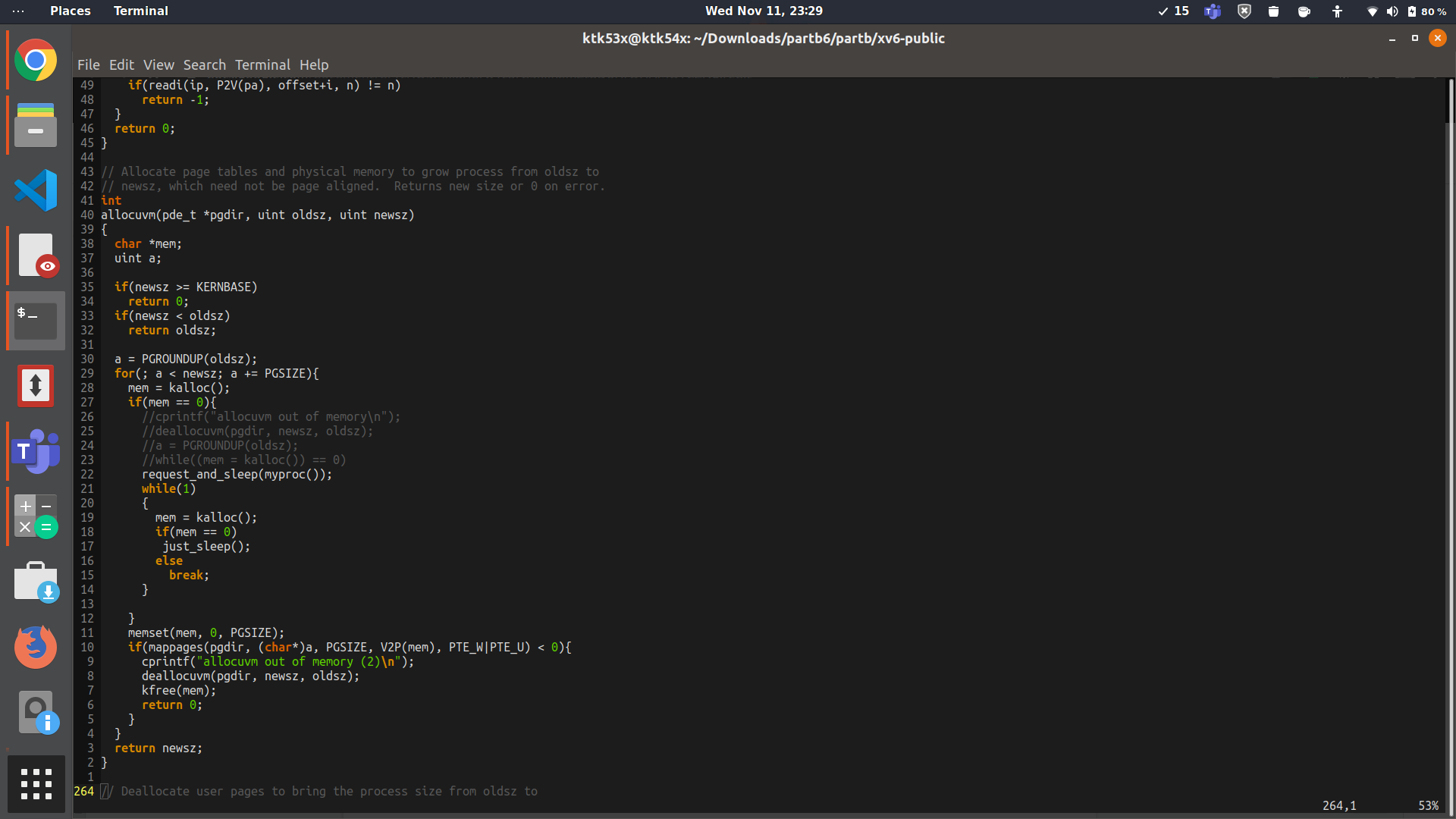
Also to check whether these two processes created are really kernel processes or not , we have tried to acquire kernel structures like ptable lock from these functions which can only be accessed in a kernel process .



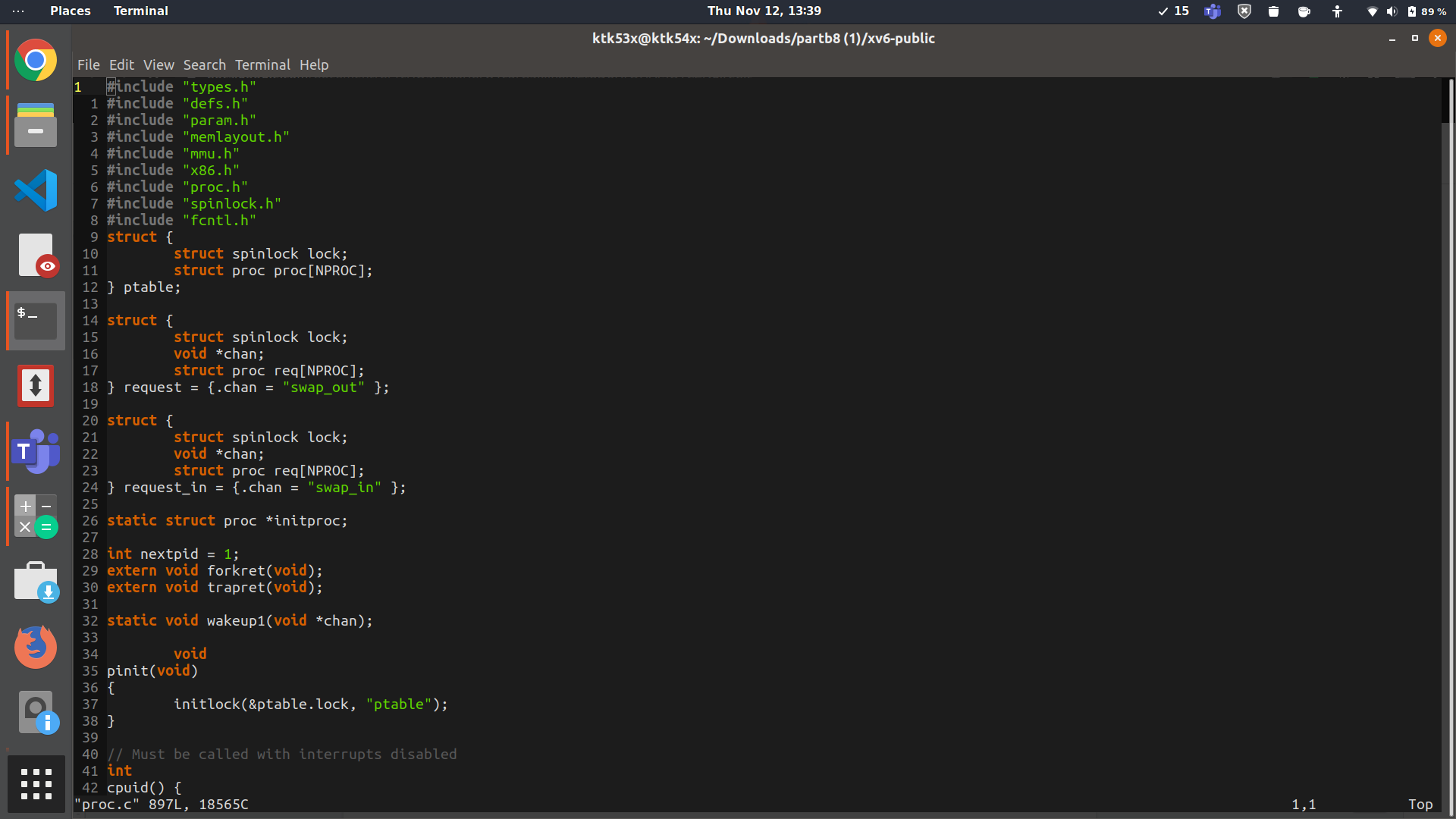
Now, to create the swap\_in and swap\_out functions and pass them as arguments to create\_kernel\_process in init.c so that these processes are there when xv6 initial process loads.



Task2-Implementing Swap Out Process

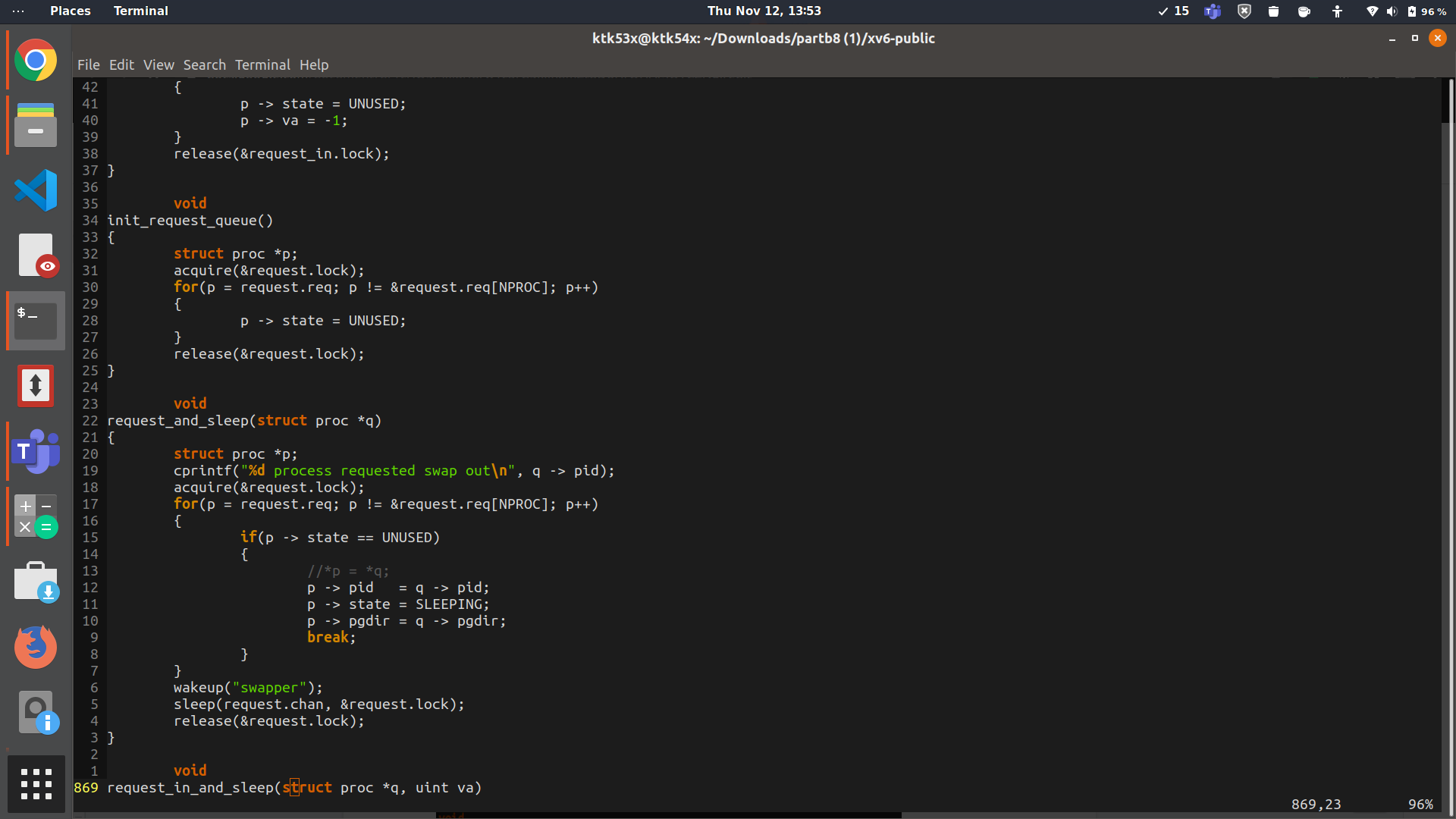


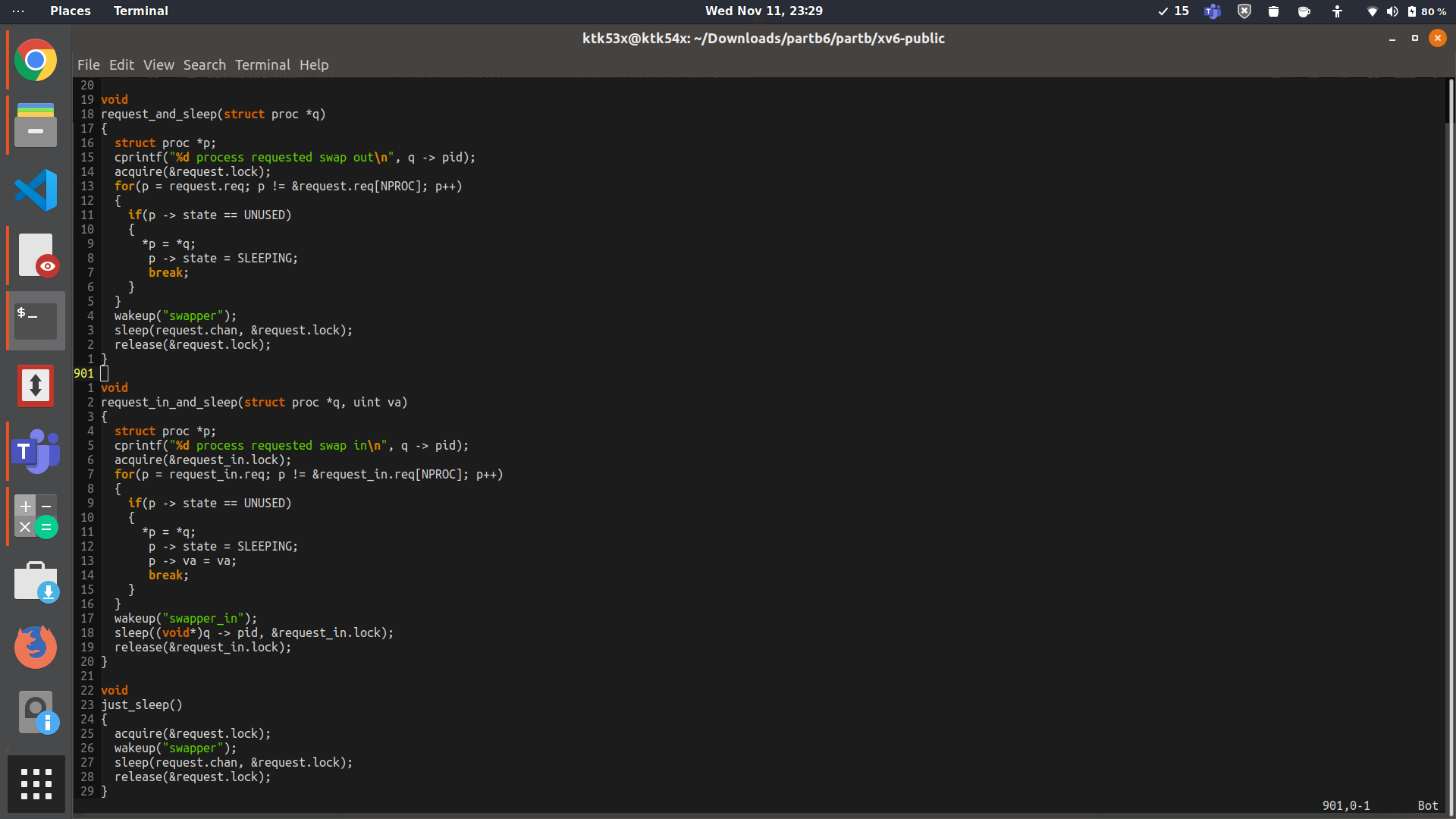
Whenever the process is needing memory and wants to increase its size from p->sz to new size , it calls allocuvm. In our test\_scheduler , this being called through malloc which in turn calls sbrk which calls allocuvm. Now in allocuvm if kalloc() returns 0, then we have no free physical frames and the process must swap out a page from physical memory. This page must belong to this process itself as we are going to implement a local page replacement policy. For this a request must be submitted to the swap\_out process and the process should go to sleep which is done by request\_and\_sleep() function. When it wakes up , if it gets the freed memory then it continues otherwise it sleeps again.



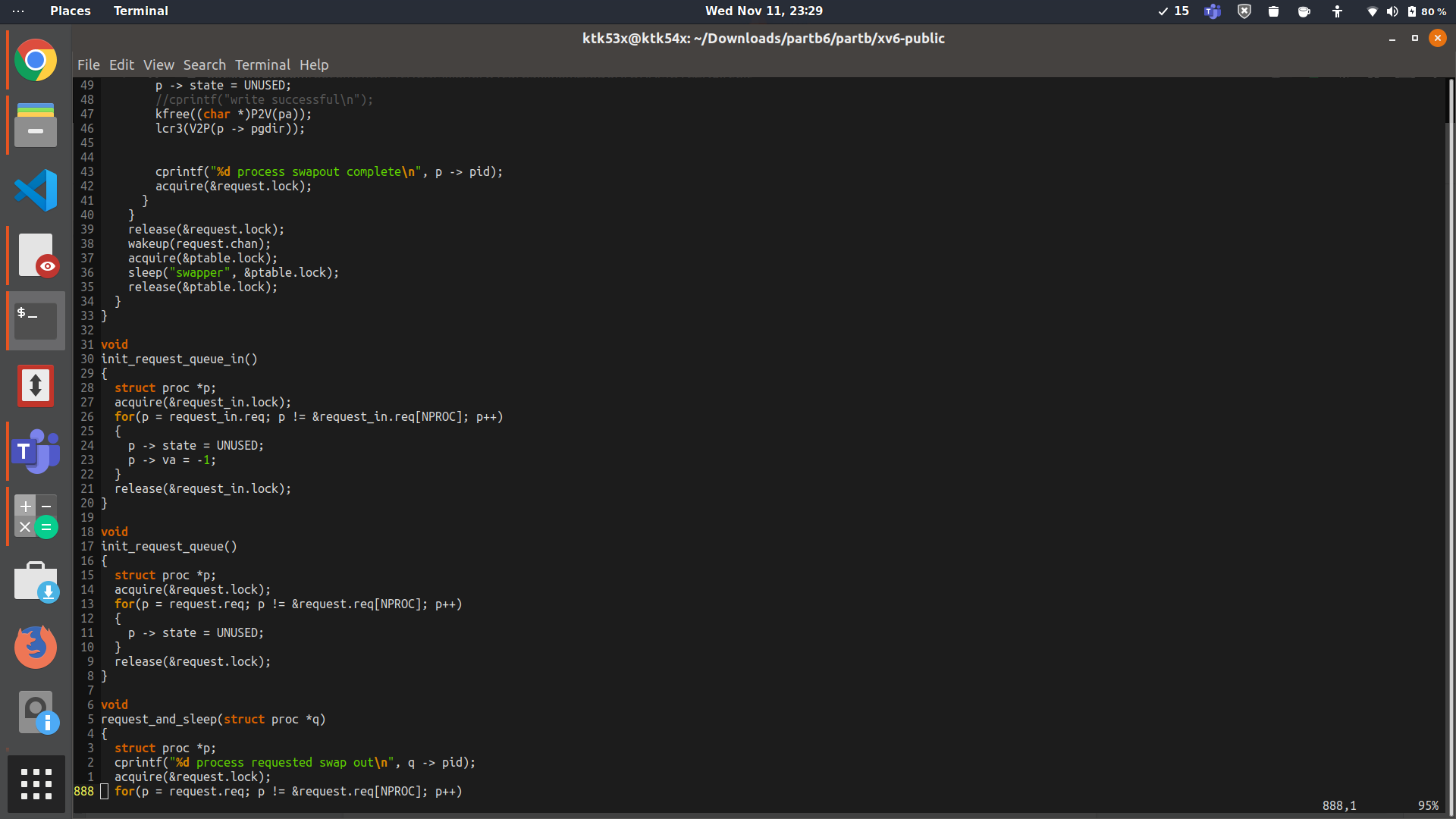
SWAP\_OUT\_QUEUE

The request structure declared above holds an array named req which has a copy of pcb’s in all those slots which are filled while the empty ones have their states as UNUSED. When an empty slot is filled , the state is change to SLEEPING , and as it is again emptied it is turned to UNUSED again

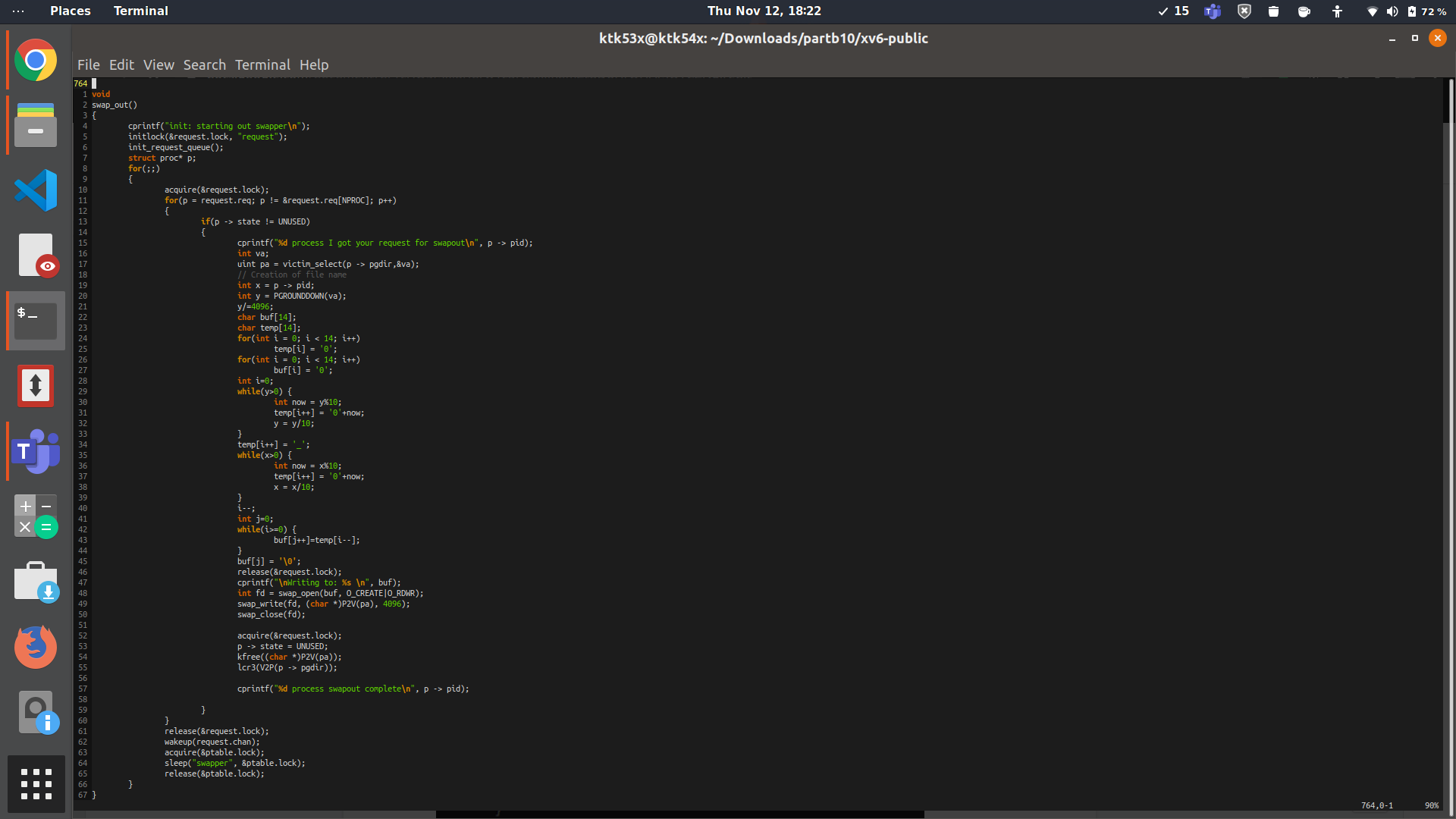




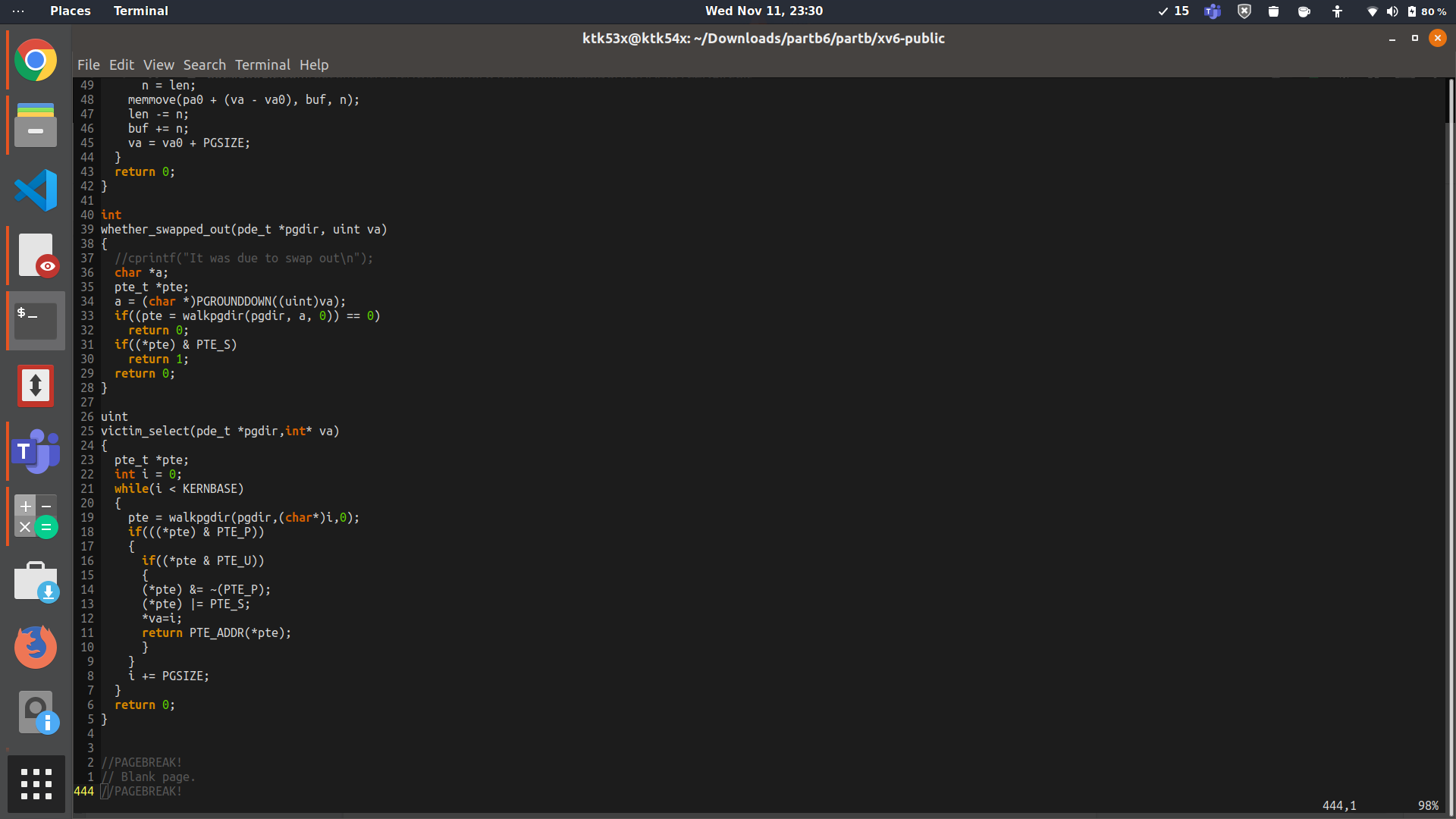
In the request\_and\_sleep function above which is called in allocuvm , we add the process to the request queue for swap out process, which is implemented just like the ptable structure. It searches through the array and in an empty slot , it copies the details of the process requesting swap out to my queue. Also now as the swap out process now has a request , it is woken up by calling wakeup.



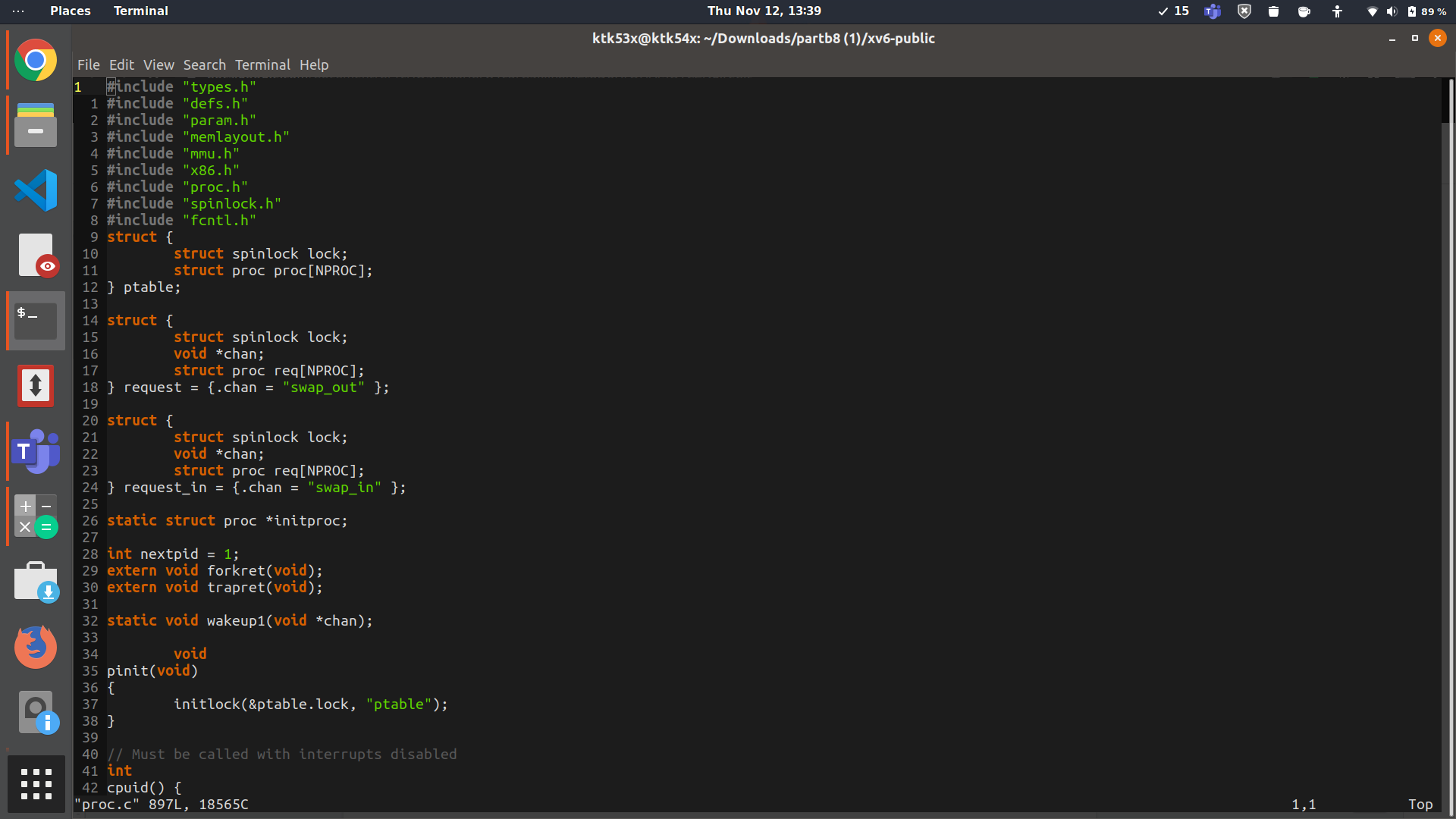
Also in above calls , the request queues for swap in and swap out are initialized to make all slots empty.



When the above swap out process wakes up, first it acquires its request queue lock as it's going to edit it by freeing the slots as and when their request is satisfied. The first task it has to do is to select a victim for swapping out which is done by victim\_select function . This function returns the physical address of the victim pa and stores the virtual address in va. Then the name of the file is stored in buf array, by converting pid and then first 20 bits section of va to characters.Then it creates a file by this name and gets its file descriptor by calling swap\_open which is just like sys\_open but modified so as to instead of taking arguments from stack by doing argfd and argint, arguments to it are passed directly. Analogously swap\_write and swap\_close write the content in pa to this file and then close the file descriptor. The memory held by this page is now freed by calling kfree and also the slot is designated as UNUSED i.e free. After going through the request table one time and servicing all their requests , the swap out process now wakes up the processes which slept on request.chan and sleeps itself.



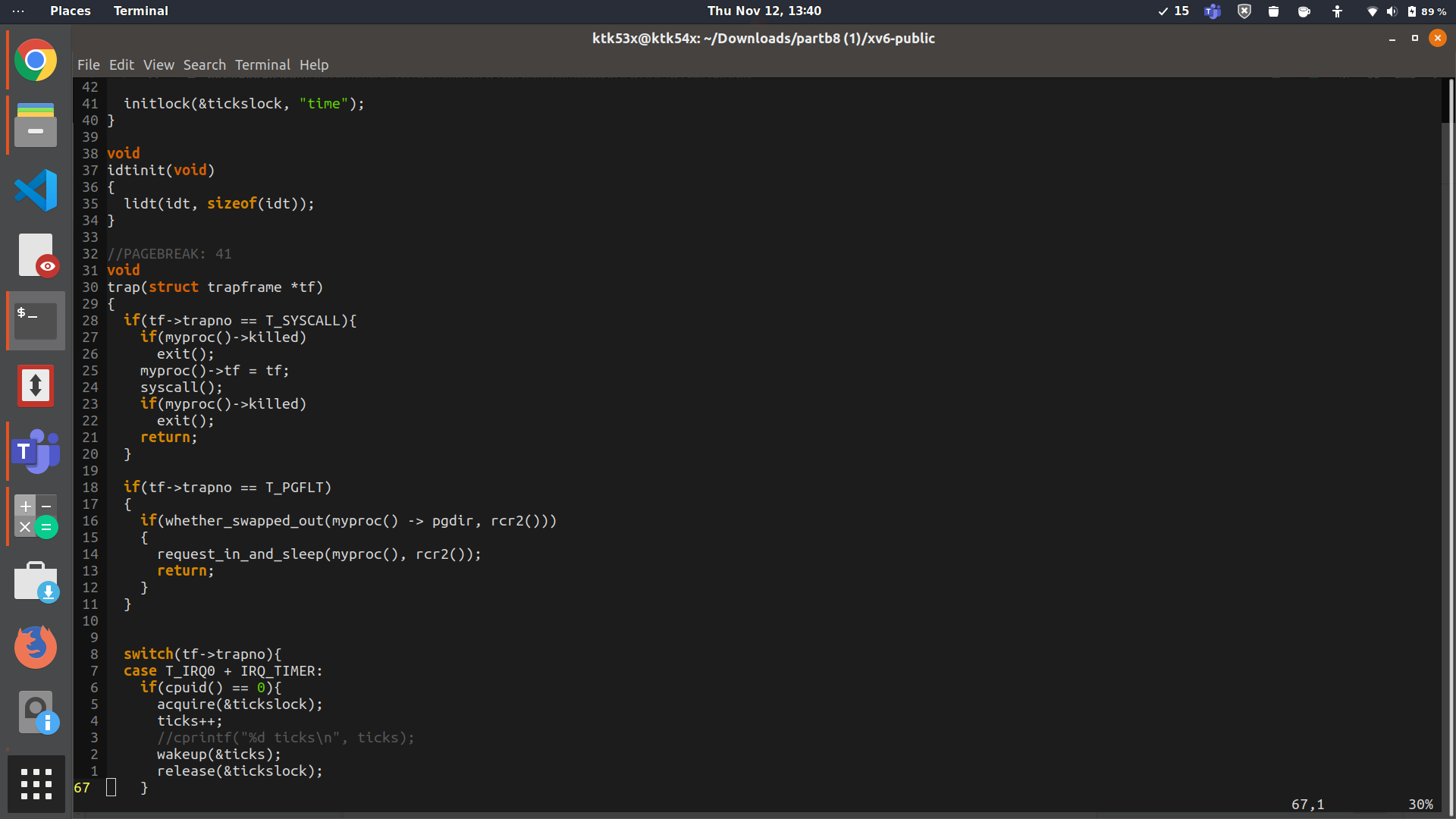
In selecting a victim function called in swap out, basically we are selecting the page to be swapped out. We are getting the page table entry corresponding to our virtual address by looking into the page table directory by using walk pgdir. We are going from virtual address 0 till KERNBASE , finding a page which has valid bit(i.e PTE\_P) set and user accessible bit (PTE\_U) set , clearing its valid bit as it's going to be swapped out , setting the PTE\_S bit to mark that this page has been swapped out, setting virtual address \*va equal to i , and returning the physical address in page table entry.



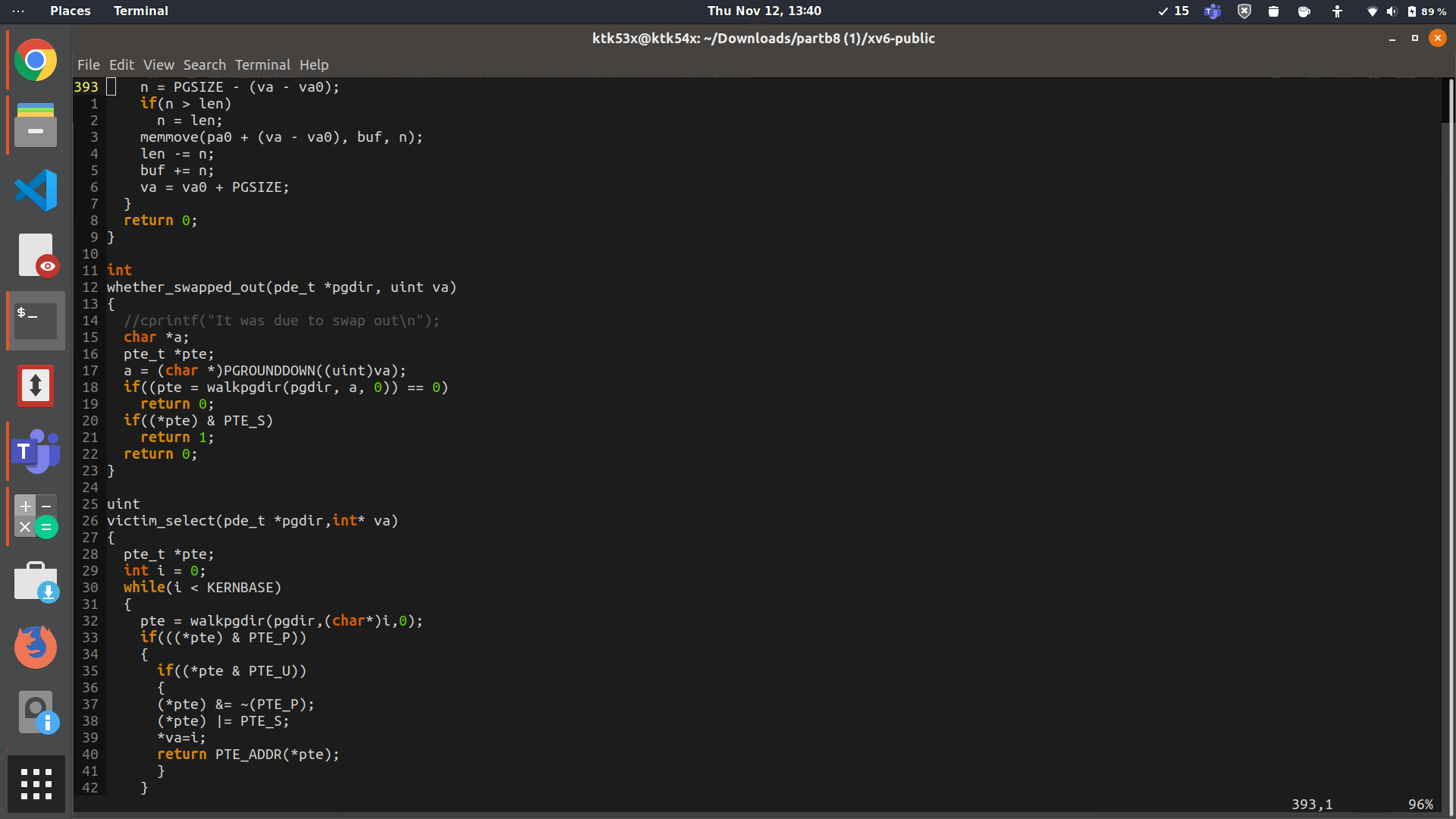
SWAP\_IN QUEUE

The request in queue has a structure just like swap out queue described above.But here we need the virtual address of the page too which the requesting process is demanding to be swapped in, hence a va field is added as metadata in proc.h structure , so that when we get a copy of pcb of the requesting process , after it we can set its va field to the virtual address argument provided by the process in its request.

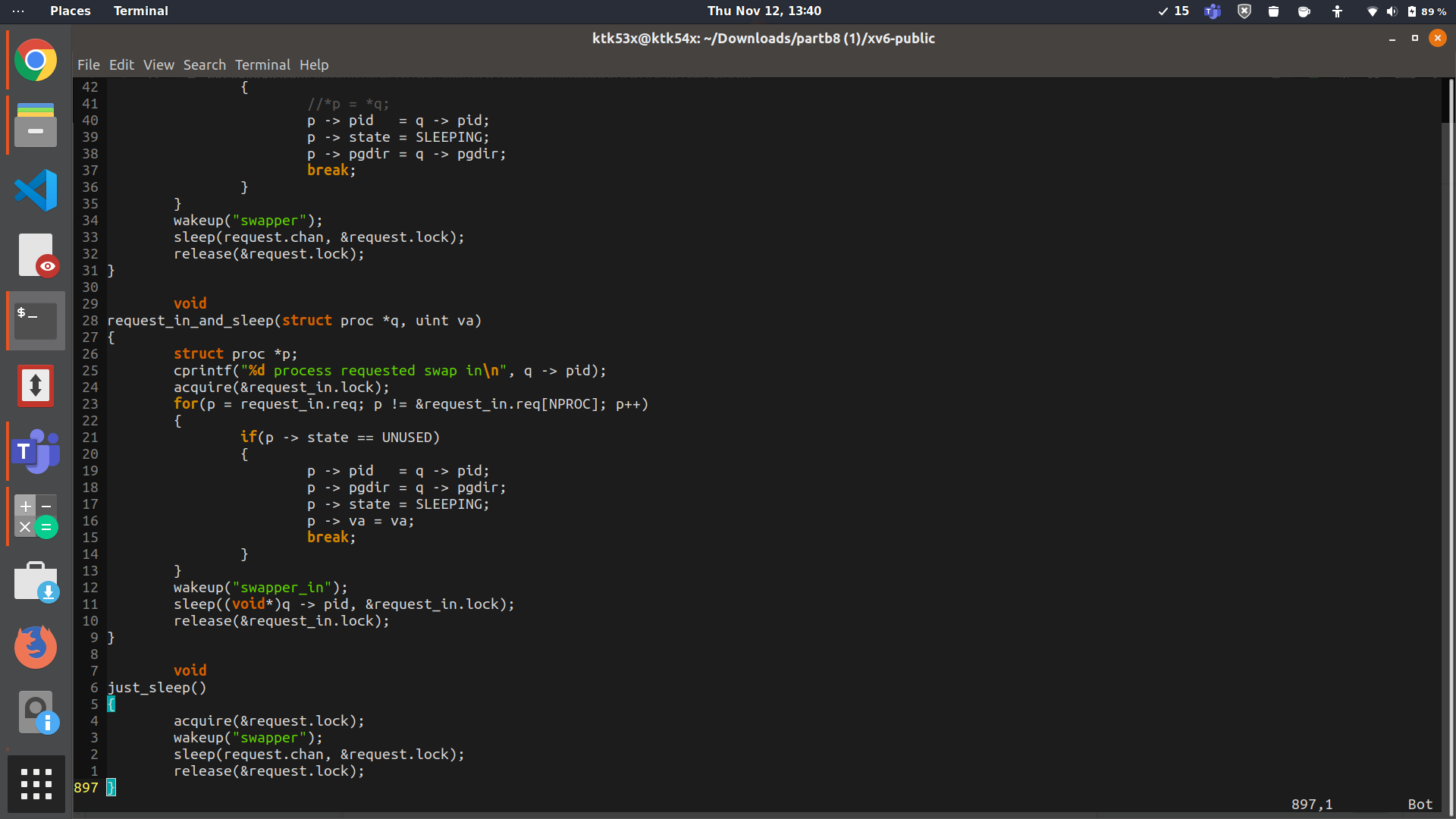
TASK3- IMPLEMENTING SWAP IN



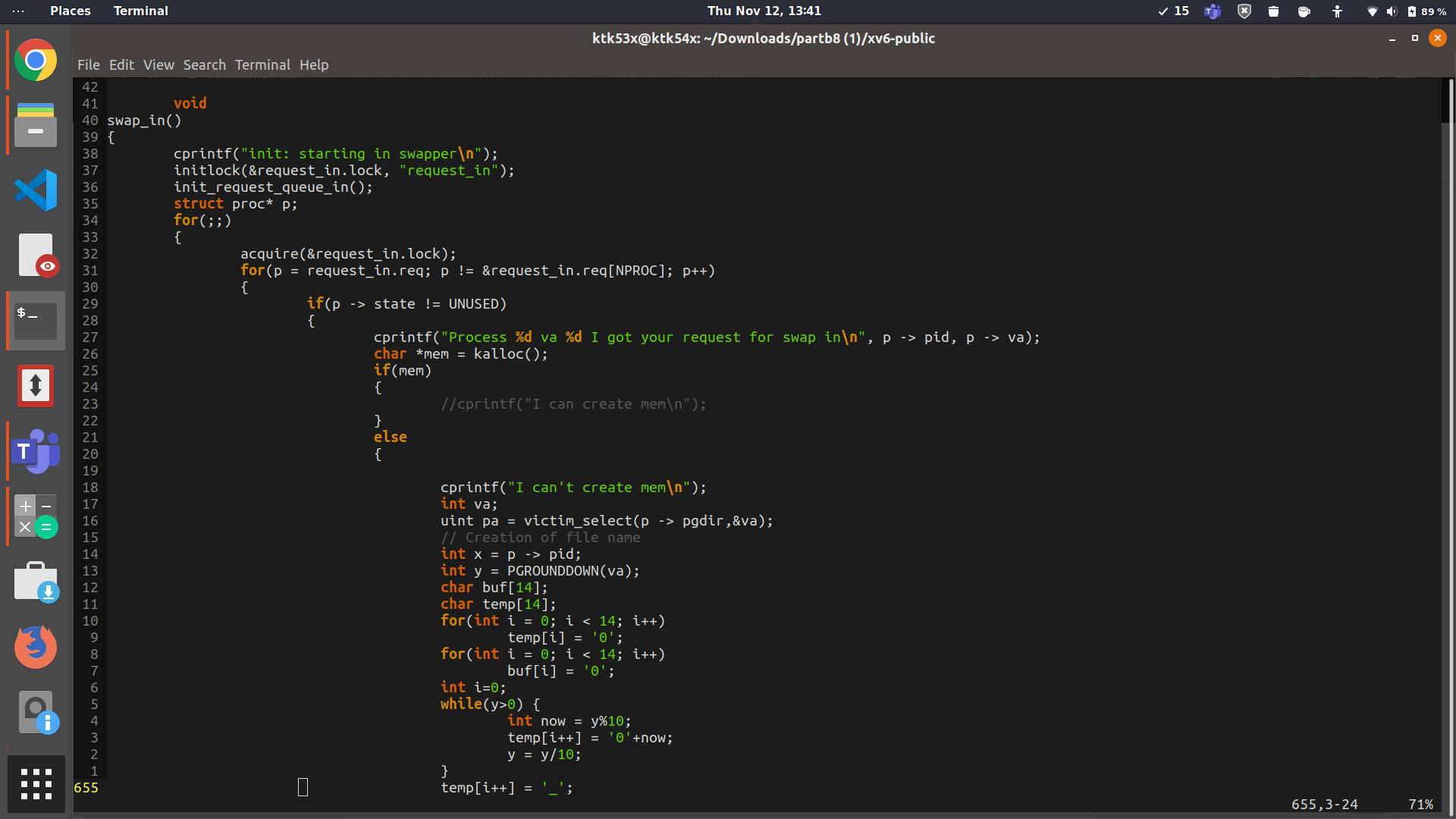
Here whenever a page fault is generated ( detected in trap.c) , we check by calling the whether\_swapped\_out whether this page fault has occurred because this particular page was swapped out when the process needed more memory . IF that is the case , then we make the process submit a request to our swapin process and go to sleep by calling request\_in\_and\_sleep.



In implementation of the function above, we are basically getting the page table entry corresponding to our virtual address , and checking whether the PTE\_S bit is set in it or not(PTE\_S is set whenever the page is swapped out to disk by the swap out process). This bit has been defined in mmu.h to be the fourth bit from last ( as the first three are reserved for PTE\_P, PTE\_U and PTE\_W) .

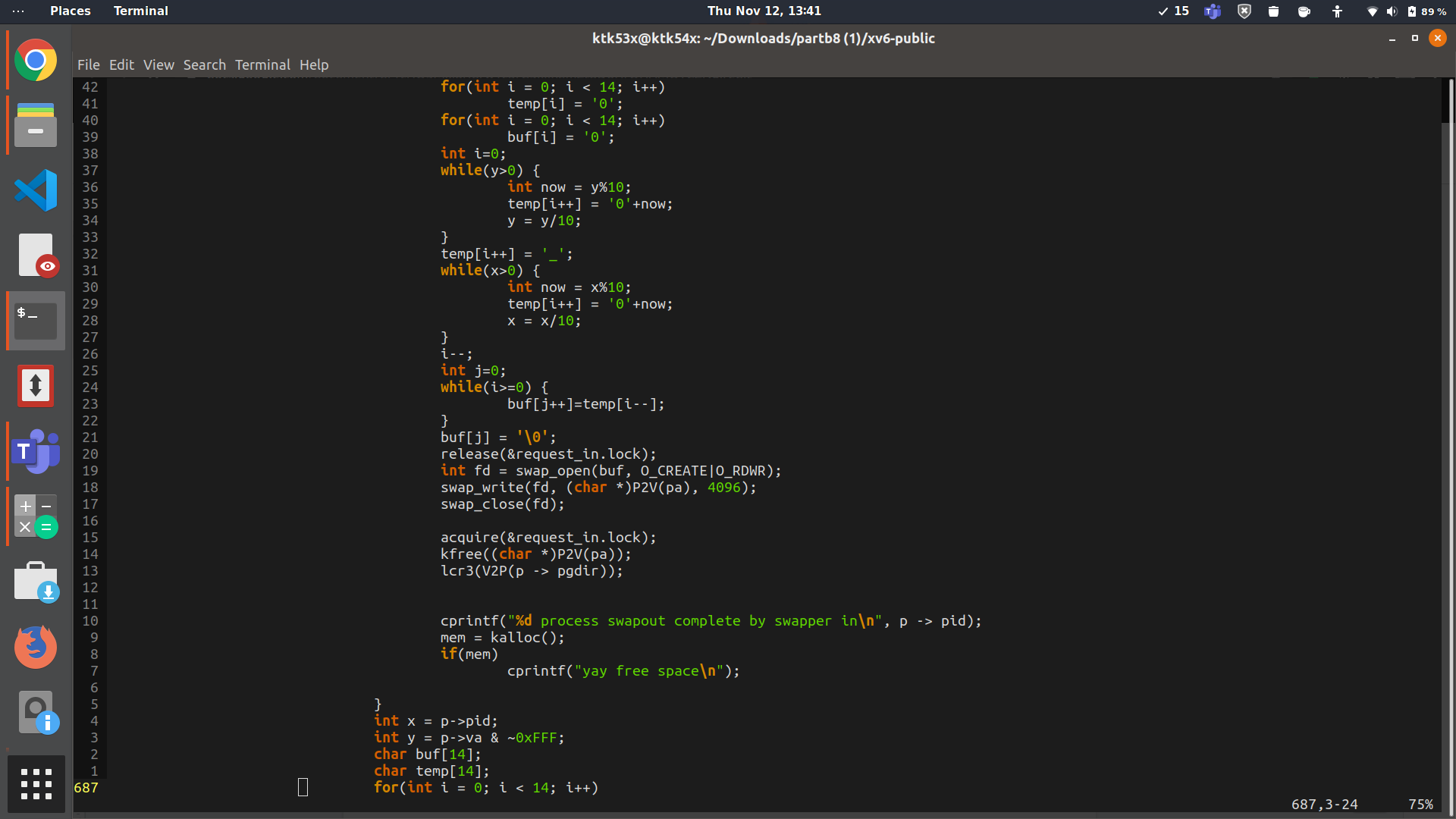


This function is called in trap.c to make the process request to swapin and then go to sleep as implemented above. Here we go through the request\_in queue to find a empty slot(i.e UNUSED) , copy the metadata of the requesting process like pid,pgdir pointer , and make the p-> va field equal to the virtual address of the page which caused the page fault. Then we wake up our swap in process which has been sleeping on the channel swapper\_in because now it has a request to serve . After that we make the requesting process to go to sleep till its request is not served.

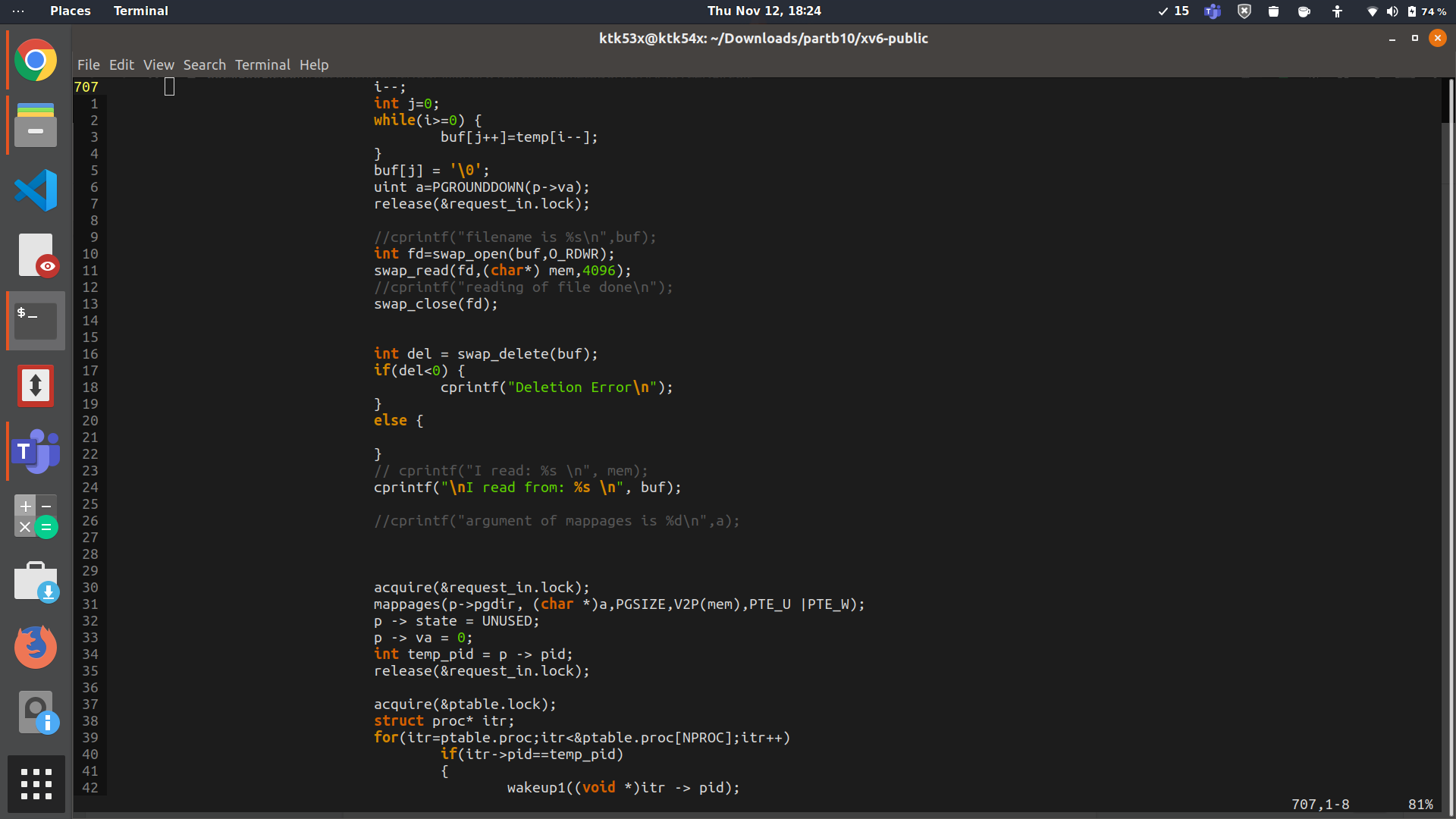


In the swap in function the init\_request\_queue\_in() is called at the very first when this process was made to clear its request queue by setting its slots to UNUSED. After that it continues to go through an infinite loop , serving the requests it has got then going to sleep at the end.

Inside the for loop , we iterate through the requests. There are two cases for each request , either we have enough physical memory left to swap in the page at some location , or the swap in process has to swap out some page from physical memory in order to swap in (latter case returns mem as zero after kalloc()). For the latter case , we swap out a page by executing a code just like our swap out process, finding a victim , writing its page to disk and then finally freeing its memory and adding it to the free frames list.



After completing the swap out segment of the code. After that, similar to swap out, we store the process pid in x and the first 20 bits of the virtual address in y. We concatenate them and store them in buf.



UPDATE ()

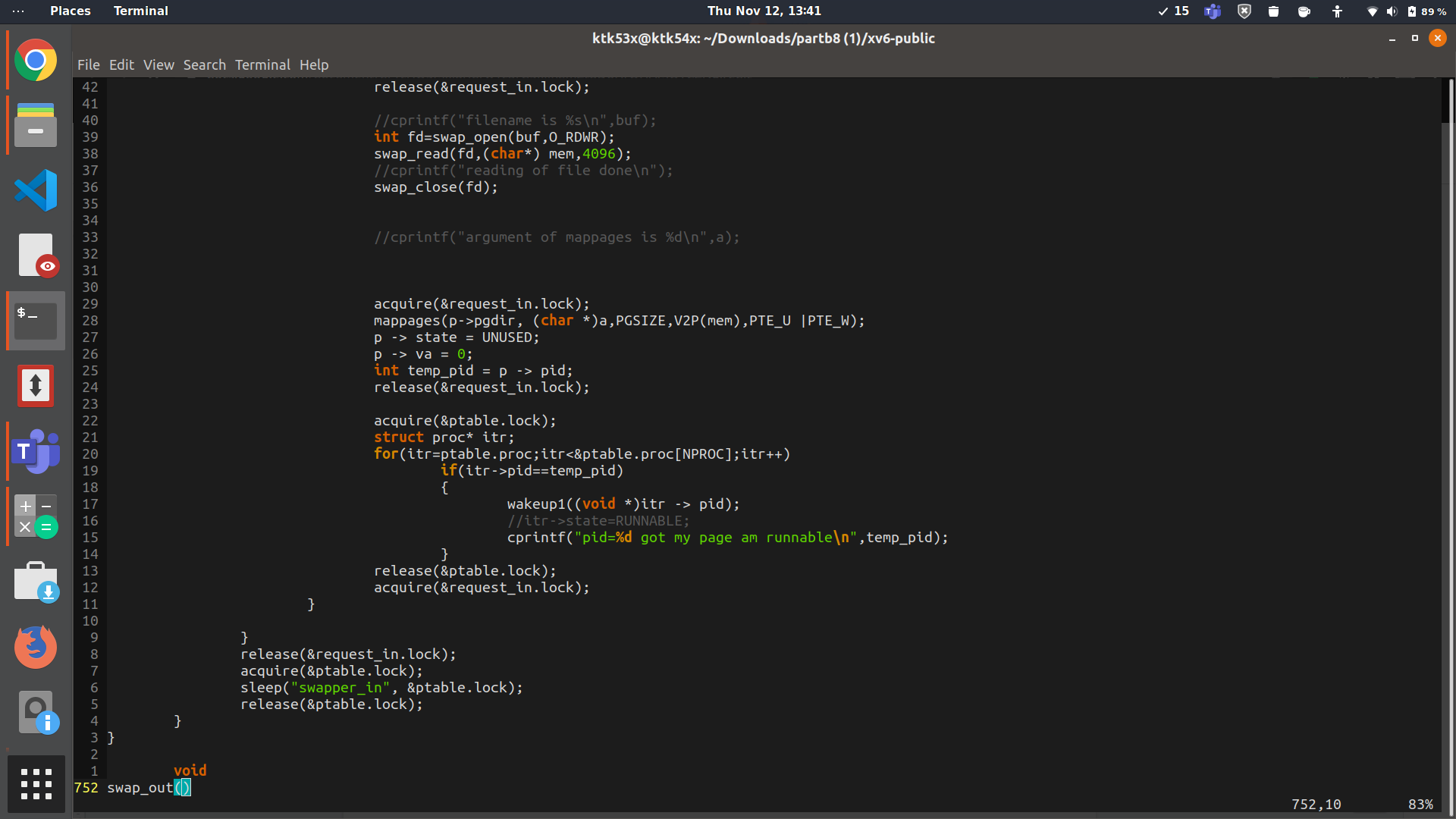
Now as we have got a free frame finally(after the printing of yay free space) , we can swap in our page.

We open buf, giving read and write access. The contents of this file (for example - 4\_3 ; where 4 is the pid and 3 is the Page Number), is read into the character array *mem*.

*a* stores the beginning of the page where this memory has to be mapped and then finally, we use mappages() to map the contents of mem to the address a and add the entry in the page table of the corresponding process.

After swapping in, the dummy process p is rendered UNUSED again.

Also, the swapped in page file is deleted from the external memory.

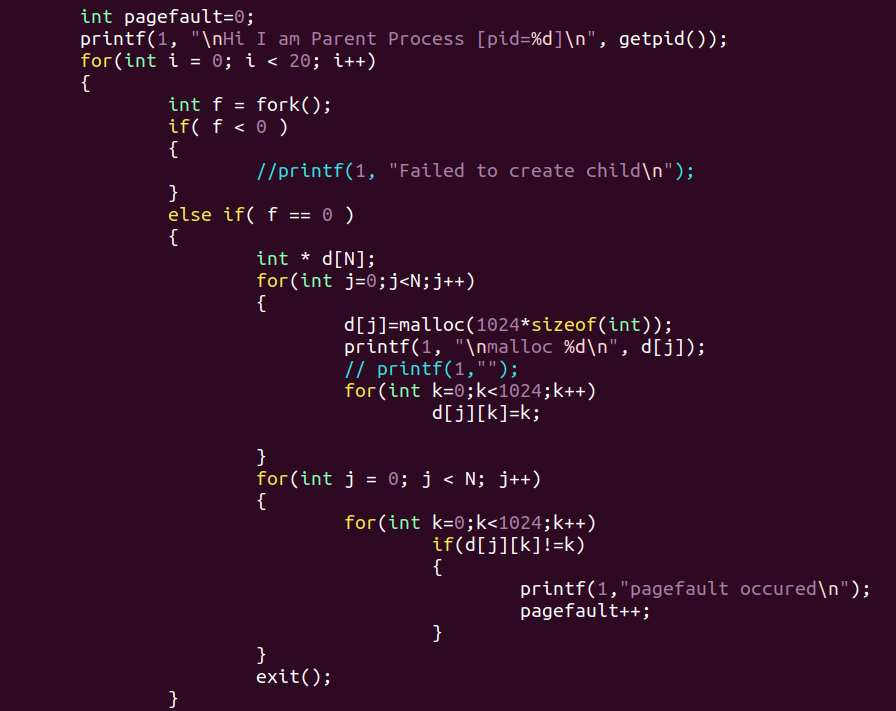


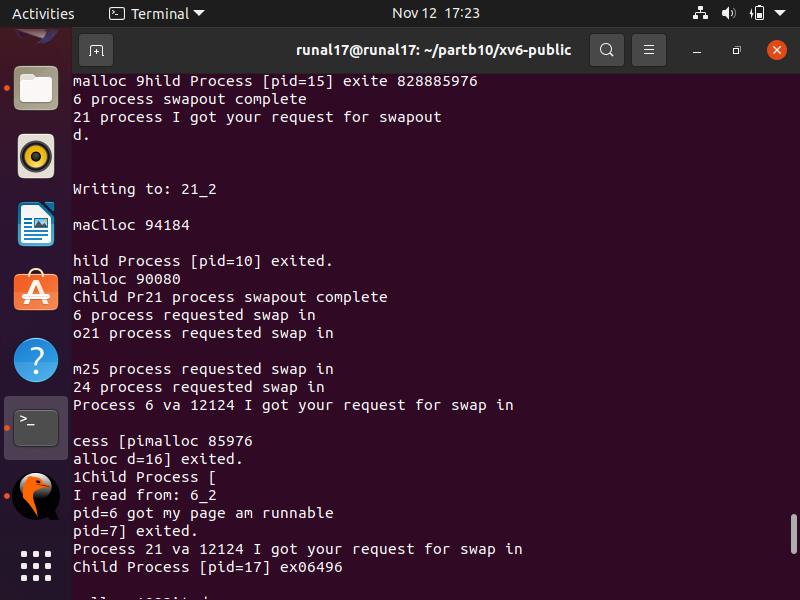
We find the process in the process table whose pid corresponds to the one that had been stored earlier in temp->pid, and then wake that process up.

Finally, request locks is released, and the *swap in* process is sent to sleep, as no process presently requires any page to be swapped in.

Whenever any process requires a page to be swapped in, swapper\_in wakes up.

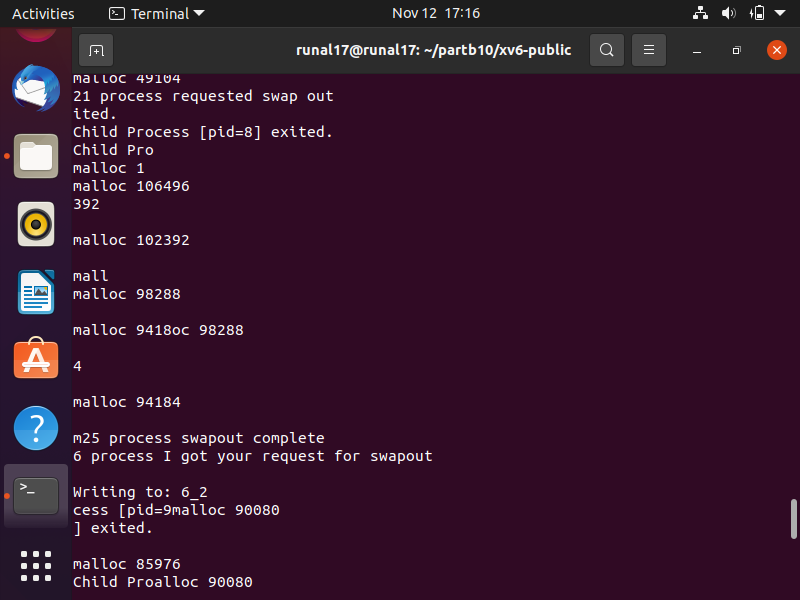
**Test Scheduler:**

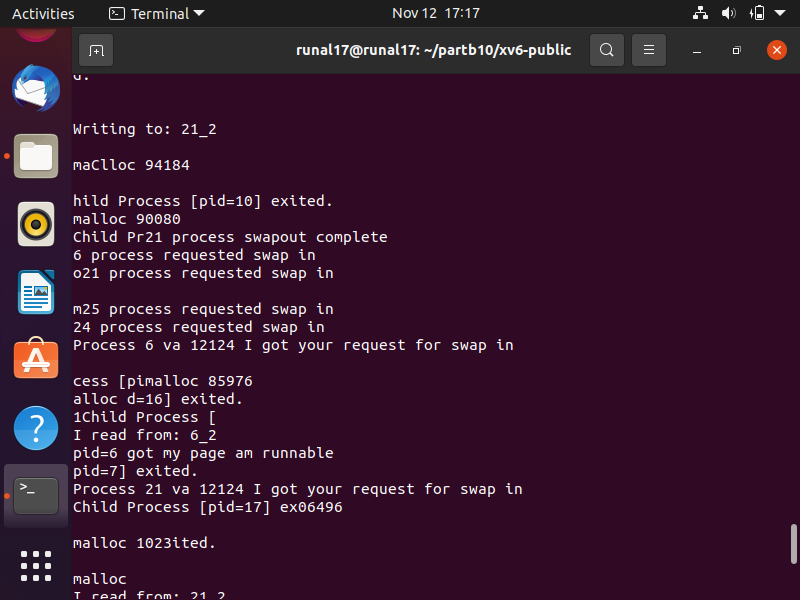
The test\_scheduler.c file contains the code of our test bench which checks if our implementation of the swap in, swap out scheme is working. The way it works is it forks 20 processes. Each one of these processes first writes data into N (a constant defined in the code) pages of memory. The process then goes on to read the data from the N pages that it had written to initially. At any point if the process cannot find a free page in the main memory, a page of its memory is swapped out and a free page is obtained. 



**Swap out Example:** Process 21 requested swap out. The swapper out kernel process then wrote the contests of the second page of process 21 to “21\_2”.

If a process requests the OS for a page that had been swapped out before, the swapper in process reads it from the disc and places it back in the main memory.





**Swap in Example:** In the picture above, the process with PID 6 requests the OS to swap in the second page from the disc into the main memory. The OS then reads this page from the file “6\_2” and brings it to the main memory. The process then declares that it got its page and that it is runnable now.

When all the processes have completed reading and writing data, they exit and the test\_scheduler terminates as well.

