Forfattere:
13.07-89 Rane Eriksen BVL193
05.01-90 Christian Nielsen BNF287
14.06-79 Sebastian O. Jensen GJX653
Hold 1

Datalogisk Institut

Københavns Universitet

Operativsystemer og Multiprogrammering

G-Assignment 4

9. marts 2015

Indhold

Rane Eriksen BVL193
Christian Nielsen BNF287
Sebastian O. Jensen GJX653

1 Task 1: TLB exception handling in Buenos

In this task we want to handle how the TLB(Translation Lookaside Buffer) is filled in Buenos.
The previous implementation was handled when the process was about to run by filling the entire TLB with the program code and program data, this is inconvenient since the TLB only have a few entries which would also limit the size of the program because the program would have to fit into the very small TLB.

Because the TLB is very small we want to handle TLB misses instead of trying to fit everything inside the TLB at once.
It may appear counter intuitive to accept that the TLB will need to handle exceptions instead of having everything available, but it makes sense in that it gives the program written to the operating system to be larger than actual available physical memory because we only load in parts of the program when it is needed.

The idea is simple and straightforward

1. Remove every piece of code in buenos that fills the TLB which will cause load or store exception.

2. Handle the exceptions cause by TLB miseses and have them fill the TLB as program code or program data is needed

3. Test that the new implementation works

The first thing in our list is easy since we only need to delete or outcomment "tlb_fill" in the current buenos, to ensure that every "tlb_fill" was outcommented we simply delete the actual function since it's no longer needed and when compiled it would result in an compilation error if buenos still referred to the function that no longer exists.
The last thing we want to do is to set the ASID in "kernel/interruption.c" in order to know what thread is causing the exception.

The second thing is a bit more complicated because we want to handle exception for USERLAND processes and KERNEL processes.
Differentiating between user mode and KERNEL mode was made easy by buenos since only userland processes have a pagetable and we can therefor

check the thread causing the exception for a pagetable, if it's NULL then we are dealing with a KERNEL thread which means we are in KERNEL mode and if the pagetable is anything else than NULL we are dealing wit ha USERLAND thread which means we are in user mode.
We want to handle exceptions caused in KERNEL mode by panicing which results in the entire system stopping at that point.
We do it this way because the KERNEL should not cause exceptions.

We to handle exceptions caused in user mode differently.
When an exception is caused in user mode we want to check that the "local" TLB contains the virtual address provided by the exception state, we do this by iterating through every entry in the 'local" TLB, if a match is found we want to check that the thread causing the exception is in fact entitled to use the entry in the 'local" TLB, we do this by checking that the ASID obtained by the exception state is the same as the "local" TLB entry's ASID.
If we fail to find any entry in the 'local" TLB we are left with an access violation because the thread that caused the exception did not have permission to the data it had access to.
If however we find an entry we then want to check if the address is an odd or even page since buenos stores two pages per entry.
We then check if the page is valid, if it is not it is treated as an access violation otherwise we fill the "global" TLB with the "local" TLB entry by using the _tlb_write_random function which fills the "global" TLB or replaces an entry in the "global" TLB with the found "local" TLB.

We handle access violation by terminating the thread, setting the corresponding process state to ZOMBIE and the return value of the process to -1 to indicate an error.

The above method is handling EXCEPTION_TLBL and EXCEPTION_TLBS

We want to handle EXCEPTION_TLBM differently in that it is always considered an access violation which means we may skip the search and replacement of TLB entries and jump to the access violation part we explained above.

The third thing in our list requires our test to access some memory to test that the new TLB exceptions are implemented properly, we also want to test that accessing memory that is not in the "local" TLB is treated as an access violation.

To do this we have created two tests and a main program to run these tests.
The first program is called faul_addr_test.c - This tests for accessing memory that it does not have access to thus creating an access violation.
The second program is called no_faul_addr.c which tests for memory that it has access to.

We primarily test the load exception since the store exception must do the exact same thing, also we do not test the modified exception since it only handles access violations which we already test when testing the load exception.

in faul_addr_test.c we create two variables A and B with type pointer to Int and int respectively.
We then assign A to an address to an Int which the program does not have access to, however this is no problem because this is simply an address, but when we try to convert the address to and Int and we try to give the value to B we then create an access violation.
The process should then be terminated and the main program should get a return value corresponding to -1

The test of this program is to test if our exception handler terminates the program correctly by checking that process_join corresponds to -1 - If it does we have succeeded otherwise we have failed.

We repeat this process 4 times just to be safe.

In no_faul_addr.c we do not want to have the thread terminate but we want to test that runs through the entire program with no errors at all, this way we know that the "global" TLB is filled correctly.

The program is again very simple.
Instead of using pointers to Int for A we use real values which we then assign to B which results in store and load exceptions, if the program terminates with 0 everything went as expected and when the main program does a process_join on the spawned process the return value should be 0.
If that is the case we have succeeded otherwise failed.

Which files we changed

In order to complete the assignment we have modified the following files:

- kernel/interrupt.c
  We outcommented tlb_fill and included _tlb_set_asid(thread_get_current_thread()); to set the ASID as mentioned earlier

- kernel/exception.c
  We replaced KERNEL panics with function calls to their respective handler in tlb.c

- proc/exception.c
  We replaced KERNEL panics with function calls to their respective handler in tlb.c

- proc/process.c
  We outcommented tlb_fill

- tlb.c
  We implemented the handlers as described above.

2 Task 2: Dynamic memory allocation for user processes

Unfortunately we did not manage to make sense of this task which results in an unimplemented version of Dynamic memory allocation for user processes.