Assignment-2: System calls and exception handlers

Description

Recap: gemOS is in 64-bit mode executing itself as the first context (say the boot context). The boot context sets up the page table, stack, segment registers for itself. Further, it implements basic input output to a serial console, and puts itself into a basic shell.

At this stage, GemOS implements a command called `init` which creates the first user process (named as the `init` process with PID = 1). Source code for `init` process can be found in `user/init.c` file. The current init process invokes two system calls—`getpid()` and `exit()`. The objective of the assignment is to implement new system calls and exception handlers and enable support for lazy memory allocation. To handle the system calls, GemOS installs an IDT entry at offset 0x80. The system call handlers in in GemOS are defined in `entry.c`.

Part A

In this part of the assignment, you are required to implement three system calls as described below,

`int write(char *buf, int length)`
Outputs the characters in buffer into the console and returns the number of bytes written. The length parameter is limited to 1024 bytes. You must check the validity of `buf` address by accessing the `init` process page table. Note that, for convenience, CR3 is not switched on syscall or exception entry points. `write` system call should return -1 for invalid memory addresses and incorrect values of length.

`void *expand(u32 size, int flags)`
Expands the memory area by number of pages specified in the `size` parameter where maximum value of `size` can be 512. The memory segment is selected based on values of flags—`MAP_RD` expands the `MM_SEG_RODATA` segment while `MAP_WR` expands the `MM_SEG_DATA` segment. These segments are part of `struct exec_ctx` and can be accessed by accessing the member named `mms`, which is an array of `struct mm_segment`.

```
struct mm_segment{
                unsigned long start;
                unsigned long end;
                unsigned long next_free;
                u32 access_flags;   /*R=1, W=2, X=4, S=8*/
};
```

An element `next_free` is added in this assignment to denote the next unused virtual address (at 4K boundary). So at any point of time, for `MM_SEG_RODATA` and `MM_SEG_DATA`, the allowed virtual address range is `start` to `next_free - 1`. `expand` system call should expand the allowed virtual address range by 'size' number of pages and return the start of newly added virtual address region (in other words, return value of `next_free` before expansion). *Note that, expand does not allocate any PFN to map the expanded virtual address region.*

`void *shrink (u32 size, flags)`
Shrinks the memory area by number of pages specified in the `size` parameter. The `shrink` system call should bring down the `next_free` marker of the memory segment by 'size' number of pages and return the current end address of the virtual address segment (updated `next_free`). The memory segment is selected depending on the `flags` argument, similar to the `expand` system call. Further, the `shrink` system call must *release all the PFNs mapped to by virtual pages being released and update the page table entries appropriately.* Both `expand` and `shrink` system calls should return `NULL` (0) if the `size` argument grows the segment beyond `mm_segment.end` or shrinks the segment below `mm_segment.start`, respectively.

Part B

In this part of the assignment, you are required to implement two exception handlers in gemOS as described below.

Divide-by-zero

If the user process (in this case `init`) performs a division by zero, this exception handler is invoked. In the IDT, the handler (`handle_div_by_zero` defined in `entry.c`) is already registered. You are required to implement the handler in which you should call `do_exit()` after printing an error message as, "Div-by-zero detected at [RIP]", where RIP is the instruction pointer address of the faulting instruction. Note that, the hardware does not push any error code into the OS stack before invoking the handler. The OS stack points to user RIP when the exception occurs (see slide 20 of Userland.pdf).

Page fault

If the user process (in this case `init`) accesses any virtual address for which valid mapping is not present or permissions (R/W or U/S) are not sufficient in the page tables, the page fault exception handler will be invoked. In the IDT, the handler (`handle_page_fault`) defined in `entry.c` is already registered. When the exception occurs, the hardware will push an error code along with user RIP and other information into the OS stack as shown in slide 20 of Userland.pdf. The `CR2` register contains the faulting virtual address. Relevant fields of the hardware generated error code are described below.

bit-0: value 0 implies present bit in page table is not set in one of the paging levels, value 1 indicates protection violation.

bit-1: value 0 implies the access causing the fault is a read, value 1 indicates the access is a write.

bit-2: value 0 implies the access causing the fault is from CPL=0, value 1 indicates the access is from CPL=3

You can ignore all other bits.

Your page fault handler should handle the fault for different virtual memory segments differently. If the virtual address does not belong to any of the following memory segments, you should print an error message with RIP, accessed virtual address and error code before terminating the process by invoking `do_exit`.

`MM_SEGMENT_DATA`**:** A page fault in this region can occur only if the page table mapping for a virtual address is not present. Before creating a new mapping in the page tables, you must check the `next_free` value of the memory segment. If the address is not legitimate (not between `start` and `next_free`) you should terminate the process by printing an error message with accessed virtual address, RIP and error code.

`MM_SEGMENT_RODATA`**:** A page fault to this region can occur if the page table mapping for a virtual address is not present or the access is a WRITE access. Before creating a new mapping in the page tables, you must check the `next_free` value of the memory segment. If the address is not legitimate (not between `start` and `next_free`) or the access is a WRITE access, you should terminate the process by printing an error message with accessed virtual address, RIP and error code.

`MM_SEGMENT_STACK:` A page fault to this region can occur if the page table mapping for a virtual address is not present. Before creating a new mapping in the page tables, you must check the `start` and `end` value of this memory segment. If the address is not legitimate (not between `start` and `end`), you should terminate the process by printing an error message with accessed virtual address, RIP and error code.
For all of the above cases, if the virtual address is legitimate, *you must install the page table for the accessed virtual address* (like assignment-1) before returning to the user process. Note that, after fixing up the page fault, you must execute `iretq` instruction by placing the stack pointer register (RSP) to the user process RIP as shown in slide 20 of Userland.pdf. Note that, the page table structure is same as assignment-1.

Submission guidelines

- You are required to submit `entry.c` file with implementations for system call and page fault handlers.
- Remove/comment all print statements used for debugging.
- Your implementation will be tested with several test cases. So, your implementation should be generic.
- Maximum size of allocated physical memory will not exceed 16MB at any point of time. This will be ensured in all the test cases.