Assignment-3: Signal, Timers and Multi-tasking

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 supports five system calls—`getpid()`, `exit()`, `write()`, `expand()` and `shrink()`. gemOS also implements lazy memory allocation by handling page fault exception. Further, divide-by-zero exception is also handled by the gemOS. Now we are ready to take the next step.

Objectives of the assignment are to implement new system calls (`signal(), alarm(), sleep() and clone()`) along with basic signal handling and multitasking using a round robin (RR) scheduler. To enable multi-tasking design of gemOS, a periodic timer interrupt is initialized with a handler function `handle_timer_tick` defined in `schedule.c`. Every invocation of the periodic timer interrupt handler is counted as one *tick*. Note that, the timer interrupt handler has the same semantic of a div-by-zero fault handler, albeit with a separate interrupt stack.

For this assignment, a list of contexts is maintained in gemOS—accessed using `get_ctx_list()`, which returns the pointer to the first process (PID = 0). You can iterate the list as an array of pointers using PID as an index. Currently, the maximum number of contexts is defined by a macro `MAX_PROCESSES` which is 16 (PID=0,1...15). For more details, please refer to the definitions of `struct exec_context` and process states in `include/context.h` and `include/schedule.h`, respectively. A template of the required implementation is provided in `schedule.c` file.

Part A - Signals

In this part of the assignment, you are required to implement the signal handling functionality for three signals (enumerated in `include/schedule.h` file) and described as follows,

- *SIGSEGV:* This signal corresponds to an invalid access of a memory location by the program.
- *SIGFPE:* This signal corresponds to a divide-by-zero operation by the user program.
- *SIGALRM:* This is an alarm signal generated after every *numticks* number of timer interrupts where *numticks* is specified using `alarm()` system call. For example, `alarm(5)` will set the *numticks* to 5. See the man page—`man alarm` for more details.

For all of the above signals, signal handlers are registered using `signal(signo, handler)` system call that is required to be implemented as part of the assignment. In the extended definition of `struct exec_context`, a bit vector and a signal handler array is provided to maintain the pending signals and the handlers, respectively. In the design of the signal handling mechanisms, assume that there will be no nested signal handler invocations. For SIGSEGV and SIGFPE, the function `insert_sync_signal` is invoked from the exception handlers which has the following semantics, `long invoke_sync_signal(int signo, u64 *ustackp, u64 *urip)`

signo is the signal number.
ustackp is the pointer to the stack pointer location in the exception entry stack.
urip is the pointer to the instruction pointer location in the exception entry stack.

The `alarm(ticks)` system call should initiate counting of the ticks using `ticks_to_alarm` member of `struct exec_context`. When the ticks expire, a signal must be sent to the user space if the signal handler for SIGALRM is registered, ignored otherwise. Note that, you are required to invoke the `invoke_sync_signal` function with appropriate interrupt stack pointers to deliver this signal.

Note that, for this part of the assignment, uni-process test cases (only with `init` process) will be used and not be tested with features mentioned below.

Part B - System calls (`sleep()`) and Swapper process

As part of this assignment, you are required to implement sleep functionality for the `init` process.

`int sleep(u32 ticks)`
Suspends the execution of the calling context for *ticks* number of timer ticks. During this time, the context is moved to WAITING state (see `struct exec_context` in `include/context.h`) and the swapper process (with pid=0, already created in the system which is in ready state) is loaded. If timer interrupts arrive while the swapper process is RUNNING, the ticks should be accounted for (using `ticks_to_sleep` member of `struct exec_context`) and depending on the remaining ticks either the swapper process is rescheduled or the sleeping context is scheduled.

Initially, the swapper process context along with the `regs` member which is of type `struct user_regs` is initialized in a manner such that if the currently used OS stack (the current context OS stack or the interrupt stack) is loaded with the last five elements before executing `iretq`, the swapper process is scheduled. In general, this strategy may be employed to switch between any two processes.

Please note that, before performing the actual switch to the new process (see `schedule_context()` in the template code), your code must invoke `set_tss_stack_ptr(next)` and `set_current_ctx(next)`, where `next` is the incoming context.

Part C - `clone()` and scheduling

In this part of the assignment, you are required to implement context creation functionality using `clone()` system call. Further, you are required to implement a round robin scheduler to schedule the contexts (in READY state) in the system.

`int clone(void *th_func, void *user_stack)`
th_func is the pointer to the function that will be executed by the newly created context.
user_stack is the pointer to the stack which will be used by the newly created context. This has to be a memory location in the MM_SEG_DATA region after expansion (using expand() system call followed by initialization) .

Assume that the above two virtual memory parameters are always correct and not required to be validated by walking the page tables of the calling process. The system call handler for `clone()` must create a new context which is a copy of the parent process with the below mentioned exceptions. You should use `get_new_ctx()` declared in `include/context.h` to allocate a new context. The pid of the returned context will be already initialized and the status will be set to NEW. All other fields of the context should either be initialized or copied from the parent context. The values which are not copied are,

- `os_stack_pfn` must be allocated for the new context using `os_pfn_alloc(OS_PT_REG)`
- `name` must be the name of the parent appended with the pid value of the context. You may use `memcpy()` call declared in `include/lib.h`.
- `regs` must be appropriately initialized so that when it is scheduled, the new context executes the `th_func` using `user_stack`. You may set the values of SS and CS to 0x2b and 0x23, respectively (see slide-20 of Userland.pdf). The value of RFLAGS must be set to the RFLAGS value of the parent.

Please note that, this `clone()` implementation is neither a thread or a process. This is because, even though the CR3 is the same, the `mm_segments` field is copied and separate for the parent and the newly created context. Therefore, to avoid cases with memory issues, `clone()` will be invoked from the main process (i.e., `init`) only. After the creation of the new context, the parent returns from system call and the child state is set to be READY to be scheduled. Note that, the behavior of `exit()` system call should be modified (please see the template for `do_exit()` in `schedule.c`). The exit() system call should free the `os_stack_pfn` and change the process state to UNUSED. Further, if there are no other contexts in the system except the swapper process, it must invoke `cleanup()`. Otherwise, the scheduler should be invoked to schedule the next READY process or the swapper process.

The scheduler can be invoked in three different ways—when a process exits, when a process blocks using sleep() system call or on a timer interrupt. In the RR scheduling scheme, the next READY process in the list after the currently running process (in a circular order including the current process as the end) is scheduled. If there are no ready contexts in the system, swapper process must be loaded. Note that, whenever a context switch happens, the PID of old and new must be printed.

Submission guidelines

- You are required to submit `schedule.c` file with implementations for system call, signal and scheduling.
- Remove/comment all print statements used for debugging.
- Your implementation will be tested with several test cases and hence should be generic.
- Please refer to the code comments before asking doubts regarding assignment.