PROCESS MANAGEMENT IN OPERATING SYSTEM

Metin Balaban
S131105
Danmarks Tekniske Universitet
+4550299770
s131105@dtu.dk

ABSTRACT

In this report, I describe my design details of *createprocess* and *terminate system calls in an operating system*. Beside these system calls, I describe details of *kfree* and *kmalloc* system calls, which are other two system calls which I have implemented and reported before.

The implementation that I describe may not be too much efficient. However, it is easy to implement and enough for a small embedded system. I check my implementation in terms of consistency and give the details of test cases.

1. INTRODUCTION

Before giving the details of my work in this assignment, I want to give a brief description of the expectations in *createprocess* and *terminate* system calls. *createprocess* routine is called from a user program in order to create new process and start executing it. *terminate* system call is carried out automatically when main function of a computer program terminates. According to assignment description, the output of execution of sample program is :

```
Process 0: Trying to start process 1.

Process 1: Trying to start process 2.

Ta da!

Process 1: Process 2 terminated.

Process 0: Process 1 terminated.
```

After I have investigated the source files of sample programs and expected output, I have realized that execution of processes show stack properties. More clearly, processes should be stored in stack and each recently created process should be put on top processes stack. For a single process operating system, this property is supposed to be the solution. As we will see in further sections, queues are more appropriate for multiple processes operating systems to keep running and ready processes. Accordingly, I set an infrastructure describing a process stack.

In section 2, I describe the details how I have transferred my previous memory allocation system to the operating system being implemented. In section 3, I define the infrastructure of process management. In section 4, I describe the algorithm of *createprocess* system call. In section 5, I give the algorithm of *terminate* system call. In section 6, testing these two system calls are discussed. Section 7 and 8 are dedicated to scheduling, multi-processes, multi-threads, and multi-processors.

2. IMPLEMENTING MEMORY ALLOCATOR SYSTEM CALLS

As a part of assignment 1, I had already implemented a memory allocation system for an embedded system. I have ported my implementation on the operating system that I work on. However, in sake of compatibility, several changes have been necessary. In the forthcoming subsections, code transfer details are explained.

2.1 Updating globals.h

I have integrated the data structure needed by kalloc and kfree to globals.h file. I have updated type information of the fields of data structure according to a 64-bit system. I have transferred following definitions.

```
#define BLOCKSIZE 32

#define ALLOCATE_SIZE
(((amd64_top_of_available_physical_memory-
amd64_lowest_available_physical_memory))<<6)
```

As a result of pointer arithmetic, result of the subtraction above has to be multiplied by 64, which is equivalent to 6 left shifts.

2.2 Updating system_initialization.c

I define global pointers used by *kfree* and *kmalloc* in this file. These global files are

```
blockPtr base=0;

blockPtr last;
```

I also define helper functions of these two methods in this source file. I make definitions of *kmalloc* and *kfree* functions, regarding to the description. Functions now return ERROR in erroneous cases. In all the functions I have added to, I replace memory referencing pointers with *amd64_lowest_available_physical_memory* variable.

3. BUILDING PROCESS INFRASTRUCTURE

For an operating system executing a single process until that process terminates, I assert stack is the best data structure to keep processes and the processes created by them. The idea is that each valid *createprocess* system call pushes a new process to stack and each valid *terminate* system call pops the top process (current process) from stack.

3.1 Stack Interface

```
extern struct process_stack_element *
top_process;


extern uint64_t
is_empty_process_stack();

extern void push_process_stack(struct
process_entry);

extern struct process_entry
top_process_stack();
```

```
extern struct process_entry
pop_process_stack();
```

The interface is declared in *globals.h* . Implementation of these function resides in *process_stack.c* which a file I have added to the project. Accordingly, I have edited the *Makefile*. Stack units are in `struct process_stack_element` type. Each unit in this type, has a pointer to another struct with same type. In brief, it is a linked-list implementation of stack. Details about data structures are given in next section.

3.2 Stack Data Structure

```
struct process_stack_element {
        struct process_entry        element;
        struct process_stack_element * next;
};
```

This data structure is a straightforward linked-list node. Linked list holds `struct process_entry` as its main element. This data structures is explained in later sections. At the moment, to know that each process is expressed as one `struct process_entry is enough.`

3.3 Stack Implementation Details

When pop or top instruction is called while stack is empty, /*change code*/ or *kalloc* or *kfree* instruction fails, system halts and related error message is printed to screen.

3.4 Process Data Structure

```
    struct process_entry {
        uint64_t                  id;
        uint64_t        memory_location;
        struct AMD64Context*  context;

        //state will come here
        //threads will come here. May be a
semaphore
    };
```

Each process has an id which is simply the index of the corresponding ELF executable in the list of ELF executables. In addition, process structure contains an address called *memory_location*, which corresponds to start of the memory claimed during ELF file parsing process, in brief segments of the program. This field will be used in *terminate* system call in order to deallocate the occupied memory. In addition to these fields, a *process_entry* structure contains an address of the context of the process.

Once *createprocess* system call occurs, the context structure and *process_entry* structure is created and the former is bind to the latter. After that, this *process_entry* is pushed to the process stack.

3.5 New *copy_ELF* Structure

Since I thought that present condition of copy_ELF did not satisfy my need, I have altered confugiration of it.

```
extern void copy_ELF(const struct
Elf64_Ehdr* const elf_image, uint64_t *
entry_point, uint64_t * memory_block);
```

The function does not return anything, but it updates two fields, *entry_point* and *memory_block*. The function puts the address of the first executable to the *entry_point* parameter and puts the first address of the all memory block occupied for segments to the *memory_block* parameter.

Now it is the time for the reader to see the details of *createprocess* system call.

4. *createprocess* SYSTEM CALL

The algorithm is:

1. Parse ELF image by calling copy_ELF
2. Check validity of parsing
3. Create a new context
4. Initialize rflags and rip registers
5. Create a new *process_entry*
6. Push entry into the stack
7. Set the active context as recently created context
8. Return

Switching to user mode happens in TRAP code. In this operating system, *AMD64_enter_kernel* function fulfills that trapping. After the new context is set in *createprocess*, I need to update the local context pointer in TRAP code.

5. terminate SYSTEM CALL

The algorithm is:

1. Pop process stack
2. Deallocate the context of terminated process
3. Deallocate the memory segments of terminated process
4. Check the emptiness of the stack. If empty, return.
5. Set active context as context of the top process in the stack

6. TESTING createprocess AND terminate SYSTEM CALLS

For testing purposes, we have limited number of methods and environments as the program is an operating system kernel. In this case, the method I have employed is adding my test codes to the somewhere near to the end of the *amd64_system_start* routine. That function is the final step of kernel initialization. After initialization is completed, I create the first process and exit from kernel mode to user mode. I make it by adding these two lines to the end of *amd64_system_start* function:

```
createprocess(0);
returnToUserLevel(getActiveContext(),0);
```

My operating system gives the expected output after adding those two lines to the place stated above.

I haven't carried out a systematic testing for the functions I had implemented. Testing of those functions have stayed in very basic level. Systematic testing and research of different methods to do it has been left as a future work.

7. SCHEDULING

I have implemented round robin scheduling algorithm for executing multiple single-threaded processes. The reason behind this selection is the fact that implementing that algorithm is quite easy and round robin algorithm is obsessively fair [1].

Round robin algorithm is described by Fabio Kon as:

"A small unit of time, called timeslice or quantum, is defined. All runnable processes are kept in a queue. The CPU scheduler goes around this queue, allocating the CPU to each process for a time interval of one quantum. New processes are added to the tail of the queue.

If the process is still running at the end of the quantum, the CPU is preempted and the process is added to the tail of the queue. If the process finishes before the end of the quantum, the process itself releases the CPU voluntarily. In either case, the CPU scheduler assigns the CPU to the next process in the ready queue." [2].

In this sense I have had to re-implement process data structure by replacing process stack with queue in sake of multi-processes support. My structure is completely parallel with the description given above. Whenever an interrupt happens in 32th gate, a static variable increases. When this variable is a multiple of eight, scheduler is called. Therefore, scheduler period is:

$$T_{PIT} = 1 / f_{pit} = 1/200 \text{ sec} = 5 \text{ ms}$$

$$T_{scheduler} = T_{PIT} \text{ x } 8 = 40 \text{ ms}$$

Scheduler is called with 40 milliseconds period. If there is a single process in process queue, scheduler refrains from switch context. Otherwise, each process gets fair CPU service.

7.1 Queue Interface

```
extern struct process_queue_element *
top_process;
extern struct process_queue_element *
back_process;


extern uint64_t
is_empty_process_queue();
extern void
push_back_process_queue(struct
process_entry);
extern struct process_entry
top_process_queue();
extern struct process_entry
pop_process_queue();
```

The interface is declared in *globals.h* . Implementation of these function resides in *process_queue.c* which a file I have added to the project. Accordingly, I have edited the *Makefile*. Queue units are in `struct process_queue_element` type. Each unit in this type, has a pointer to another struct with same type. In brief, it is a linked-list implementation of queue. Details about data structures are given in next section.

7.2 Queue Data Structure

```
struct process_queue_element {
        struct process_entry element;
        struct process_queue_element *
next;
};
```

This data structure is a straightforward linked-list node. Linked list holds `struct process_entry` as its main element.

7.3 Process Data Structure

Compared to the structure in single process case, data structure is almost same. Once multi-threading is implemented, this structure will completely be renovated.

```
struct process_entry {
        uint64_t id;
        uint64_t memory_location;
        struct AMD64Context * context;
        uint64_t state;

        //struct Thread_entry *
threads_linked_list;
        //*threads will come here. May
be a semaphore
};
```

The difference between the structure in section 3.4 is state field. State field is set to RUNNING if the process is currently using CPU. Otherwise, it is set to READY. The first element of the queue always has its state field set RUNNING. The rest of the queue (if there is), has a READY state field. The value changes during context switches.

7.4 Scheduling Algorithm

The algorithm is:

1. If process queue is empty, then do nothing (return).
2. If process queue has a single element, then do nothing.
3. If queue has multiple elements, remove the first element of the queue and push it into back of queue.
4. Set state fields of the first and last element of queue accordingly.
5. Set first element's context as active context.

The selection algorithm has O(1) complexity where there is N processes on the queue;

7.5 Testing Scheduling

I haven't carried out a systematic testing for the scheduling algorithm in context. Testing of whole system has stayed in very basic level. Systematic testing and research of different methods to do it has been left as a future work. The reason of this is the fact that in order to create test cases I need to implement the time system call. This work is also left as a future work.

8. IMPLEMENTATION OF MULTI-THREADING AND MULTI-PROCESSORS

Implementations of multiple threads in a process and using multiple processors are set as a future work.

9. ACKNOWLEDGMENTS

I want to thank to Sven Karlsson, due to his inspiring statements, adored foresight, and encouraging personality.

10. REFERENCES

[1] Tanenbaum, A. S. (2007). Modern operating systems . (3rd ed., p. 154). Upper Saddle River, NJ: Prentice Hall.

[2] Kon, F. [Web log message]. Retrieved from http://choices.cs.uiuc.edu/~f-kon/RoundRobin/node1.html