Operating System Lab Assignment 03: Design and Deploy Syscall, Thread Creation in RAM and Multi-tasking Schedule for DUOS

Dr. Mosaddek Tushar, Professor
Computer Science and Engineering, University of Dhaka,
Version 1.0
Demo Due Date: The following weeks as separate parts.

November 24, 2024

Contents

1 Objectives and Policies **2**
1.1 General Objectives . . . 2
1.2 Assessment Policy . . . 2

2 What to do? **2**
2.1 Implementation of Syscall Exception (SVC) and PendSV . . . 2
2.1.1 Syscall using SVC . . . 2
2.1.2 Task Scheduling using SysTick and PendSV . . . 3
2.1.3 Stack and Heap Memory . . . 3
2.1.4 Syscalls and kernel services . . . 3
2.2 Open File/Resource Table . . . 5
2.3 How does the syscall and context switch work from top to bottom? . . . 6
2.4 What to submit . . . 7
2.4.1 Syscall . . . 7
2.4.2 Task Management . . . 8
2.5 Prototyping, data structure and C code . . . 9
2.5.1 Data Structure for TCB . . . 9
2.6 Testing . . . 10
2.7 How to compile . . . 10

Appendices **11**

A kunistd.h **11**

B syscall_def.h **11**

C DUOS Directory Structure – Tree – Update required **14**

1 Objectives and Policies

1.1 General Objectives

The objectives of the lab assignment are to understand and have hands-on training to understand

1.2 Assessment Policy

The assignment has three level objectives (i) primary objectives, (ii) advanced objectives, and (iii) optional boost objectives. Every student must complete the primary objective; however, they can attempt advanced and optional Boost-up objectives. The advanced objective will be a primary objective in the subsequent assignment. Further, you can achieve five (5) marks for completing the optional objectives and add these marks at the end of the semester with your total lab marks. However, you cannot get more than 100% assigned for the labs. The current lab does not contain advanced or optional boost objectives.

2 What to do?

- Deploy OS service call using SVC and SVCPend
- Deploy Heap and Stack memory and services: SYS_open, SYS_read, SYS_write, SYS_malloc, SYS_free, SYS_fork, SYS_execv, SYS_getpid, SYS_exit, SYS_yield, and SYS_yield
- Develop above kernel services for creating a new thread and process in RAM and Schedule such as: time sharing and FCFS and compare.
- Deploy OS Multi-Tasking with at least 20 (twenty) tasks.

2.1 Implementation of Syscall Exception (SVC) and PendSV

The following description envisions implementing SVC and PendSV to enable unprivileged user applications to access kernel services and schedule user tasks. You must use the class lecture and programming manual to complete the assignment, and note that you must comprehend the underlying notion of the SVC and PendSV.

2.1.1 Syscall using SVC

SVC is an integral part of the ARM processor and exception, enabling the user-to-kernel interface to deliver essential kernel services to the user program. Usually, user programs running in unprivileged mode cannot directly access the connected hardware devices, drivers, and many other processor resources. Therefore, the user program needs a way to get access to available services from the kernel in privileged mode. Almost all microprocessors have the features to facilitate access to kernel functions. ARM processors provide a particular instruction, 'SVC,' to manipulate an exception to get into it. Generally, it is known as SysCall. However, you cannot set any interrupt or exception from other handlers or exceptions; in that case, it generates a HardFault. In this part of the assignment, you must implement syscalls to create an interface between the user program and kernel. Test it from the unprivileged user program.

2.1.2 Task Scheduling using SysTick and PendSV

Next, you must implement the PendSV services for switching between tasks. Every 10ms, the DUOS resumes or executes a task that remains in the task queue. The task queue preserves the task header or TCB containing the last addresses of the active task or task currently awaiting execution. In the operating system, the queue is called the ready queue. We will use the ready queue later in the scheduling assignment. For this part of the lab, you use it for switching between tasks. Therefore you require a kernel stack that will hold the ready queue with the most updated stack addresses of the tasks. The kernel acquires the stackframe address from the ready queue, redirects to the task stack, loads the registers' value, and jumps to the code to execute or resume the execution of a task. For the switching interval, you must use the SysTick exception handler configured to interrupt every 10ms. The SysTick handler (at the tail) initiates the PendSV for a context switch. Usually, we should keep the priority level low for SysTick and PendSV for the smooth operation of the other interrupt ISR. Note that the ISR routine, including SVC and PendSV, must be simple to avoid heavy-weight functions like 'kprintf' or 'kscanf.' The PendSV must perform the context switching to execute or resume the next task.

2.1.3 Stack and Heap Memory

DUOS must include two memory segments: the 'stack' and the 'heap.' These segments are essential for managing kernel functionalities, including stacking kernels, maintaining memory allocation tables, managing open file tables, storing kernel variables, and handling dynamic memory allocation when necessary. The 'stack' segment is utilized for the kernel stack, while the 'heap' segment is used for dynamic memory allocation to create new processes and allocate memory as needed. To effectively implement this, account information is required to identify the amount of memory allocated at a specific start address in the 'heap' and its size. This process relies on a critical component of the DUOS system: the memory allocation table.

Start Address	Size (32-bit word)	Allocated to Process
0x20AF3425AH	100	23
.....	.....	....

2.1.4 Syscalls and kernel services

You must implement the syscall using following three files

- 'syscall_def.h' contains the unique number for each syscall.
- 'syscall.h' contains all prototype or function definitions for the kernel privileged services
- 'syscall.c' includes implementing the function call defined in the 'syscall.h'. Moreover, the 'syscall.c' file consists of the syscall_def.h to get the requested syscall number. Nonetheless, the kernel incorporates various auxiliary files for defining and detailing the actual function tasks.

The 'syscall_def.h' in Appendix B contains the following syscall with unique number. You do not need to implement all the syscall listed here. However, you must implements (i) SYS__exit, (ii) SYS_getpid, (iii) SYS_read, (iv) SYS_write, (v) SYS___time, (vi) SYS_reboot, (vii) SYS_yield, (viii) SYS_malloc, (ix) SYS_free, (x) SYS_fork, and (xi) SYS_execv See the description of the kernel functions below.

- SYS__open: The kernel service opens a device and adds the related record to the device file. The function will increase 't_ref' if the device is already open. The SYS_open function takes exactly two arguments, device symbol (name) and access (t_access) type, and returns the device record index when successful; otherwise, it returns '-1'. The SYS_open must use the ID listed in syscall_def.h.

- SYS__exit: terminate a process and call a SYS_yield() function. Whenever an exit() calls, it will do two tasks (i) change the process state to 'terminated' and call a 'yield()' function. The 'yield' function activates the PendSV exception to run or resume the next task listed in the ready queue. No argument required in the exit function. However, 'exit' function must call the SVC with the appropriate kernel service ID listed in syscall_def.h.

- SYS_getpid: return task_id (given in TCB) of the current task. The application layer function or library function is 'getpid()' returns process or task ID by invoke SVC with the approriate 'SYS_getpid' service ID.

- SYS_read: takes exactly three arguments (i) file descriptor, (ii) input buffer, and (iii) size. The kunitstd.h (Appendix A) contains the default 'STDIN_FILENO' file descriptor when user application kscanf calls the SYS_read function. The size parameter defines the maximum input size in bytes. However, the termination character, such as '\n,' determines the input size. If there is no termination character, then the maximum input limit is 'size.' You may define the largest input size as 256 bytes. Beyond this, the read function will ignore the characters and takes precisely 256 bytes as the input. In the current setting SYS_read with the 'STDIN_FILENO' descriptor, use _USART_READ to read from the character terminal. The application layer library is 'read(fd,buff,size)'. The application library function call SVC with the argument listed before and the 'SYS_read' syscall ID given in syscall_def.h. Note that the first argument should be the service ID.

- SYS_write: This function takes exactly three arguments to write bytes to the targeted file. The arguments are (i) file descriptor, (ii) out buffer, and (iii) size. When print use this system to display the characters in the terminal (character terminal or display monitor), then the kprintf passes 'STDOUT_FILENO' as the file descriptor. In the current setting SYS_write with the 'STDOUT_FILENO' descriptor, use _USART_WRITE to send a string to the character terminal. The application calls the application/library layer function 'write(fd, buff, size)'; however, the write function uses the SVC to call kernel services 'SYS_write' argument beginning with a service ID and three given function arguments.

- SYS___time: Returns the current elapsed time of systick in milli-second. The application layer function name is 'getSysTickTime()'.

- SYS_reboot: This service call to reboot or restart the microcontroller. The application layer function name is 'reboot.'

- SYS_yield and yield The system must make the yield() function available for user-level execution. The user application may voluntarily call of yield() function to pass the CPU to the next task. In this case, the yield() directly syscall the SYS_yield. The yield function calls SVC with the service ID.

- SYS_close The system must make the close() function available for user-level execution. The user application may voluntarily call the close() function to close the open device file in the

dev_table when t_ref is zero. The close() function from the user layer calls the SYS_close to do the job.

- SYS_malloc The function allocates a memory block on the heap and returns a pointer to it. The requester can cast the allocated memory into any desired data type. However, when allocated, the memory is uninitialized, which means its values are unpredictable. Additionally, the function impacts the memory accounting table above. Returns address when success otherwise '0'.

- SYS_free When the memory allocation is no longer required, the corresponding pointer is sent to the SYS˙free function to release and deallocate that memory. This process directly impacts the memory accounting table, as it removes the relevant entry associated with that memory allocation, ensuring that resources are accurately tracked and freed for future use. Returns '0' when success otherwise '-1'.

- SYS_fork This system call creates a new process by duplicating the calling process. The new process is an exact copy of the parent process but has its own address space and memory. This duplication means both processes have separate stacks and memory areas for their variables. The system achieves this by copying the parent stack into a newly allocated child stack, and it also creates a new Thread Control Block (TCB) with a unique process ID that points to the new stack. This duplication affects the memory accounting table and the file table as well. The return value is the child process ID for the parent and '0' for the child process. (Delay this implementation for the next assignment part).

- SYS_execv This function accepts two parameters: (i) the name of the executable file and (ii) a set of arguments to that executable. Its primary purpose is to replace the existing executable or binary object in the current process's memory. Upon successful execution, the function does not return any value; however, if an error occurs, it will return '-1'. (Note: you delay the implementation may in a subsequent assignment.) In this scenario, the function uploads the executable code, stored in flash memory, into the RAM of the current process and initiates its execution.

2.2 Open File/Resource Table

Operating systems generally creates many tables and bit maps to keep track of the open files, currently running processes, other resources. This assignment must implement a simple file table to keep track of the open files in global kernel space. The data structure of the open device (file) table is

```
typedef struct  dev_t
{
        char name[32]; // Device name or symbol
        uint32_t  t_ref; //Number of open count
        uint8_t t_access; //open type  O_RDONLY, O_WRDONLY, O_APPEND
        uint32_t *op_addr; //Address of the datastructure operations
}dev_table
```

The system dev_table maximum size is 64, which implies that the table can hold 64 device information. The 'name' contains the unique symbol for the device, and '*op_addr' is the address of the operations available to use the device. The first three device descriptor are STDIN_FILENO,

STDOUT_FILENO, and STDERR_FILENO. The following records are devices like seven segment display, input switch, etc. 't_ref' open count; this value increases and decreases for each opening and closing of the device. System devices 't_ref ' never be zero for convenience. However, the 't_ref' value '0' delete the record from the device open table. The above function, 'SYS_,' gets the device address from the device open table in the memory. It would be best if you created the device open table when booting. The initial table contains at least three records, as described before.

2.3 How does the syscall and context switch work from top to bottom?

After resetting or power-on, our DUOS initializes the required drivers, update open file/resource table (dev_table), setup necessary interrupts, exceptions, task stack, stackframe, and TCB entry in the ready queue, and changes the task status to new. Next, the DUOS switches to user mode and starts the current task in the ready queue. Any task mandating kernel services, including reading and writing, uses the SVC instruction to switch to the kernel (privileged mode) and get serviced. Suppose the user application calls 'kprintf', then all the algorithms related to the 'printf' must be complete (conversion to string or char) first. Then the 'printf' calls the library function 'write' with the previous argument. In the case of 'printf' the 'fd' is 'STDOUT_FILENO.'

Consequently, the write function calls the SVC as described before. In the SVC handler, the kernel saves the registers (if required) and redirects to the syscall function. In the syscall.h, the syscall function selects and calls the actual function according to the syscall ID. The first argument of the SVC is the service ID or syscall ID specified in syscall_def.h. Following fig. 1 shows the syscall (SVC) steps in short. In this case the 'write' utility function use SVC to get access to the 'SYS_write' kernel service.

For example: 'printf'. For SVC follow slides in lecture 05 for detail

```
function printf(args ... ){
        string s = convertToSTR(args ...) //see your kprintf implementation
        return_code=call write(s);
}

function write(File_descriptor fd, char* s,size_t){
        //stacked the arguments
        //first argument must be SYS_write service ID
        SVC call for SYS_write
}

function syscall(){
        1. use switch case to determine the actual function
        based on SVC service ID (first argumant) and call the function
        2. Return with exec_return code
}
function __sys_write(args ..){
        do the actual job with the USART driver
}
```

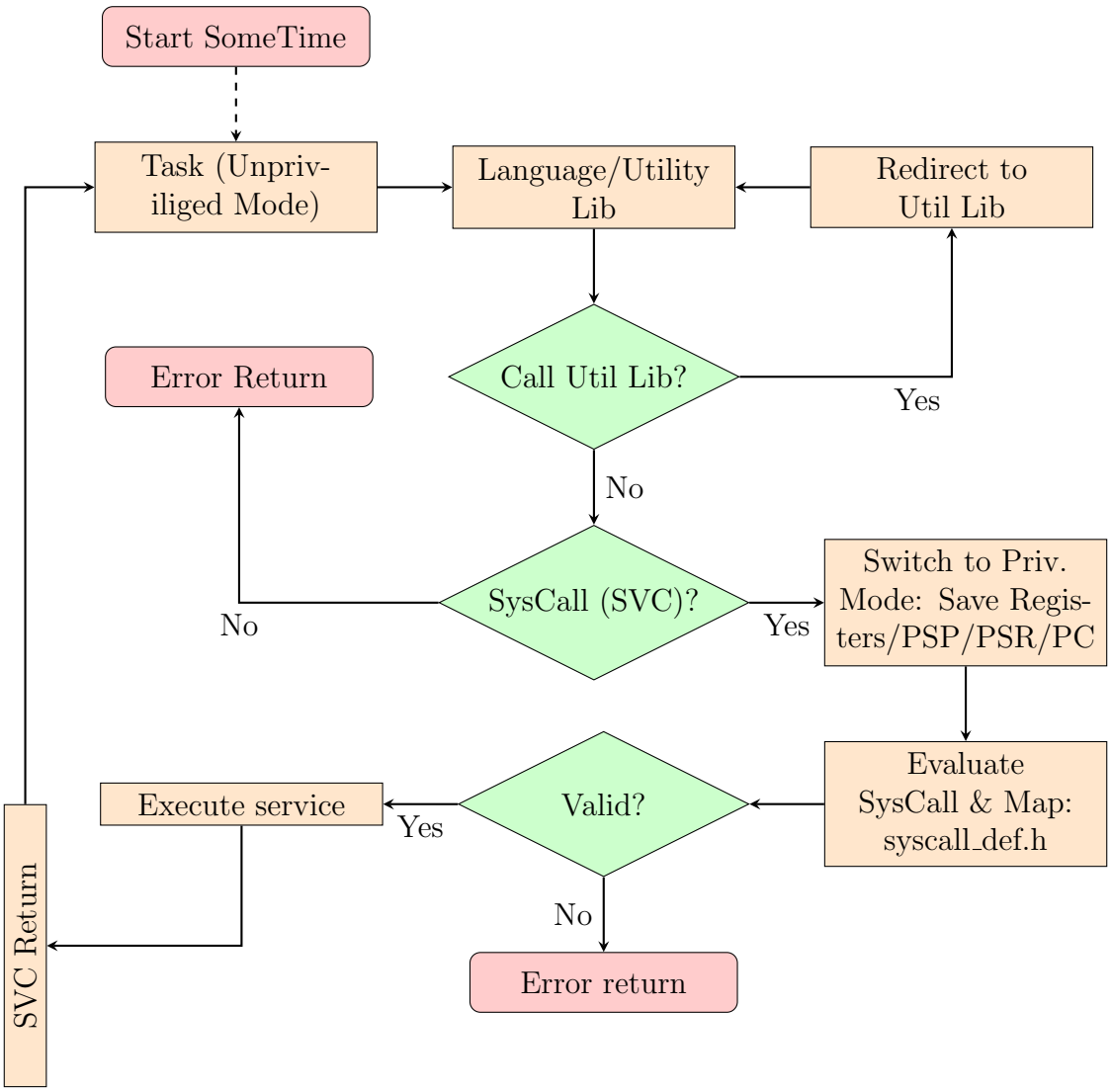

Figure 1: Syscall Steps in short

2.4 What to submit

The assignment envisions creating a set of functions and policies to carry out the service call and task management with Round-Robin (timesharing) policies. The work contains two parts, (a) syscall and (b) task management, given below.

2.4.1 Syscall

Implement high-level test code to evaluate the SVC and test all SYS_*. The implementation must contain (i) high-level functions (such as 'printf', convert to string and call write utility function with an appropriate file descriptor and generated string), (ii) application layer utility functions, such as write, (iii) syscall function (susch as SYS_write) in syscall.c and finally, kernel service functions. The exec_return code either returns a success or an error code. The list of functions you need to implement in different level of system access modes are:

Full Spectrum Syscall Implementation				
Unprivileged Mode		Privileged Mode		
User Function	**Utility Library** (Optional)	**Service ID**	**Actual Function**	**Driver or service**
fopen(···)	open("symbol", t_access)	SYS_open	__sys_open	initialize the device or increase t_ref
printf(···)	write(fd,data,size_t)	SYS_write	__sys_write	UART (if fd == STDOUT_FILENO[a])
scanf(···)	read(fd,data,size_t)	SYS_read	__sys_read	STDIN_FILENO
reboot()	sys_reboot()	SYS_reboot	__sys_reboot	NMI (reset)
exit()	sys_exit()	SYS_exit	__sys_setTaskStatus	Terminate (TCB status)
getpid()	sys_getpid()	SYS_getpid	__sys_getpid	TCB/PCB – task_id
gettime()	sys_gettime()	SYS___time	__sys_gettime	Systick Time
yield()	–	SYS_yield	force PendSV	PendSV for reschedule
fclose()	close(···)	SYS_close	__sys_close	delete record from device table or decrease t_ref
fork()	fork()	SYS_fork	__sys_fork	allocates memory for stack and a TCB, affects memory accounting
malloc()	malloc(size_t)	SYS_malloc	__sys_malloc	affects memory accounting table (add entry)
free()	free()	SYS_free	__sys_free	delete entry from memory accounting table
execv()	execv(file*,args*)	SYS_execv	__sys_execv	execute new process and overwrite the existing process

[a] In our current i/o [console] connections

Deploy actual functions: __sys_write(···), __sys_read(···), __sys_setTaskStatus(···), __sys_getpid(···) in kunistd.h and kunistd.c and __sys_gettime(···) in ktimes.h and ktimes.c. If you use the SysTick for timesharing/round-robin policy, then enable PendSV in SysTick. Deploy 'yield(···)' in 'schedule.h' and 'schedule.c'. The schedule.c/h must have a function 'schedule(···)' for implementing a scheduling policy. The 'schedule(···)' function save the context of the current process and load the context of the next process for resume/start executing. The PendSV will use the 'schedule(···)' for the context switch.

Note: At the beginning of assignment 3, add your NVIC functions prototypes in cm4.h and NVIC functions to the cm4.c file. Also, remember that you cannot enable or set any interrupt/exception from any interrupt handler other than the PendSV. Exception SVCall must not call from another interrupt/exception handler. For this assignment, you must call SVC from the unprivileged access mode of Cortex-m4. However, you can call SVC from the privileged access mode as well. The return must switch the processor to the calling access mode. Thus the SVC call from an unprivileged mode must return to the unprivileged processor access mode.

2.4.2 Task Management

Create three tasks to modify a global variable say 'count'. The variable 'count' intilizes to '0'. Each of the tasks increase the 'count' by '1' (until $count < 10000000$) as following:

```
uint32_t value;
uint32_t inc_count=0;
volatile uint32_t count=0;
 while(1){
        value=count;
        value++;
        if(value != count+1){ //we check is someother task(s) increase the count
                printf("Error %d != %d\n",value,count+1); /* It is an SVC call*/
        }else{
```

```
                count=value;
                inc_count++;
        }
        if(count >= 10000000){
                uint16_t task_id = getpid(); /* It is an SVC call*/
                /* display how many increments it has successfully done!! */
                printf("Total increment done by task %d is: %d",task_id,inc_count);
                /* above is an SVC call */
                int fd=fopen("S_DISPLAY",O_WRDONLY);
                 /* int x declare as local */
                fprintf(fd,"%d",x);
                x=(x+1)%8;
                fclose(fd);
                break;
        }
}
```

2.5 Prototyping, data structure and C code

Appendix C shows the directory tree structure of the DUOS. Use the 'tree duos' shell command from Linux to get the detailed directory tree of the duos source code. You can use a part of the algorithms for kprintf and kscanf to implement user-level printf and scanf in kstdio.c/h.

2.5.1 Data Structure for TCB

You must impelement the data struture for the task as,

```
struct t_task_tcb{
        uint32_t magic_number; //here it is 0xFECABAA0
        uint16_t task_id; //a unsigned 16 bit integer starting from 1000
        void *psp; //task stack pointer or stackframe address
        uint16_t status; //task status: running, waiting, ready, killed, or terminated
        uint32_t execution_time; //total execution time (in ms)
        uint32_t waiting_time; //total waiting time (in ms)
        uint32_t digital_sinature; //current value is 0x00000001
} TCB_TypeDef
```

To To enforce round-robin scheduling, create a ready queue in the kernel (privileged mode) to store the TCB. The scheduler (PendSV and 'schedule') scans the ready queue to select and execute a task. Therefore the kernel process must have a stack of 4kB. A process or task must switch to the privileged mode to access the ready queue. Thus the function 'sys_getpid' or 'sys_exit' in unprivileged mode requires an SVCall. Each task stack size is 1kB. The process and the kernel use stack to store local variables and task stackframe, as shown in fig. 2.

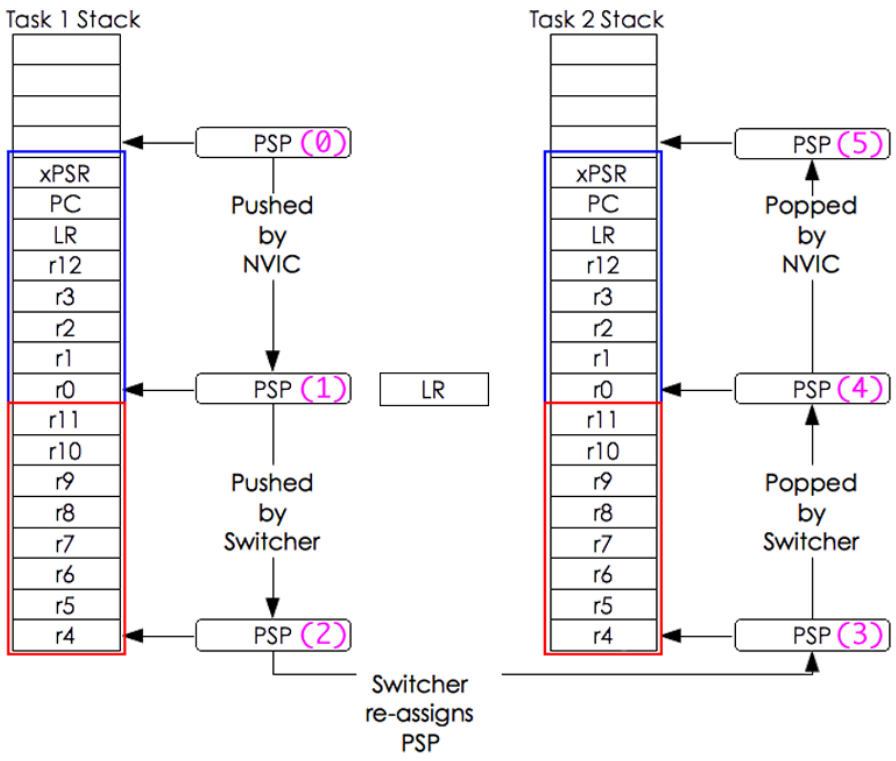

Figure 2: Task stack

After resetting or rebooting, the system initializes as it does before (assignment 1 & 2). The 'kmain' function creates the ready queue and tasks. Then switches to the unprivileged mode and executes (or resumes) the current (first) task. At the end of each time slice (SysTick), the kernel (PendSV and scheduler function) switch to the next task (ready state) and executes. The task may contain/require an SVC call such as printf in the increment of 'count' (above). Before switching to a new task, the scheduler must update the leaving status (ready or terminated) and enter tasks (running). The kernel displays the 'count' after finishing all tasks and stops.

2.6 Testing

You must write a user-level test program to verify the SVC call for printf, scanf, reboot, exit, gettime, getpid, and yield functions. For the PendSV, you must implement at least three tasks to display the statistics of the number of increment

2.7 How to compile

Review the 'Makefile' in the 'compile' directory and comprehend the make rules. Go to the 'duos/src/compile' directory and execute the 'make all' command to create/update the binary in the 'target' and map file in 'mapfile' directory. The binary (downloadable) file's current name is 'asst03'. To download the 'asst03' to the controller flash memory, execute 'make load.' The load rule uses 'openocd' with two config files. Modify according to your need. Revise the copy of your Makefile to add the new file(s) required to complete the assignment.

Appendices

A kunistd.h

```
#ifndef _KERN_UNISTD_H_
#define _KERN_UNISTD_H_

/* Constants for read/write/etc: special file handles */
#define STDIN_FILENO  0      /* Standard input */
#define STDOUT_FILENO 1      /* Standard output */
#define STDERR_FILENO 2      /* Standard error */

#endif /* _KERN_UNISTD_H_ */
```

B syscall_def.h

```
#ifndef _SYSCALL_DEF_H_
#define _SYSCALL_DEF_H_

#define SYS_fork         0
#define SYS_vfork        1
#define SYS_execv        2
#define SYS_exit        3
#define SYS_waitpid      4
#define SYS_getpid       5
#define SYS_getppid      6
//                             (virtual memory)
#define SYS_sbrk         7
#define SYS_mmap         8
#define SYS_munmap       9
#define SYS_mprotect     10
//                             (security/credentials)
#define SYS_umask        17
#define SYS_issetugid    18
#define SYS_getresuid    19
#define SYS_setresuid    20
#define SYS_getresgid    21
#define SYS_setresgid    22
#define SYS_getgroups    23
#define SYS_setgroups    24
#define SYS_getlogin   25
#define SYS_setlogin   26
//                             (signals)
#define SYS_kill         27
#define SYS_sigaction    28
#define SYS_sigpending   29
```

```
#define SYS_sigprocmask  30
#define SYS_sigsuspend   31
#define SYS_sigreturn    32
//                              -- File-handle-related --
#define SYS_open         45
#define SYS_pipe         46
#define SYS_dup          47
#define SYS_dup2         48
#define SYS_close        49
#define SYS_read         50
#define SYS_pread        51

#define SYS_getdirentry  54
#define SYS_write        55
#define SYS_pwrite       56

#define SYS_lseek        59
#define SYS_flock        60
#define SYS_ftruncate    61
#define SYS_fsync        62
#define SYS_fcntl        63
#define SYS_ioctl        64
#define SYS_select       65
#define SYS_poll         66

//                              -- Pathname-related --
#define SYS_link         67
#define SYS_remove       68
#define SYS_mkdir        69
#define SYS_rmdir        70
#define SYS_mkfifo       71
#define SYS_rename       72
#define SYS_access       73
//                              (current directory)
#define SYS_chdir        74
#define SYS_fchdir       75
#define SYS_getcwd     76
//                              (symbolic links)
#define SYS_symlink      77
#define SYS_readlink     78
//                              (mount)
#define SYS_mount        79
#define SYS_unmount      80

//                              -- Any-file-related --
#define SYS_stat         81
#define SYS_fstat        82
#define SYS_lstat        83
```

```
//                              (timestamps)
#define SYS_utimes       84
#define SYS_futimes      85
#define SYS_lutimes      86
//                              (security/permissions)
#define SYS_chmod        87
#define SYS_chown        88
#define SYS_fchmod       89
#define SYS_fchown       90
#define SYS_lchmod       91
#define SYS_lchown       92
//                              -- Sockets and networking --
#define SYS_socket       98
#define SYS_bind         99
#define SYS_connect      100
#define SYS_listen       101
#define SYS_accept       102
#define SYS_socketpair 103
#define SYS_shutdown     104
#define SYS_getsockname  105
#define SYS_getpeername  106
#define SYS_getsockopt   107
#define SYS_setsockopt   108

//                              -- Time-related --
#define SYS_time       113
#define SYS_settime    114
#define SYS_nanosleep    115

//                              -- Other --
#define SYS_sync         118
#define SYS_reboot       119
#define SYS_yield        120
/*CALLEND*/

#endif
```

Alert: You can discuss with your classmates, however, do not copy code from others.