Phase 2 - Kernel Services

Due Oct 31 by 11:59pm **Points** 100

Project Repositories

Teams will use existing team project repositories for development and project submissions.

Project Phase Submissions

Submissions must be on the main branch (teams are welcome to create other branches for individual team purposes) but ONLY the main branch will be considered for submissions.

Submissions for phase 2 must be tagged as such:

- phase-2.0

You may re-tag any commit as a submission. If you have additional submissions PAST the due date with corrections to be made, you may tag them with a sub version, such as:

- phase-2.1
- phase-2.2
- …

Phase 2 Goals and Requirements

In this project phase, we extend our kernel bootstrap to implement kernel services that user processes may use via system calls.

The services that we will implement in this project phase include:

- Obtaining information about the system:
  - Current system time
- Obtaining information about the running process:
  - Process ID
  - Process Name
- Adding new kernel functionality
  - Adding the ability for a process to "sleep"

- Ability for a process to exit itself

In addition, we will make some enhancements to our kernel operation:

- Increase the number of processes that can actively run (5 -> 20)
- Creation of a new "idle" queue so that the idle task will only run if no processes are running
- Add a new process state for when a process is sleeping

Some recommendations:

- Remove debug_printf() calls for previous phases/implementations

Project Phase 2 Demo

You may download a demo phase 2 image for testing via:

https://athena.ecs.csus.edu/~cristg/classes/2020/fall/159/project/phase2/DemOS.dli (https://athena.ecs.csus.edu/~cristg/classes/2020/fall/159/project/phase2/DemOS.dli)

File Layout/Organization

In addition to your current set of files, you will be adding four new files, outlined below.

Filename	Scope/Context	Description
syscall.h	User	Declares the system call APIs that will be made available for user processes
syscall.c	User	Implements the system call triggers
ksyscall.h	Kernel	Declares the kernel system call handlers
ksyscall.c	Kernel	Implements the kernel system call handlers

Pre-requisites:

To begin phase 2, you will need to apply some changes to your repository from phase 1.

- Within your repository, run the following to download/apply new files:

```
wget -O - http://athena.ecs.csus.edu/~cristg/classes/2020/fall/159/project/phase2/setup.sh 2>/dev/null | b
ash
```

- Confirm that no errors were encountered.
- Review the git log to see the new additions/changes

```
git log
```

- Commit the changes:

```
git commit -m "Phase 2 pre-requisites"
```

Project Phase 2 Implementation

Data types and structures

In addition to the existing data types and structures from phase 1, you will be adding some new items and modifying or extending additional items.

Enumerations

The following new enumerations will be created to define the possible system call identifiers:

```
// Syscall definitions
typedef enum {
    SYSCALL_PROC_EXIT,
    SYSCALL_GET_SYS_TIME,
    SYSCALL_GET_PROC_PID,
    SYSCALL_GET_PROC_NAME,
    SYSCALL_SLEEP
} syscall_t;
```

The process state enumeration will be extended to include a new "SLEEPING" state:

```
// Process states
typedef enum {
    AVAILABLE,
    READY,
    RUNNING,
    SLEEPING
} state_t;
```

Data Types

No new data types will be created during this project phase.

Data Structures

The following new data structure instances will be created:

```
extern queue_t idle_q;
extern queue_t sleep_q;
```

The idle queue will be used for the kernel idle task, such that it will only run if there are no other processes running.

The sleep queue will contain a list of processes that are in the sleeping state.

The following items will be added to the process control block:

```
// The time when a process is done sleeping
int wake_time;
```

Kernel Initialization

The following must be implemented during initialization:

- During kernel initialization, all new data structures need to be initialized.
- New interrupts/entries need to be added to the Interrupt Descriptor Table
- The idle task should be executed using the idle queue instead of the run queue

Kernel Run Loop

The following must be implemented during run-time

- New interrupts/routines need to be handled when processing interrupts

Interrupt Behavior

kisr_syscall()

This is a new interrupt service routine to process the system call software interrupt

- Detects the system call that was performed (via the eax register)
- Calls the appropriate system kernel function
  - May be existing kernel function (such as kproc_exit())
  - May be a new kernel function (such as ksyscall_get_proc_pid())

kisr_timer()

The timer interrupt must be modified so that when it is triggered, it can determine which processes need to be woken up. It should:

- Inspect the sleep queue to determine which process(es) should be woken up
  - Need to move to the process to the run queue
  - Need to set the state accordingly
- Move any process that is not yet ready to wake up back into the sleep queue.

System Calls (User-Context)

New in this project phase is adding an initial set of system call APIs that user processes may call. The user-context APIs are all located/coded in 'syscall.c'.

The "get_proc_pid()" system call has been implemented for you. Examples of different implementations of a system call have been provided as comments.

You must implement the remaining system call APIs:

- proc_exit
- get_sys_time()
- get_proc_name()
- sleep()

proc_exit() System Call

This system call will cause the calling process to exit.

get_sys_time() System Call

This system call will return the current system time (in seconds) to the calling process.

get_proc_pid() System Call

This system call will return the process id (pid) of the calling process.

get_proc_name() System Call

This system call will return the process name of the calling process.

sleep() System Call

This system call will cause the running process to "sleep" for a specified number of seconds by moving it into the sleep queue. The process will "wake up" once the specified number of seconds has passed. During this time, the process will not actively execute any instructions.

System Call Handlers (Kernel Context)

The kernel system call handlers are located/coded in 'ksyscall.c'. These functions are the kernel-side component of the system call and run in kernel-context.

The "ksyscall_get_proc_pid()" kernel system call handler has been implemented for you.

You must implement the remaining kernel system call handlers:

- ksyscall_get_sys_time()
- ksyscall_get_proc_name()
- ksyscall_sleep()

If you notice, there is no stub in ksyscall.c for the kernel-side component for the "proc_exit()" system call. Consider what existing kernel functions might be of use to facilitate the functionality, or consider writing a new ksyscall_proc_exit() handler that would trigger the kernel-side behavior.

Process Scheduler

In project phase 1, we made use of only a single run queue. In project phase 2, we will make use of the new idle queue to run the idle process in the event that no other processes are running. Modify kproc_schedule() to add this functionality.

If you have not done so already in project phase 1, make sure that your scheduler utilizes the queue that is associated with the process (via the pcb data structure) vs hard-coding the run queue when a process is unscheduled.

Idle Task

In project phase 1, our idle task performed a simple "busy-delay" loop. However, busy-delays make for an inefficient use of the CPU, and also due to the implementation, perform unnecessary sets of instructions. To account for this, we will utilize an x86 instruction to halt the CPU when the idle task runs, until the timer interrupt occurs again to wake up the CPU.

Modify the idle task to:

- Remove any printing/display of output during the run loop
- Replace the busy loop/delay with a single inline assembly instruction: hlt

User Process

In project phase 1, the user process was a simple loop that indicated it is running. In this project phase, we will expand the user process to add new functionality. Most importantly, we will want for it to make use of each of the system calls that have been implemented:

- get_sys_time()
- get_proc_pid()
- get_proc_name()
- sleep();

When the user process starts, it should print the current system time, it's process ID, and it's process name.

Based upon the process ID, calculate a sleep time using the following formula:

- 2 x the process ID, modulo 10, + 1

For example:

- PID 1 -> sleep time is 3 seconds
- PID 3 -> sleep time is 7 seconds
- PID 5 -> sleep time is 1 second
- PID 12 -> sleep time is 5 seconds

In the process run loop, it should check if the process has run for more than 20 seconds. If so, it should print a message and then exit.

Otherwise, the process should continually print the current system time, it's process ID, and it's process name, the calculated sleep time. Subsequently, it should sleep for the calculated sleep time.

Process Lifecycle