MULTITHREADED, TIME-SHARING KERNEL FOR RISC-V ISA

C/C++ Faculty project

Bogdan :: markovicb1
School of Electrical Engineering, University of Belgrade
August 2022

Abstract

This project represents simple, yet fully functional kernel for RISC-V architecture. It is completely made from scratch, meaning it fully lies on the most basic concepts of C/C++ programming languages. Additionaly, no external (nor internal) library is used in creation process. The whole project was sophomore year's biggest obstacle, which was covered on Operating Systems 1 course at School of Electrical Engineering, University of Belgrade.

Contents

1 Before reading

This documentation is completly made from the README.md file available on GitHub repo, which means that there is no real difference between these documents.

This documentation was made using LaTeX on *Overleaf*, with the help of **Math Notes Template**. For any kind of info you can't find in this documentation nor in README.md file in GitHub repo, feel free to contact.

2 Kernel overview

A *kernel* is a fundamental component of an operating system that acts as a *bridge between the hardware and software layers*. It serves as the **core of the operating system**, responsible for *managing system resources*, such as memory, CPU, and input/output devices. The kernel provides essential services and functionalities to enable various software applications to interact with the hardware in a controlled and efficient manner.

Furthermore, the kernel plays a **crucial role** in *multitasking*, allowing multiple programs to run concurrently by allocating and managing processor time efficiently. It also handles tasks like *process scheduling, memory management, and device driver management*. The kernel's stability and performance directly impact the overall reliability and speed of the operating system.

This simple one has several functionalities that act the behaviour of the real ones:

- Memory allocator
- Threads
- Semaphores
- HW and SW interrupts
- System and user modes
- Process FIFO Scheduler

3 Important technical notes related to the kernel

userMain is used as ordinary main in other projects, where main in this case is used only in system mode (regime), it prepares whole kernel for user activities - initialization of memory and preparing for userMain and user mode.

Kernel and user app are compiled as one .exe file which is then run on the host OS - **modified version of MIT's xv6 OS**. Using *QEMU emulator*, xv6 creates virtual space for our .exe file, making it see the whole system as it's own HW with one whole available memory space.

Kernel supports 3 different layers connected to users app:

1. Application Binary Interface / **ABI**
2. C Application Programming Interface / **C API**
3. C++ Application Programming Interface / **C++ API**

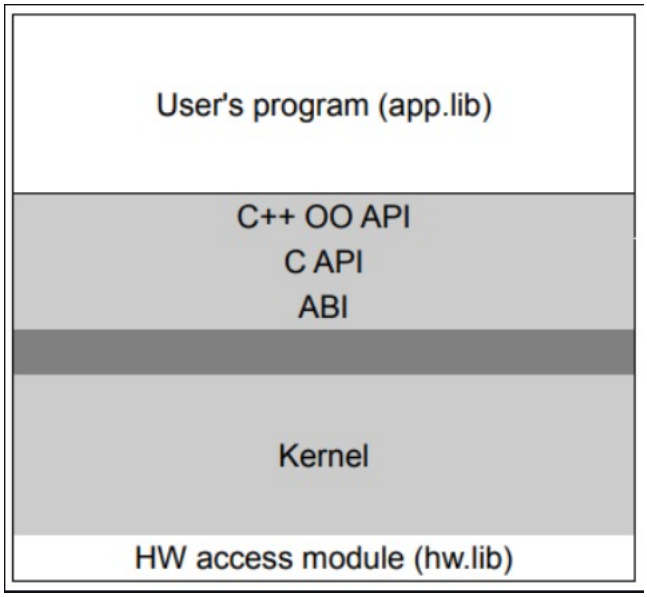

Figure 1: Abstract kernel layers

4 Application Binary Interface - ABI

ABI is **the lowest layer** in kernel hierarchy and is completly made for system mode, or more precisely, it is set of system calls that are used in higher levels. It transfers all the variables and return values from upper level to the processors registers.

ABI is made of the following classes:

- **MemoryAllocator** - for dynamic allocations in user apps and for initial instatiation of whole memory
- **Machine** - set of methods that are composed of assebly code, working with processors registers
- **threadHandler** - handler class that wraps real thread procedures, made in C API
- **semaphoreHandler** - handler class that wraps real sempahore procedures, made in C API

4.1 Memory Allocation

The smallest blocklike unit in memory is object of structure called *MemoryBlock*, which stores info about how much of size (size_t, in bytes) does the block use (without structures header), and a pointer to next available MemoryBlock.

Initialy, MemoryAllocator allocates whole memory as one available block.

MemoryAllocator uses 2 fundamental methods used in all the upper layers as well as in the user apps:

1. **void* mem_alloc(size_t sz)** - returns pointer to the newly allocated block in memory, or returns nullptr if some error occured. It uses *FirstFit algorithm* for block choosing. More on this algorithm can be read here.
2. **int mem_free(void* ptr)** - returns 0 if everything is good, in other case it returns corresponding error code. It is a must to pass a pointer of previously allocated block from mem_alloc to this function in order for this function to work properly! mem_free appends the block pointed by the passed argument to the tail - the last available block tracked in MemoryAllocator.

Return type	Name	About
const void*	HEAP_START_ADDRESS	Start address of the virtual memory
const void*	HEAP_END_ADDRESS	End address of the virtual memory
const size_t	MEM_BLOCK_SIZE	The unit used for memory allocation

4.2 Interrupt handling

Every computer system has SW and HW interrupts, as a way to do various different tasks. Kernel handles interrupts in a void Machine::handleSuperVisorTrap() method, called from the trap routine (assembly code1). It distinguishes several interrupt causes, here are some:

- Timer interrupt, for time-sharing processor context change
- Console interrupt
- External hardware interrupt
- Illegal instruction
- Read permission denied
- Write permission denied
- Software interrupt, the most used interrupt since it calls void Machine::callerFunction(...), the true wrapper that calls particular system call according to the function code passed as the first argument

More on computer interrupts can be read here.

5 C Application Programming Interface - C API

C API is a **layer above** the ABI. It consists of the classic methods as wrappers of ABI methods, making it an *ABI wrapper*.

C API provides *classical, procedural programming interface* of system calls in C language. It implements methods described in the following table:

Code	Function	About
0x01	void* mem_alloc(size_t size);	Allocates at least *size* bytes
0x02	int mem_free(void* ptr);	Frees memory pointed by ptr
0x11	int thread_create(thread_t* handle, void(*start_routine)(void*), void* arg);	Starts thread on *start_routine*
0x12	int thread_exit();	Kills current thread and changes processor context
0x13	void thread_dispatch();	Potentially changes context of processor
0x14	int thread_create_non_scheduler(thread_t* handle, void(*start_routine)(void*), void* arg);	Creates thread but doesn't store it in Scheduler
0x21	int sem_open(sem_t* handle, unsigned init);	Creates Semaphore with initial value of init
0x22	int sem_close(sem_t handle);	Frees Semaphore and empties Wait Queue
0x23	int sem_wait(sem_t handle);	Does wait operation on given semaphore
0x24	int sem_signal(sem_t handle);	Does signal operation on given semaphore
0x25	/	Special code - Kernel enters system (privileged) mode

6 C++ Application Programming Interface - C++ API

C++ API is a layer above the C API. It provides object oriented programming interface for Threads and Semaphores, as well as OOP C++ dynamic memory managment.

```
class Thread{
public:
    Thread(void (*body)(void*),void* arg);
    virtual ~Thread(){}

    int start();
    static void dispatch();
protected:
    Thread();
    virtual void run(){}
};
```

Thread class supports 2 ways of creating Threads:

- POSIX way: creating thread also starts it
- Object oriented way: thread starts with start() method

Semaphores:

```
class Semaphore{
public:
    Semaphore(uint init = 1);
    virtual ~Semaphore(){}
    int wait();
    int signal();
};
```

7 Threads implementation

The core functionality of threads is behind the internal class called TCB. TCB stands for **Thread Controll Block** and works together with System Scheduler.

7.1 Scheduler

Scheduler is implemented as Singleton class for scheduling of active threads. It is made of dynamic list seen as Queue for ready threads. It uses 2 simple methods: *get* and *put*, which dynamicly unblock/block threads. Scheduler uses FIFO algorithm.

7.2 TCB

TCB provides different static and non-static methods necessary for thread creation, deletion and watch:

- **TCB(Body body, uint64 timeSlice, void * args)** - makes active thread with corresponding function body and its arguments, as well as timeSlice, and puts it in the Scheduler
- **createThreadNonScheduler(Body body, void *args)** - does the same job as previous constructor without puting thread in Scheduler. This is used for semaphores
- **yield()** - change of processors context on request. Scheduler puts current thread in, and chooses new one according to FIFO algorithm
- **isFinished()**
- **setFinished(bool value)**
- **getTimeSlice()**

TCB tracks which thread is *current*. Context change is completly processed in assembly block.

8 Semaphore implementation

A *semaphore* is a **synchronization mechanism** used in computer science to *control access to a shared resource* by multiple processes or threads in a **concurrent system**. It acts like a counter that restricts the number of entities that can access a particular resource at any given time. Semaphores maintain a non-negative integer value and support two fundamental operations: **wait** and **signal**.

Semaphores are crucial for *preventing race conditions* and ensuring proper synchronization in multi-process or multi-threaded environments. They are often used to control access to critical sections of code, manage buffers, and coordinate the execution of concurrent tasks. The core functionality of semaphores is behind the internal class called **ksemaphore**, where prefix k represents kernel, meaning function **works in privileged mode**.

Sempahores in this kernel work as expected: there are simple methods for signal and wait. Each semaphore has its own queue for blocked threads.

9 Differences between the modes

There are 3 modes in RISC-V, though only 2 are used in this kernel:

- **User mode** / Unprivileged regime
- **System mode** / Privileged regime

While System mode has permission to *access every register and use every instruction in processor*, User mode is restricted. **The kernel is entirely made to be run in system mode, as well as the main method**.

For that reason, userMain method is made and set in different source file. *User mode for userMain is prepared in main method so all the threads made in userMain can run in Unprivileged regime*, and main method remain as priviliged one.

When all unprivileged threads from userMain are finished, userMain finishes and changes context to main method, which terminates the whole kernel.

10 See more about the project

The whole text project is available in **serbian langauge** on courses website in section Project.

It is recommended to understand basic terms and concepts used in this project, such as:

- Memory managment and organisation
- Basics of assembly for RISC-V (available in documentation previously mentioned)
- Processes and threads
- Process sync using Semaphores

If you are not familiar with any of these terms, feel free to search for it on sites such as geeksforgeeks or youtube. There are also free courses over the web so search for them too.

11 Prepare project for use

1. **Get CLion** as it is shown that this IDE works great with qemu emulator and xv6 OS
2. **Install these Linux packages**:
   - build-essential
   - qemu-system-misc
   - gcc-riscv64-linux-gnu
   - binutils-riscv64-linux-gnu
   - gdb-multiarch
   - g++-riscv64-linux-gnu
3. **Get the project** using

   ```
   git clone https://github.com/markovicb1/riscv-kernel.git
   ```

   Open the folder in CLion
4. There are several **make commands**:
   - make qemu - used to start the kernel
   - make qemu-gdb - used for debugging (uses GNU debugger)
   - make all - used to compile project, produces kernel.asm
   - make clean - cleans result of make all and several extra files

   For the first time, the order should be: clean - all - qemu
5. In order to run particular test, such as C Threads test, **corresponding method in userMain should be decommented.**