# Design of Mini Multi-Process Micro-Kernel Embedded OS on ARM

Bo Qu
School of Mathematics and Information Technology
Nanjing Xiaozhuang University
Nanjing, China
e-mail: Mr.QuBo@126.com

Zhaozhi Wu
School of Mathematics and Information Technology
Nanjing Xiaozhuang University
Nanjing, China
e-mail: wzz5958@126.com

***Abstract*—This paper describes the design and implementation of a mini multi-process micro-kernel embedded Unix-like operating system on ARM platform in technical details, including MMU and memory space mapping, init process, inter-process communication, process management, TTY and tiny shell, multi-level priority-queue schedule, and signaling. The mini OS is developed on Linux platform with GNU tool chain by the author of this paper. The architecture of the mini OS is analogous to that of Minix. Based on it, other operating system components such as file system, network management, and copy-on-write can be appended to form a full-featured embedded operating system. The mini OS can be used for both embedded system application development and related curriculum teaching.**

***Keywords-embedded operating system; multi-process; micro-kernel; inter-process communication; ARM***

## I. INTRODUCTION

With the rapid developments of electronic and computer technologies, embedded systems have already become more and more popular in the wide variety of fields. As the core component of computer system as well as embedded system, operating system has been playing a very important role.

For the purpose of technical research and curriculum teaching, a mini multi-process micro-kernel embedded Unix-like operating system [4] on ARM platform [6] is developed by the author of this paper. The advantage of the system is described in the following.

- Multi-process micro-kernel architecture. Unlike uC/OS [9], which can only create tasks but no process, the mini OS is designed as multi-process micro kernel analogous to the famous Minix [1]. With such kind of architecture, the portability and scalability of the system can be improved significantly therefore is suitable for embedded systems. The mini OS consists of several system-level tasks and various user-level processes, of which system-level tasks perform the core operations all in the kernel address space, while each of user-level processes running in its own independent address space.
- For both embedded development and curriculum teaching. On one hand, the essential techniques related to operating systems [2] and ARM machines [7] are involved, e.g., kernel boot, MMU, exceptions, task creation, process forking, multi-process schedule, inter-process communications, etc. All of these are obviously helpful for development on ARM based embedded systems as well as for students to learn and study. On the other hand, the mini OS is designed more readable, of which the source codes can be provided to students, guiding them to design tiny embedded operating system on ARM platform from scratch.
- Simple and extensible. The mini OS accomplishes the essential part of an embedded operating system, while the amount of source codes is only about 4,100 lines, small enough. Based on it, other components, such as copy-on-write, file system, network management, etc., can be appended to form a full-featured multi-process micro-kernel embedded operating system on ARM platform.

This paper firstly gives a brief overview of this ARM based mini multi-process micro-kernel OS, then describes the key techniques of its design and implementation, and finally gives an example to show the performance of it.

## II. OVERVIEW OF THE MINI MULTI-PROCESS MICRO-KERNEL EMBEDDED OS

### A. Architecture of the Mini OS

The architecture of the mini OS resembles that of Minix. The entire system is divided in to four layers: kernel layer, driver layer, server layer and user layer, as shown in Fig. 1.

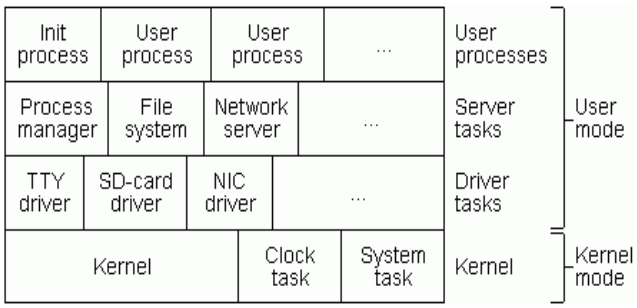


Figure 1. Architecture of the mini micro-kernel embedded OS

The kernel in the bottom layer schedules tasks and processes, and manages the transitions between the ready, running, and blocked states of them. The kernel also handles all messages between tasks and processes with inter-process communication (IPC) routines. In addition to the kernel itself,

Published by Atlantis Press, Paris, France.

this layer contains two modules that function similarly to driver tasks. The clock task is an I/O device driver in the sense that it interacts with the hardware that generates timing signals, but it is not user-accessible like SD-card or NIC drivers – it interfaces only with the kernel and tasks. The system task is to provide a set of privileged kernel calls to the drivers and servers above it. Although the clock task and system task are compiled into the kernel's address space, they are scheduled as separate processes and have their own call stacks.

The three layers above the kernel are all limited to user mode. Each task or process in these layers is scheduled to run by the kernel and none of them can access I/O ports directly. However, the tasks and processes have different privileges. The tasks in layer 2 have the most privilege, those in layer 3 have some privilege, and the processes in layer 4 have no special privilege.

### *B. MMU and Memory Space Mapping*

For default, the process spaces on ARM architecture [5] are designated from the lower end of address 0, one for 32M. While for most NAND FLASH based ARM machine, the lower part of the address space is designated to FLASH memory except the lowest 4K, and the program space is designated from address 0x30000000. Therefore MMU must be used to map the entire address space to accommodate the requirement of the kernel. The address space mapping for the mini OS described in this paper is shown in Fig. 2.

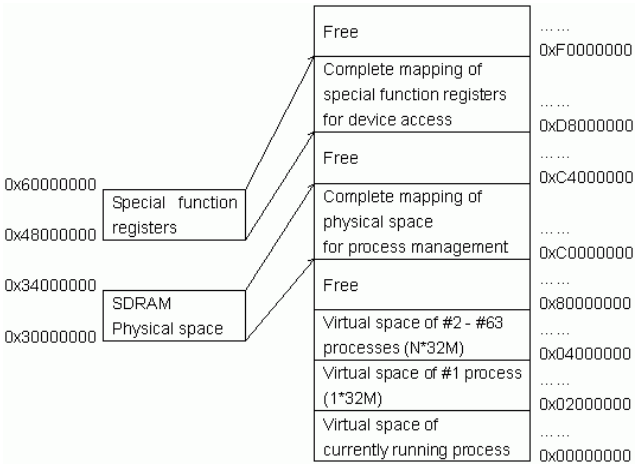


Figure 2. Memory space mapping of the mini OS

After mapping, the physical addresses for program codes are mapped to 0xC0000000 through 0xC4000000, and the physical addresses for special function registers are mapped to 0xD8000000 through 0xF0000000. The lower part of memory spaces is designated to 64 processes, each for 32M, while the physical address space of each is in fact only 1M within 0xC0000000 to 0xC4000000.

### *C. Development Environment*

The mini OS is developed on Linux platform with well-known GNU [10] tool chain in C [8] and ARM assembly [12] programming language.

The advantage of using such kind of development environment is that, firstly, the tools are free and open sources, secondly, Linux platform is just a well-known programming environment never becomes outdated, and finally, convenient for development and research.

Considering the conventions of development and porting, boot loader is not included into the mini OS. In the development environment formed by the author of this paper, the destination host is an embedded development board equipped with a multi-function boot loader.

## III. DRIVER AND SERVER TASKS

For the mini OS described in this paper, five system-level tasks are designed, which are task_clk, task_sys, task_tty, task_pm, and task_fs.

Task_clk performs the operations related to system clock, such as "alarm", or invoke the scheduler periodically.

Task_sys manipulate system calls related directly to the kernel, e.g. signaling functions.

Task_tty is to implement a simple TTY terminal, based on which a tiny shell is designed to show the performance of the TTY, e.g. simple shell command and CTRL-C.

Task_pm plays a very important role in the kernel, which accomplishes the essential functions for multi-process management such as fork(), exit(), and wait(), etc.

Task_fs is to perform file system management. By now, however, only a basic frame is designed to implement system call functions read() and write(). A whole and entire file system will be designed on next stage.

## IV. INIT PROCESS

For a multi-process kernel [3], init process is the foundation. Firstly, it is the first process of the entire process space; secondly, all other processes in the system are spawned by it or its children while it own is created directly by the kernel when booting; thirdly, when any process loses its parent, the init process becomes its new parent. For ARM architecture [11], the default size of the address space designated to each process is 32M, from 0 to 32M – 1. That means the code of a process should be allocated at the bottom space (that is started from address 0). For ordinary ARM development boards, however, the lower address space is designated to SRAM FLASH memory while the code space starts at 0x30000000. Based on such a space it is not convenient to design a lower address space process, e.g. starting from address 0.

To resolve the problem, a dedicated way is designed to create and load a process starting at 0. By this way, two files are designed, i.e. an assembly program and a C program to implement the init process itself. The former in essential is the frame of the init process program which invokes the C program to fulfill the performance of the init process. The code size of the init process program is saved in the beginning of the assembly program which will be used when the init process is loaded into kernel. The latter implements the real function of the init process. Both programs are compiled and linked to form a binary image, which is attached to the tail of the kernel image. With such a way, the binary image of the init process can be loaded into SDRAM

Published by Atlantis Press, Paris, France.

memory together with the real kernel image from FLASH memory during booting. After that, the init process is loaded into the virtual process space with process id 1. Other user-level processes will be forked by this process.

## V. MULTI-LEVEL PRIORITY-QUEUE SCHEDULE

Obviously, schedule is the core role for multi-process operating systems. The schedule routine of the mini OS is designed in such a way that the tasks and processes be scheduled all together. To ensure the priority of all the system-level tasks over user-level processes, multi-level priority-queue schedule strategy [1, 2] is used.

Five priority queues are designed, of which the highest priority queue is for kernel tasks task_clk and task_sys, the secondary for driver tasks, the third for server tasks, the fourth for init process and all other user level processes, and obviously, the lowest is task_idle which is the only user-level task located in kernel space and can only be scheduled when no any task or process is ready. By this way, all the tasks can be scheduled as soon as possible. Only when no any task is running can a user-level process be scheduled. Fig. 3 shows the allocation of the five queues.

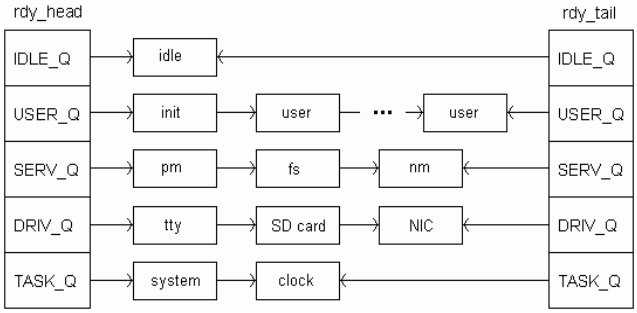


Figure 3. Five queues for multi-level priority-queue schedule

## VI. INTER-PROCESS COMMUNICATION

There are several techniques to accomplish inter-process communications, such as message passing, signaling, and pipes, etc. The mini OS described in this paper implements the first two, i.e. message passing and signaling.

The foundation of the inter-process communication is software interrupt. The SWI (SoftWare Interrupt) on ARM is just such a mechanism. Since the message passing is the core way to accomplish the communications and system calls, there only one type of calls to SWI, i.e. message passing, named as IPC (Inter-Process Communication) by which all other system call functions are implemented, e.g. fork(), read(), write(), and signaling functions.

IPC is the core routine in the mini OS, which manipulates the sending, receiving, notifying of the messages among tasks and processes. The algorithm and the code of IPC are designed analogous to that of Minix with the strategy known as “rendezvous” [1].

In such a way, if the send operation is done before receive, the sending process is blocked until receive happens, at which time the message can be copied directly from the sender to the receiver, with no intermediate buffering. Similarly, if the receive is done first, the receiver is blocked until a send happens.

## VII. PROCESS MANAGEMENT

According to Unix [4] specification, all the processes in an operating system must be forked by another one, as its parent, except for init process just mentioned above. The mini OS described in this paper follows this kind of specification. To realize the creation of a new process based on an existing process, function fork() is designed. The algorithm [1] of the function is shown in Fig. 4.

1. Check to see if PCB table is full;
2. Get a new pid for the child;
3. Copy the parent’s code, data and stack to the child’s memory;
4. Duplicate PCB for child;
5. Set child’s pid in its PCB;
6. Send a message to system task;
7. System task clears some fields of the child’s PCB, such as alarm, signal and signal handlers, etc.;
8. After receiving from system task, the new child is really created;
9. Send a message to child as the receiving message for child process with return value as child’s pid;
10. Return to parent process with return value as 0.

Figure 4. Algorithm of the function fork()

When a process has exited or been killed but whose parent has not done a wait for it, the process enters a kind of suspended animation with the state set as M_HANGING. It is prevented from being scheduled and has its alarm timer turned off, but it is not removed from the PCB table. When the parent finally does the wait, the PCB table slot of the suspended process is freed and the kernel is informed.

## VIII. TTY AND TINY SHELL

A brief TTY terminal is designed for the mini OS, on which a tiny shell is designed to perform several simple internal shell commands such as hello, echo, exit, etc.

The TTY maintains two queues, one for TTY reading and another for writing. The input and output of the TTY terminal are designated to super terminal of the remote host via UART (serial port). To get the key strokes from the super terminal, the receiving of the UART is designed in interrupt mode by which an UART interrupt will occur when a key is pressed in the remote host super terminal. The sending mode of the UART is, however, designed as polling mode. Due to the limit of the paper space, the code of UART interrupt and tiny shell routines is not described in detail further.

## IX. SIGNALING

Besides message passing, which forms the foundation of the inter-process communications (IPC) in this mini OS, another important communication mechanism is signaling which are popularly used in Unix and Linux. Some essential signaling functions are implemented in the mini OS and

Published by Atlantis Press, Paris, France.

some commonly used standard signals are supported, e.g. SIGALRM, SIGHUP and SIGCHLD, etc. Signal registration function, signal(), is also designed which is used to set the handler of a signal. The right time to perform the signal handler is when IRQ or SWI finished. This can be done by inserting a signal process routine at the end of IRQ or SWI.

Firstly, the original return address of the IRQ or SWI is saved onto the stack of user process, and then a manipulate function, named build_sig(), is invoked, which checks the signal setting of the currently running user process and saves the corresponding parameter (signal number) and the signal handler onto the stack of the process.

Secondly, when the IRQ or SWI handler exits, returning to user process space, the system jumps to a wrap function, sig_action(), which will invoke the real signal handler set by build_sig(). After that, all the register values are restored from the stack of user space and the program counter is set to the original return address of the IRQ or SWI.

## X. DEMO OF THE MINI OS

To show the effects of this mini multi-process micro-kernel embedded OS, several internal shell commands for test are designed in the init process, by which some useful system call and signaling functions are tested. Fig. 5 shows the result of command forktest.

```
Multi-process micro-kernel embedded Unix-like operating system
Mikenix V0.01  (C)2009-2013 === Author: Bo Qu,  <http://www.qu99.net>

[root]# forktest
Enter number of processes (1-9): 2

=== Fork test ===
Forking 2 proecesses, and then exit within 5 seconds ...

=== Process[3] created.
        Process[3] will exit in 3 seconds.
=== Process[4] created.
        Process[4] will exit in 3 seconds.
        Process[3] will exit in 2 seconds.
        Process[4] will exit in 2 seconds.
        Process[3] will exit in 1 seconds.
        Process[4] will exit in 1 seconds.
        === Process[3] now exit ===
        === Process[4] now exit ===

Fork test finish ...

[root]#
```

Figure 5. A demonstration of the mini OS

The system call functions used by the command are fork(), wait(), exit(), getpid(), gets(), signal(), alarm(), and pause(), in which alarm() and pause() are invoked by function sleep().

When command is executed, it prompts user to enter the number of sub-processes. After that, some sub-processes are created by fork(), and then sleeping 3 times, once a second. When the three times sleeping finished, each of them exits. The parent process uses function wait() waiting for the exit of each sub-process and then return.

Before exit of the sub-processes, user can press CTRL-C to quit all the sub-processes. In such case, signal SIGINT is captured by TTY and a message is sent to system task, which installs the default signal handler for SIGINT, exit(), as mentioned previously. After IRQ or SWI, this default signal handler will cause all the sub-processes to exit.

## XI. CONCLUSION

The purpose of the mini OS described in this paper is to design a multi-process micro-kernel embedded operating system. Some further improvements will be developed on next stage, including following main components:

- File system. A file system will be added to this mini OS, including the corresponding root file system. The function of loading and running executable files, of course, should be implemented as well since it is the foundation of shell commands.
- Network process routines. A light weight TCP/IP routine is going to be added. By which, some network related functions can be implemented, e.g., for network sniffer, simple Web service, etc.
- Copy-on-write. With the paging mechanism of ARM, copy-on-write way can used to realize a simple virtual memory space which not only can break the limit of 1M byte process space but also reduce the time cost of process creation.

In view of limited space, some technical details, such as UART, IRQ, SWI, process switch, real time clock and time management, etc., are omitted.

As mentioned above, the mini OS can be not only suitable for practical embedded systems development but also beneficial to the related curriculum teaching for under graduate computer majors. It can be improved further to form a full-featured multi-process micro-kernel embedded operating system. To research on it is of great value.

## REFERENCES


[1] A. S. Tanenbaum and A. S. Wookhull, Operating Systems: Design and Implementation, 3E, Prentice Hall, Inc., 2008

[2] A. Silberschatz and P. B. Galvin, Operating System Concepts (6th Edition), John Wiley & Sons, Inc., 2002.

[3] L. F. Bic and A. C. Shaw, Operating System Principles, Prentice Hall. Inc., 2003.

[4] M. J. Bash: The Design of the UNIX Operating System, Prentice Hall, Inc. (2006)

[5] A. N. Sloss, D. Symes and C. Wright, ARM System Developer's Guide: Designing and Optimizing System Software, Elsevier Inc, 2004

[6] T. Noergaard, Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers, Elsevier Inc, 2005

[7] W. Wolf, Computers as Components: Principles of Embedded Computing System Design, Morgan Kaufmann pub, 2005

[8] M. Barr and A. Massa, Programming Embedded Systems, Second Edition, O'Reilly Media, Inc., 2006

[9] J. J. Labrosse, Micro C/OS-II the Real-Rime Kernel, 2e, CMP Media LLC, 2002

[10] Stallman R M, Using the GNU Compiler Collection, 2002 (http://www.gnuarm.com/pdf/gcc.pdf)

[11] ARM limited, ARM Architecture Reference Manual, 2005

[12] ARM limited, ARM Developer Suite Assembler Guide, 2001


Published by Atlantis Press, Paris, France.