RTOS Independent Study - Spring 2019

John Tan
tan.jo@husky.neu.edu

Submit date: 4/26/2019
Due Date: 4/26/2019

Instructor: Gunar Schirner
T.A: Bruno Morais

Abstract

This report documents the implementation of a real-time embedded microkernel implemented in C++03 for the ARMv7 CPU core on the Xilinx ZedBoard platform. The design of the operating system is based on the kernel specification from the University of Waterloo's Real-Time Programming class (CS452). The kernel implements task scheduling, message passing, event handling, and clock services.

Introduction

Operating systems provide the foundation for modern software. Nearly all software relies on the resource multiplexing services provided by an OS. More and more, even simple embedded software runs on a real time operating system or even a Linux distribution.

The motivation behind this independent study is to uncover, in as much detail as time allows, how the components of an operating system are implemented and how they work together to provide a platform on top of which user applications can be implemented. In particular, this project explores the incremental implementation of the real-time microkernel on an ARMv7 CPU core as specified by Professor Bill Cowan at the University of Waterloo in his Real Time Programming class [0].

This report serves as the documentation for the real time operating system implemented for this independent study. The repository of the operating system detailed in this report can be found at the following link:

https://github.com/neu-ece-esl/rtos_study

Project Setup

Language

Though many well-known, open source operating systems are implemented in pure C, this OS kernel was written in C++03. Object-oriented programming has proven to simplify the development process by allowing developers to repurpose and reuse large chunks of code in different contexts. While this can be accomplished in pure C, it requires the developer to implement object orientation in the language themselves. As a result, a language with native object-oriented functionality (such as C++03) is a natural choice.

Other than object-oriented features, no other C++ functionality or standard libraries were used. In this respect, one could say that the language used was “C with classes.”

Hardware

This OS was developed for the Xilinx ZedBoard platform. The ZedBoard has a dual Cortex-A9 processor core, an FPGA, and a number of hardware peripherals.

External Code

The Xilinx Board Support Package (BSP) for the ZedBoard was used minimally. In particular, the BSP drivers for the CPU private timer and interrupt controller were used.

Development/Testing Environment

The Xilinx Software Development Kit (SDK) was used to develop this OS [1]. The Xilinx SDK provides a relatively painless method of building an application project with its

corresponding BSP, as well as a debugger interface that can attach to both an emulator and real hardware. For development, the Xilinx distribution of QEMU with ZedBoard support was used, and the OS was later tested and debugged on a real ZedBoard.

Design and Implementation

This operating system kernel provides four main services: task management, inter-process communication, events, and clock services. Figure 1 below shows an overview of all the operating system services, and the task state transitions that result from each of the service calls.

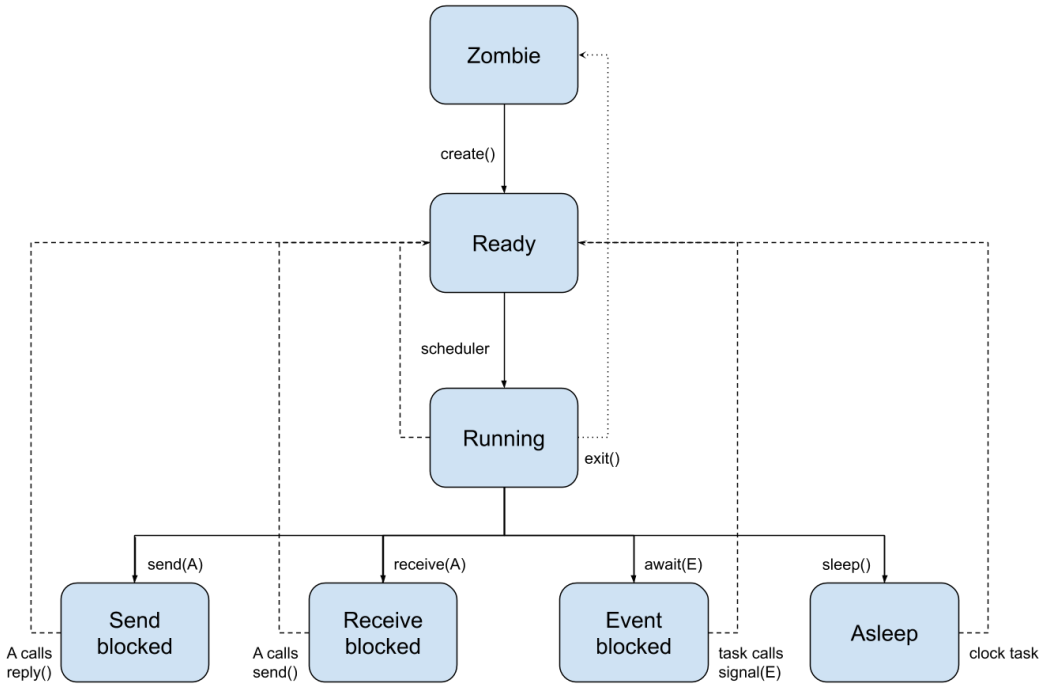

Figure 1 - Task States

Tasks begin in the ZOMBIE state, and are set to READY after being created. The scheduler determines if the task runs (and is set to the RUNNING state). While running, the scheduler can interrupt and set the task to READY, or the task itself can call one of the operating system services shown to transition to a blocked or asleep state. Once blocked or asleep, another task must call a corresponding operating system service to unblock a blocked task.

Task Management

Task Control Block

The OS has a task control block (TCB) that stores the information of each task. Each TCB entry corresponds to a single task. A TCB entry contains the following information (see `task.h` for the full definition):

- The pointer to the current top of the stack
- The task ID
- The name of the task (currently unused)
- The priority of the task
- The state of the task
- Pointers for message passing
- Receive queue, which stores all the tasks that are sending to this task

When a task is created, the OS allocates a TCB entry to the task. In doing so, it allocates a stack, a task ID, and a receive queue to the task, and sets its state as ready to run. It also sets up the initial context stored on the stack so that when a context switch occurs, it jumps to execute the function specified as a parameter to `OS.create()`.

Scheduling

The operating system implements a priority scheduler with round-robin between tasks of the same priority. The priority queue is implemented as a minimum array heap [2], meaning that priorities of a lower value run before priorities of a higher value. This data structure was chosen for its reusability in other parts of the OS, as well as its run time complexity, which is O(log n) for push and pop operations.

Tasks can cooperatively yield execution time by directly calling `OS.yield()`, or can be pre-empted and forced to yield on a timer interrupt. In addition, many of the operating system services are blocking and as a result will force the calling task to yield the CPU and, as a result, schedule a different task to run.

Inter-Process Communication

The operating system implements synchronous message passing as its only form of inter-process communication. The operating system exposes a send/receive/reply API that allows tasks to communicate with one another. When messages are exchanged, the message is copied over from one task's memory to the other. This means that inter-process communication is carried out without sharing memory between concurrently running tasks.

Because communication is synchronous, a sending task always blocks until it receives a reply. Similarly, a receiving task may block if its receive queue is empty. Figures 2 and 3 below demonstrate the blocking behavior that occurs during a typical message exchange.

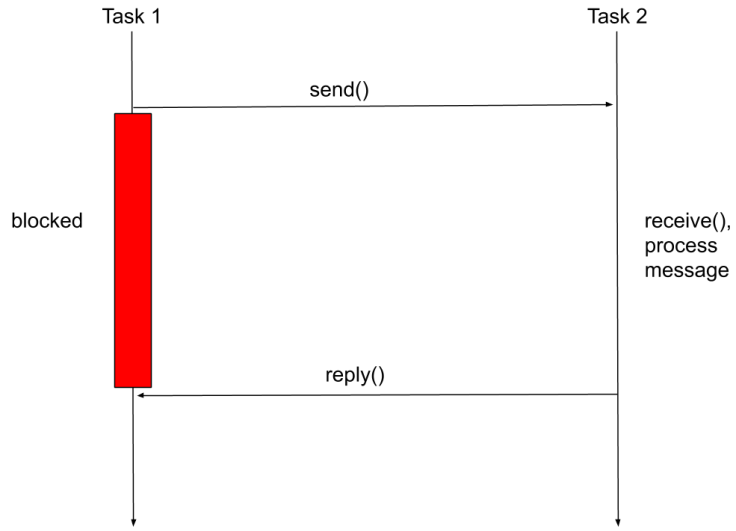

Figure 2 - Send before receive

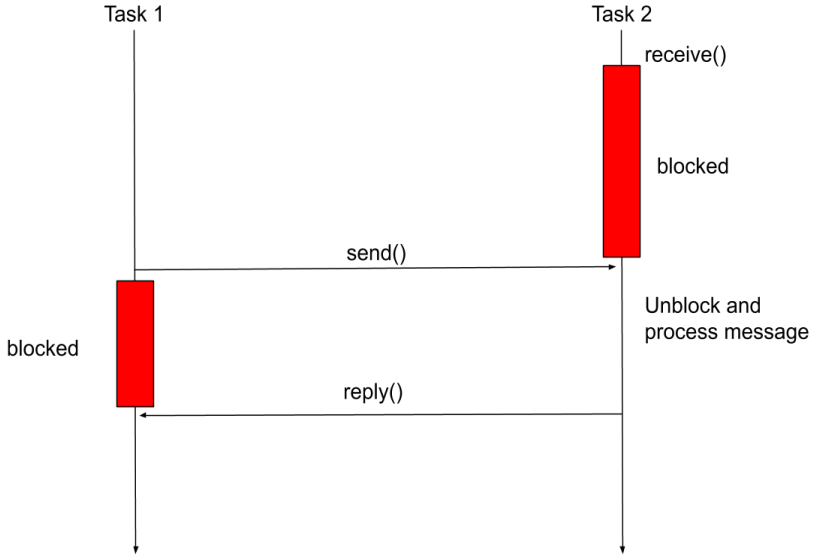

Figure 3 - Receive on empty queue

A major advantage of synchronous message passing is that it avoids sharing memory between tasks. This prevents a class of bugs from occurring as a result of shared memory, such as race conditions. However, synchronous message passing has the drawback of requiring greater overhead in comparison to shared memory, as messages are constantly copied back and forth between tasks. In addition, each message exchange requires a few context switches and may not be as performant as shared memory due to the amount of time each task spends blocked.

One other drawback to note of this message passing implementation is that tasks may block forever if the user is not careful. If, for example, Task A sends to Task B, and Task B also sends to Task A, and neither task ever calls receive, then both Task A and B will be blocked forever with no chance of running again. A simple server/client design pattern generally sidesteps this problem, and is detailed in the "Suggestions for Users" section later on.

Events

The OS exposes a simple event service that allows a task to wait on some user-defined event. When a task waits, it is marked as EVENT_BLOCKED and does not run until some other task or an interrupt handler signals that the event occurred. Upon signalling the event, all tasks waiting on an event are marked as ready to run and put back on the ready queue. Figure 4 below visualizes how tasks typically use events.

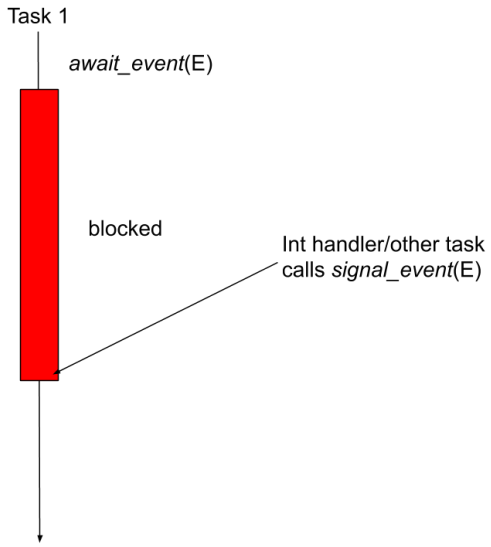

Figure 4 - Waiting on events

This is a low-overhead method of synchronization that is preferable for use in interrupt handlers instead of send/receive/reply, since interrupt handlers should be as short as possible and certainly not block before returning.

Clock Services

The operating system keeps track of time since system startup. The resolution of the system time is 3 ms, which is the interval at which the timer interrupt fires. The main clock services provided are `OS.get_time()` and `OS.sleep()`. `OS.get_time()` simply returns the time since system startup. `OS.sleep()` puts the calling task to sleep until the specified wakeup time. When a task is put to sleep, its state is changed to ASLEEP and is pushed onto a priority queue that is prioritized by descending wakeup time.

Clock services are implemented through the use of two tasks owned by the operating system: the system task, and the clock server. The system task continually waits on the timer interrupt to fire (which is defined as an event), and sends a message to the clock server indicating that a clock tick occurred. The clock server keeps track of system time, continually receives messages (and blocks if it has no senders waiting), and processes clock tick and get time messages. On each clock tick, the clock server increments the tick count and wakes up any sleeping tasks with a wakeup time less than the current system time.

Figure 5 below demonstrates how a typical `OS.sleep()` call is handled by the system task and clock server.

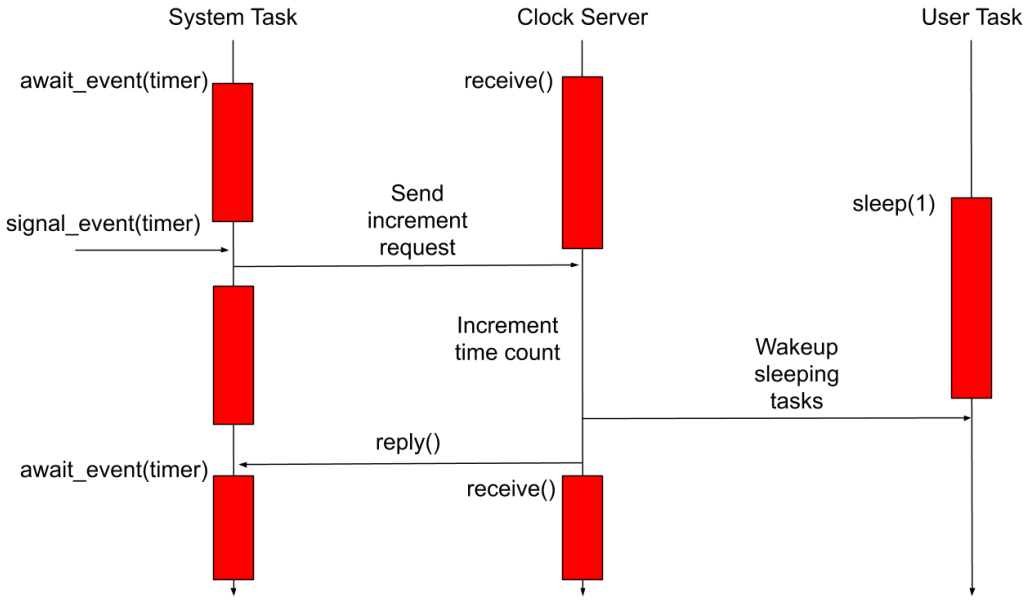

Figure 5 - How `OS.sleep()` is handled

System Initialization

On initialization, the OS first sets up the timer interrupt and interrupt controller, and zeros out all of the TCB entries. It then creates the system task at the highest priority, then the first user task at the specified priority. The system task is the first task that runs, which creates and brings up the clock server at a lower priority. The clock server then runs and attempts to receive a message, but is subsequently blocked because no task has sent to it yet. Finally, the first user task is then able to run.

API Documentation

The following section details the exposed kernel interface. Note that the operating system was implemented as a single class named `myOS`. The kernel API is exposed as public member functions of this class and is implemented in the `os.cc` and `os.h` files in the `kernel` directory. The operating system class is instantiated as a global variable - if the user wants to use operating system services, they must include the `os.h` file and add the line:

```
extern myOS OS;
```

to the top of the file.

Task API

Create

```
int OS.create(int (*task)(void*), void *task_param, int priority)
```

Params:

- task: function pointer to the task that will be created
- task_param: parameters to be passed to the task
- priority: the priority at which the task will run. A lower value means a higher priority (e.g., task with priority 2 will run before a task with priority 3).

Returns the Task ID of the newly allocated task if success, -1 if failure. Create can fail if all of the task spaces have been allocated. See "System Configuration" section for more detail.

Creates a new task. This function allocates a stack for the task, sets up its initial context, and marks the task as READY.

Exit

```
void OS.exit(void)
```

Destroys a running task. The task is removed from the ready queue and has its state set to ZOMBIE, indicating that it is no longer running and can be allocated to a new task.

Scheduling

Yield

```
void OS.yield(void)
```

The task yields the CPU and is put back on the ready queue. The next task on the ready queue is then marked as RUNNING and uses the CPU. This is considered a critical section and interrupts are disabled during this function as a result.

Yield from interrupt

```
void OS.yield_from_interrupt(void)
```

Same as OS.yield(), except it resets the interrupt stack to before the interrupt was called. Only used internally by the OS.

Message Passing

Send

```
int OS.send(int tid, void *msg, int msglen, void *reply, int rplen)
```

Params:

- tid: The ID of the task to send the message to
- msg: The message to be sent
- msglen: Length of the message, in bytes
- reply: A pointer to the buffer that the receiving task will send its reply to
- rplen: Size of the reply buffer

Returns the size of the reply on success, -1 on failure.

Sends a message to the task specified by the tid parameter and blocks until the receiving task sends a reply. The message will be copied to a buffer supplied by the receiving task.

Receive

```
int OS.receive(int *tid, void *msg, int msglen)
```

Params:

- tid: An output parameter that will hold the ID of the task that sent the message
- msg: A pointer to a buffer to receive the message
- msglen: Size of the buffer, in bytes

Returns the size of the message if the receive buffer was large enough, or -1 if the message was truncated.

Receives a message and the corresponding task ID. The message from the sending task is copied to the buffer supplied to OS.receive(). If the send queue is empty for this task, then this function call blocks until another task sends to this task.

Reply

```
int OS.receive(int tid, void *reply, int rplen)
```

Params:

- tid: The task to send the reply to
- reply: Pointer to the reply message
- rplen: The size of the reply message

Returns 0 on success, -1 if the reply was truncated, -2 if the task id was invalid, or -3 if the task being replied to is not in the SEND_BLOCKED state.

Replies to a sending task. The reply message is copied to the reply buffer supplied by the sending task when it calls OS.send(). Will fail if the task id does not correspond to a SEND_BLOCKED task.

Events

Await event

```
int OS.await_event(int event)
```

Params:

- event: The event the task will wait on

Returns the result of the event, which is provided by the caller of OS.signal_event().

Marks the current task as EVENT_BLOCKED until another task or interrupt handler calls OS.signal_event().

Signal event

```
int OS.signal_event(int event, int rv)
```

Params:

- event: The event that has been completed
- rv: The return value of the event

Returns 0 on success, -1 if the event was not defined in the configuration file.

Unblocks all tasks that are EVENT_BLOCKED waiting on the specified event.

Clock Services

Get time

```
uint64_t OS.get_time(void)
```

Returns the number of timer ticks since startup.

Internally calls OS.send() to the clock task and is therefore blocking.

Sleep

```
void OS.sleep(uint64_t time)
```

Params:

- time: Number of ticks the calling task should be asleep for

Marks the calling task as ASLEEP until the number of specified ticks has passed.

Misc. OS Services

Panic

```
void OS.panic(char *msg)
```

Params:

- msg: Message to be displayed if something goes wrong

```
Prints the message to terminal and loops forever. Used to catch
unexpected/strange situations in the kernel.
```

System Initialization

Init

```
void OS.init(int (*first_task)(void*), void (*schedule_callback)(void*),
int first_task_prio)
```

```
Params:
```

- `first_task: The first user task to run`
- `schedule_callback: Callback for timer interrupt`
- `first_task_prio: Priority assigned to the first user task.`

```
Initializes the kernel. Sets up the interrupt controller and timer,
initializes the TCB, and creates the system task and the first user task.
```

OS Configuration

The operating system is configured by the parameters found in `os_config.h` in the `kernel` directory. In this file, the user can configure the maximum number of tasks the OS can handle, the size of the stack allocated to each stack, the maximum and minimum priorities, and the events that are defined.

Suggestions for Users

Servers/Clients

As mentioned previously, one of the drawbacks of the send/receive/reply message passing paradigm is that the user must take extra care in avoiding tasks blocking forever with no chance of running again. To prevent this, a design pattern for tasks that is encouraged with use in this operating system is the server/client model. Servers are responsible for receiving/replying, and clients are responsible for sending only. Thus, if clients only send to servers, then users can entirely avoid the problem where two tasks are blocked because they are both trying to send to each other. The psuedocode for both clients and servers are shown below in Figure 6.

Example Server

```
while (1)
    // possibly blocking
    request, sender = receive()
    switch(request):
        …
    reply(sender, reply_msg)
```

Example Client

```
while (1)
    // blocking
    reply_msg = send(A, request_a)
    switch(reply_msg):
        …
    reply_msg = send(B, request_b)
    ...
```

Figure 6 - Example Server/Client psuedocode

Conclusion

This project has proven to be incredibly educational in learning about the internal mechanisms of an operating system. In completing this project, a student can learn a great deal about how operating systems manage and schedule tasks, allow tasks to communicate and synchronize with one another, and how an operating system brings up a computer from startup. In addition, this project deepens a student's understanding of assembly programming (in this particular case, ARMv7 assembly), and C/C++ calling conventions. Finally, this project also teaches students to become more systematic, disciplined, and patient debuggers as many obscure, seemingly nonsensical bugs tend to sprout especially in the early stages of operating system development.

Further Work

This operating system is by no means complete - any number of extensions can be thought of and added on. Some ideas for further work include:

- Memory protection between tasks
- Non-blocking/asynchronous message passing
- Shared memory synchronization primitives
- Hardware sharing synchronization
- Support for hard real-time scheduling
- Drivers for more hardware support
- Extensions to run on more than a single CPU core
- Clearer separation between kernel and user space

References

[0] http://www.cgl.uwaterloo.ca/wmcowan/teaching/cs452/w18/index.html

[1] https://www.xilinx.com/products/design-tools/embedded-software/sdk.html

[2] https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html