Notes and Background on Operating System for the 68000 CPU

Brent Seidel
Phoenix, AZ

March 11, 2024

This document is ©2024 Brent Seidel. All rights reserved.

Note that this is a draft version and not the final version for publication.

Contents

Chapter 1

Overview

This is a collection of notes on a simple multi-tasking operating system for the 68000 CPU, temporarily names os68k. The main goals are:

1. To be able to blink lights in interesting pattern in the Pi-Mainframe (`https://github.com/BrentSeidel/Pi-Mainframe`) project,
2. To learn something about operating system design, and
3. To actually have a somewhat useful operating system.

Note that all statements are subject to change as the project develops.

This is based on the 68000 simulator part of the Sim-CPU (`https://github.com/BrentSeidel/Sim-CPU`) project. It is targeted towards a system with 16Mbytes of memory and no MMU. The memory is divided into 16 one megabyte sections, the first (lowest) section is for the operating system and each task gets one section. The gives a system that supports up to 15 user tasks plus the operating system.

Simulated devices provided by Sim-CPU that are currently used are the clock to provide a periodic interrupt for tasking and the serial-telnet port for console I/O. Simulated disks are not yet supported.

The operating system also includes a library with utility routines that can be used by the user programs.

1.1 Kernel

The kernel is composed of several sections. The first two are at locations that are fixed by the simulated CPU and hardware:

1. The CPU vector table starts at address 0 and contains 256 long word entries. This occupies 1 kilobyte of space.
2. The I/O port section runs from the end of the vector table at an address of 400_{16} and runs to 1000_{16}.

The remaining sections are arbitrarily arranged and will probably change with development.

1. HW_SECT contains code for interfacing with the hardware devices and related interrupt service routines.
2. OS_SECT contains the main operating system code including initialization, the clock interrupt, context save/restore, scheduling, exceptions, and system calls.
3. OS_DATA contains the operating system data tables and messages. The data tables currently include the task control blocks and the console device blocks.
4. LIB_SECT contains library routines for use by the operating system and user programs. Note that routines that use system calls shouldn't be used by the operating system, at least not yet.
5. LIB_DATA contains library data. Note that since library routines may be in use by multiple tasks simultaneously, this should be constant data only. Any variable data should be allocated on the stack.
6. OS_STACK is space for the operating system stack.
7. USR_STACK is space for the operating system user stack. The is used by the null task that runs when no other task can be run. Its stack needs are minimal.

1.1.1 Hardware Abstraction Layer

The currently supported hardware includes a clock that provides periodic interrupts and a terminal interface that can be accessed externally by telnet (or gtelnet).

Clock

The clock has a settable rate for the periodic interrupts. The rate is a byte size port which allows any value 0-255. The multiplier in the simulation is 100mS, thus to get a 1 second interrupt, the rate would be set to 10. Setting the rate to one gives a 100mS interrupt (or 10 times a second). This is a reasonable rate for multitasking. As some point, the simulation may get adjusted to allow higher rates. There is a tradeoff as higher interrupt rates give more overhead. This will require some experimentation.

Terminal Interfaces

Multiple terminal interfaces can be supported. Right now, each interface has its own interrupt vector, but it should be possible to have a single vector where the service routine determines which interfaces are ready.

A terminal multiplexer is in work, but not fully implemented or debugged yet.

1.1.2 Scheduling

Currently, a simple round-robin scheduler is used.

1.1.3 System Calls

Currently, all system calls are handled by `TRAP #0`. The system call number and any parameters are pushed onto the stack prior to the call. The following system calls are currently defined:

0 SYS_EXIT - Exit program

1 SYS_PUTS - Send a string to the console

2 SYS_GETC - Get a character from the console

3 SYS_PUTC - Send a character to the console

16 SYS_SLEEP - Suspend current task for a number of clock ticks

64 SYS_SHUTDOWN - Shuts the system down

1.2 Library

The library starts with a table of addresses of the various library routines. This allows the user programs to find the desired routine by looking in a fixed address. The code for a library call looks something like (Other registers besides `%A0` can be used, but `%A7` is the stack pointer and `%A6` is often used as a frame pointer. The macros use `%A0`):

```
...  Put stuff on the stack
    move.l #LIBTBL,%A0
    move.l LIB_GETSTR(%A0),%A0
    jsr (%A0)
...  Cleanup the stack
```

Since the library routines can be preempted, the library must contain only reentrant code.

1.3 User Space

Each task is allocated 1 megabyte of space starting on a megabyte boundary. The initial PC is the start of the space and the initial SP is the end of the space. The user program can use this space as it sees fit.

Chapter 2

Kernel

This chapter describes the operating system kernel in more detail.

2.1 OS Macros and Definitions

A number of macros and data structures are defined in order to help keep the kernel code consistent and easier to understand.

2.1.1 Definitions

The data structures are defined using macros to help avoid repeating code and so that changes need only to be made in the macro definition. In addition to the macro defining the data structure, symbols are defined for various offsets and data within the structure.

The core of the multi-tasking is the task table. It is basically an array of longword addresses to task control blocks (TCBs). The table is located by global symbol `TASKTBL`. The task number of the currently executing task is stored in a word located at global symbol `CURRTASK`. It is initialized to 1. The maximum number of tasks defined is another global symbol `MAXTASK`, which is used to identify the end of the task table.

Task Control Block (TCB)

Currently the TCB is fairly basic. It will need to be expanded if FPU or MMU support is to be added.

Each task has a TCB with the following structure:

```
.macro TCB entry,stack,console
    .dc.w 0                 | PSW (0)
    .dc.l \entry            | PC  (2)
    .dc.l 0                 | D0  (6)
    .dc.l 0                 | D1 (10)
```

```
    .dc.l 0             | D2 (14)
    .dc.l 0             | D3 (18)
    .dc.l 0             | D4 (22)
    .dc.l 0             | D5 (26)
    .dc.l 0             | D6 (30)
    .dc.l 0             | D7 (34)
    .dc.l 0             | A0 (38)
    .dc.l 0             | A1 (42)
    .dc.l 0             | A2 (46)
    .dc.l 0             | A3 (50)
    .dc.l 0             | A4 (54)
    .dc.l 0             | A5 (58)
    .dc.l 0             | A6 (62)
    .dc.l \stack        | SP (66)
    .dc.l 0             | Task status (70)
    .dc.l 0             | Sleep timer (74)
    .dc.l \console      | Console device (78)
.endm
```

The offsets are defined as follows:

```
    .equ TCB_PSW,     0
    .equ TCB_PC,      2
    .equ TCB_D0,      6
    .equ TCB_D1,     10
    .equ TCB_D2,     14
    .equ TCB_D3,     18
    .equ TCB_D4,     22
    .equ TCB_D5,     26
    .equ TCB_D6,     30
    .equ TCB_D7,     34
    .equ TCB_A0,     38
    .equ TCB_A1,     42
    .equ TCB_A2,     46
    .equ TCB_A3,     50
    .equ TCB_A4,     54
    .equ TCB_A5,     58
    .equ TCB_A6,     62
    .equ TCB_SP,     66
    .equ TCB_STAT0,  70
    .equ TCB_STAT1,  71
    .equ TCB_STAT2,  72
    .equ TCB_STAT3,  73
    .equ TCB_SLEEP,  74
    .equ TCB_CON,    78
```

The current TCB status flags are defined for `TCB_STAT0`. The rest of the bits are unused. The only flags defined now are for types of waiting. This allows the

scheduler to do a simple test for zero on the `TCB_STAT` longword. If flags ever need to be defined for non-wait purposes, the 32 bit status word may be split into two 16 bit words, with one word describing waits and the other describing non-wait status. The currently defined flags are:

```
.equ TCB_FLG_IO,      0
.equ TCB_FLG_SLEEP,   1
.equ TCB_FLG_EXIT,    2
```

`TCB_FLG_EXIT` is used to indicated that a task has terminated and should no longer be scheduled. Since a shared command line interpreter is available, this flag may get depricated. At some point, `TCB_FLG_IO` may be split as more types of I/O get defined.

2.1.2 Macros

The `GET_TCB` macro is defined as follows, and is used whenever a pointer to the current task's TCB is needed:

```
.macro GET_TCB reg
    move.l #0,\reg              |  Ensure that high bits are cleared
    move.w CURRTASK,\reg        |  Get current task number
    add.l \reg,\reg
    add.l \reg,\reg             |  Multiply by 4
    move.l TASKTBL(\reg),\reg   |  Index into TCB table
.endm
```

2.2 Scheduler

Scheduling tasks consists of three basic functions. The first is saving the context of the current task, then determine which task to run next, and finally restoring context for the new task. Should the TCB be expanded for FPU and/or MMU support, the save and restore context routines will also need to be updated.

2.2.1 Saving Context

The context is saved by the following routine. It saves register `%A6` and uses it as a pointer to the current task's TCB. Some values are read off the stack and written into the TCB. Most of the registers are saved using a `movem.l` instruction

```
CTXSAVE:
    .global CTXSAVE
    move.l %A6,-(%SP)      |  A6 is used to get a pointer to the task data area
    |
    |  At this point, the stack is:
    |   0(SP) -> A6
```

```
|    4(SP) -> PSW
|    6(SP) -> PC for return
|
GET_TCB %A6
move.w 8(%SP),(%A6)              |  Save PSW
move.l 10(%SP),TCB_PC(%A6)       |  Save PC
movem.l %D0-%D7/%A0-%A5,TCB_D0(%A6) |  Most registers are now saved
move.l %USP,%A0
move.l (%SP)+,TCB_A6(%A6)        |  Save A6
move.l %A0,TCB_SP(%A6)           |  Save USP
move.l TCB_A0(%A6),%A0           |  Restore A0
move.l TCB_A6(%A6),%A6           |  Restore A6
rts
```

2.2.2 Schedule

The scheduler determines which task is to be run next. It first checks if the current task is zero. If so, the current task is set to one. Then the task list is searched, starting with the current task and wrapping around at the end, to find the next task that is ready to run. If there is no task that is ready to run, task zero is selected. Effectively, task zero runs at a lower priority than the other tasks. It only runs if all other tasks are waiting. After the scheduler runs, it falls through to context restoration and the selected task is run.

```
SCHEDULE:
    .global SCHEDULE
    clr.l %D0
    clr.l %D1
    move.w CURRTASK,%D0          |  Current task number
    tst.w %D0                    |  Check if the current task is the null task
    bne 0f
    moveq.l #1,%D0               |  If current task is 0, set it to 1
    move.w %D0,CURRTASK          |  Update memory as it is used later
0:
    move.w #MAXTASK,%D1
    subq.w #1,%D1                |  Max task number
    cmp.w %D0,%D1                |  Check if current task is max task
    bne 1f                       |  If not, start loop to find next task
    moveq.l #1,%D0               |  Otherwise, wrap to start of list
    bra 4f
1:                               |  Loop to scan through task table
    addq.l #1,%D0
4:
    move.l %D0,%A0
    add.l %A0,%A0
    add.l %A0,%A0                |  Multiply by four to use as index
```

```
    move.l TASKTBL(%A0),%A0 |  Address of task block
    beq 2f                  |  If no task block, end of table is reached.
                            |  Just use task 0.
    tst.l TCB_STAT0(%A0)
    beq 3f                  |  Found a task to select
    cmp.w CURRTASK,%D0
    beq 2f                  |  No candidate was found
    cmp.w %D0,%D1           |  Check for end of list
    bne 1b                  |  Loop, if not
    moveq.l #1,%D0          |  Go back to check task 1, if so
    bra 4b
2:
    clr.l %D0               |  If no task found, use task 0
3:
    move.w %D0,CURRTASK     |  Set the new current task and ...
```

2.2.3 Restore Context

Once a task is scheduled, the last step before running the task is restoring its context. This loads the registers from the context save area in the task's TCB and sets up the stack for a `rte` instruction.

```
CTXREST:
    GET_TCB %A6
    move.l TCB_SP(%A6),%A0     |  Get the user stack pointer
    move.l %A0,%USP            |  Set user stack pointer
    move.w (%A6),(%SP)         |  Put PSW on stack
    move.l TCB_PC(%A6),2(%SP)  |  Put PC on stack
    movem.l TCB_D0(%A6),%D0–%D7/%A0–%A6
    rte                        |  Carry on
```

Chapter 3

Library

The library contains a number of routines that can be used by tasks. Some of these routines may also be used by the operating system.

Chapter 4

Command Line Interpreter (CLI)

The Command Line Interpreter is shared code that can by used by any task.