A BEAGLEBONE BLACK OPERATING SYSTEM

DESIGN DOCUMENT

April 8, 2019

Group A:

Lizzie Adams
Erik Andvaag
Rowan MacLachlan
Braunson Mazoka
Jian Su

CONTENTS

1 BOOTLOADER

Members Responsible: Braunson Mazoka & Lizzie Adams The bootloader is designed to be completely independent from the operating that it executes. We opted to use the existing U-boot implementation to load our bootloader into memory as a 4th stage to the boot process.

The BeagleBone Black has the following boot process:

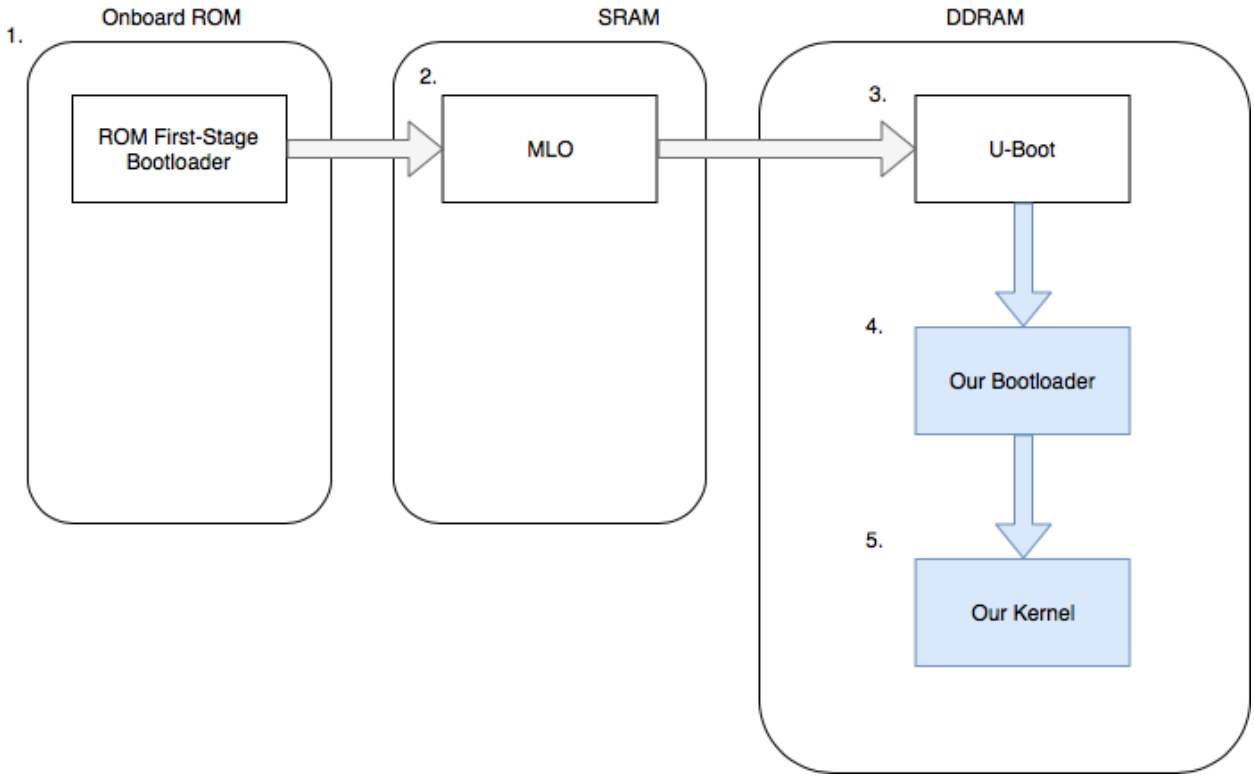

Figure 1: Depiction of the boot sequence flow, where blue indicates stages we have implemented.

We decided not to implement our own MLO because it would have been more time consuming than it would have been worth. It would have involved very specific hardware initialization that likely could not be applied to different scenarios. We felt that there was more opportunity to learn about other aspects of OS design.

The u-boot image is tightly coupled to the MLO that is generated by U-boot. Thus, we could not replace it. However, our bootloader serves a similar purpose to the u-boot.img. In order for the U-boot image to custom load our bootloader, we recompiled it by adjusting its `BOOTCOMMAND`. Details for compiling this can be found in the bootloader's README file.

We reverse engineered the flashing script that the BeagleBone Black uses to create its SD card images. This helped us understand how to organize our SD Card for booting. We decided to hide the bootloader from the main partition by loading it onto the card in RAW mode before the FAT file system begins. By doing this we keep the bootloader away from the user to prevent them from worrying about it or deleting/modifying it. Our File System of choice is FAT32 at an offset of 4mb.

FAT was chosen because of its simplicity to implement.

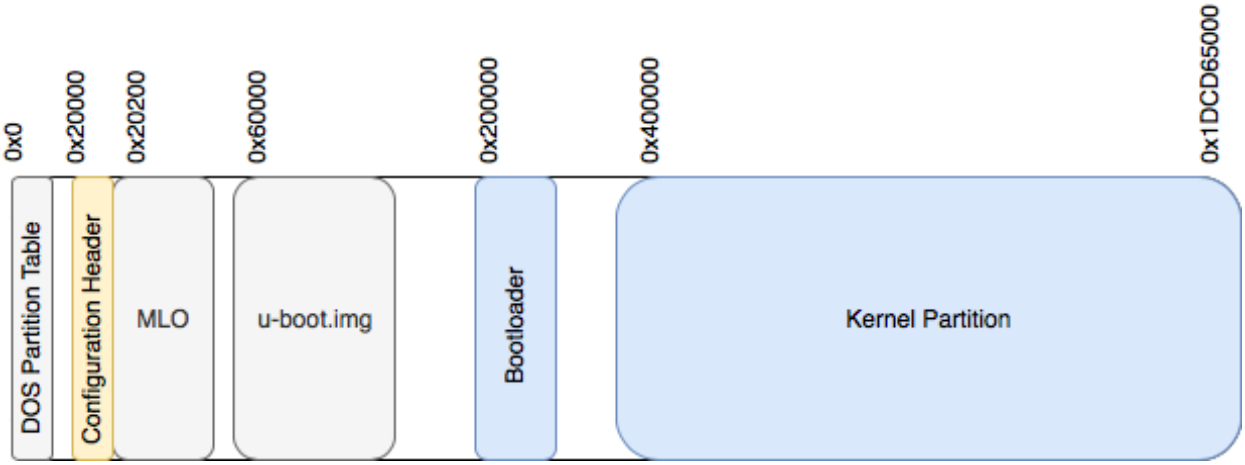

Figure 2: SD Card Layout

The kernel partition contains the `kernel7.img` which is loaded in by our MMC driver on boot. The kernel is loaded into `KERNBASE` (see mem.h) and the kernel can be a maximum size of 33.554432MB as denoted by `MAX_KERN_SIZE`. This could be increased by loading the bootloader into RAM at a different offset. By default our custom u-boot config loads it to `0x82000000`, thus when loading the kernel into RAM we have to make sure not to overwrite our own bootloader.

Another important feature of the initialization phase is clearing the BSS memory to zeros for static variables that were left uninitialized. The bootloader also installs basic exception handlers for interrupts on tick counts for the LED to flash as a pulse of life. The last driver used by the bootloader is UART which is used to communicate to the user over serial what the current status of the read operation is.

The last act of the bootloader is to switch the program counter to the start of the newly loaded kernel memory. This means that kernel developers must create a linking script as the first bits of the image must be the text section with no header.

In the future it would be worth allowing the bootloader to support headers on the kernel images to allow kernel developers to specify where in RAM they wish to be loaded into. This is common practice on linux as well as u-boot. Another worthy expansion would be to replace u-boot all together. This would involve writing the low level initialization ourselves and installing our bootloader in the location of the MLO with the configuration header that the BeagleBone Black expects. The details for hardware initialization are detailed in the Technical Reference manual. Lastly, see the section on the file system driver for notes on how performance could be improved when reading in the kernel image.

Some of the items we particularly struggled with were in finding where to place our code to be loaded in by the bootloader. This was generally due to a poor initial understanding of how the SD card was laid out. It was helpful to create a hexdump of the SD card and read the TRM on the boot process. Another learning point was understanding how to move around the text section to the start of the executable. From there it was pretty clear how we need to organize our SD card.

2 MEMORY

Members Responsible: Braunson Mazoka & Lizzie Adams The memory design of our operating system allows for the use of Virtual Addressing known as *Sections*. These *Sections* are 1MB pages that require only a single level page table of 4096 entries to store the entire address range. Our investigation into Linux and it's ARMv7 implementation revealed that, instead of using ARM's built in protection bits, each process rather has its own page table that is swapped on every context switch. We chose to employ this method, resulting in an additional 16KB of storage overhead added to each process PCB. The translation table format used was for that of large 1MB sections of memory, as shown in 3. The MMU requires that the translation table have very specific entry formats for certain memory regions. It turns out that that even a small issue with the translation table results in the MMU hanging infinitely upon initialization, something we found to difficult to debug. We eventually investigated starterware code as to learn the configuration that the system is expecting so that we could successfully initialize the MMU.

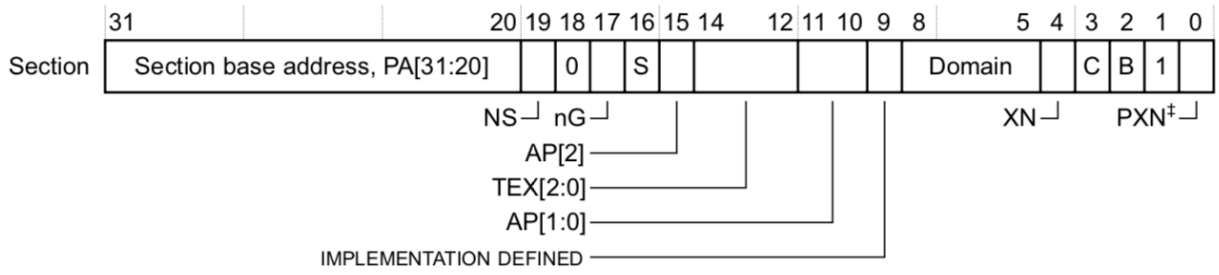

Figure 3: ARMv7 Section Format

The file `mem.S` demonstrates the translation table initialization procedure used for each memory region. The translation table entries corresponding to the DDR memory region specify shareable and cacheable memory with a write-back write-allocate inner cache policy and a write-through no write-allocate outer cache policy. The same configuration was used for the single 1MB section of OCMC region. Finally, the entries used for the device region specify shareable device memory. When initializing the kernel, all addresses corresponding to all regions must be correctly mapped by the translation table. In contrast, the memory available to each use process is specified by allocating a new page table that only maps in sections the process is to be permitted to access, while all other entries are simply fault entries so that a fault may be generated in the case of an illegal memory access by the process. During a context switch, the MMU coprocessor adjust the location that the TTBR0 (Translation Table Base Register 0) addresses to. This allows for completely different virtual memory spaces for every process. So although each process might be loaded in at the same virtual memory address, they are mapped differently in physical memory. One must take care when setting up a processes page table however, as the kernel code for exception vector handling must still be mapped in memory.

In time for our demonstrations, we were unable to complete loading each process into it's own region of physical memory. See Future Expansion for more details. As such, each process is compiled into the kernel image binary. Data abort exceptions are handled by simply printing a

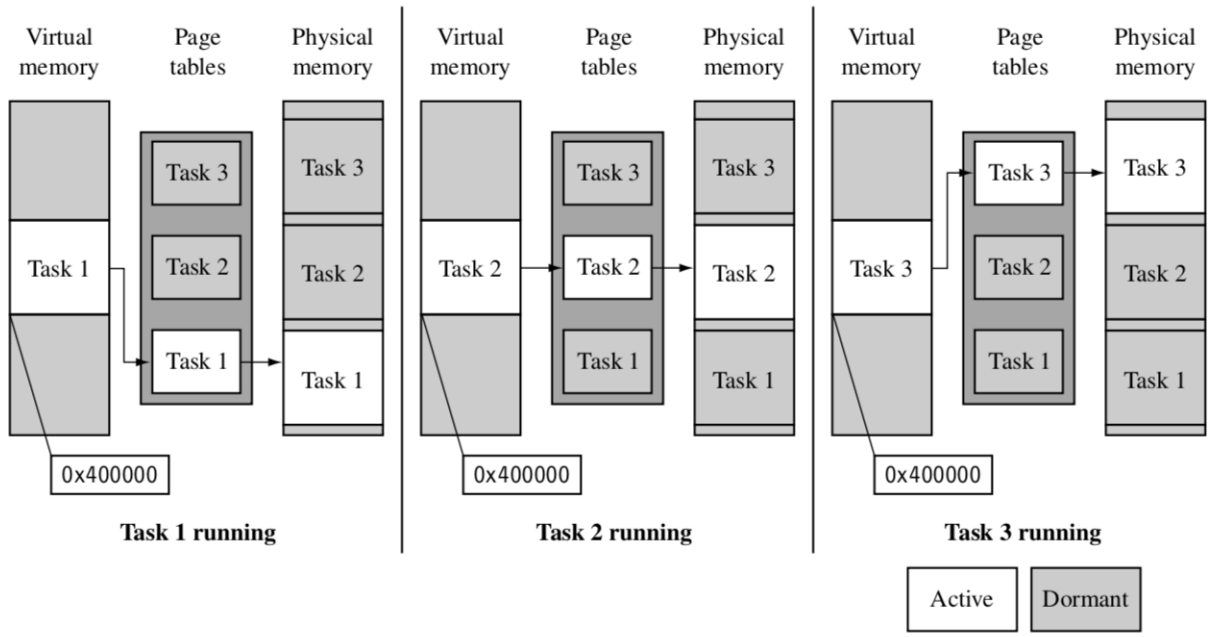

Figure 4: Context Switching Page Tables

warning from the kernel and setting the process status as *ZOMBIE* to prevent it from being scheduled again.

Many expansions to this implementation are recommended as time only permitted a basic implementation.

Using the MMC driver, it is possible to load in new processes after boot into their own regions of physical memory. We did not complete a user library in time for the project, however it would be relatively each to move our existing user facing syscalls into a library. This would allow us to compile user programs separately from the kernel and would give us more flexibility in specifying where these processes will be loaded in.

Using sections made it easier for us to implement the TTBR since we only needed a single level of page tables. In order to implement paging of 16KB, a two level page table would be required to initialize the MMU. This would involve editing the init_pgtbls procedure in *mem.S* to add entries to the second level page table in the upper level page directory.

In our current implementation, we map in the entirety of kernel memory in each processes page table. It is possible to only map in the exception vector procedures and the swtch function which would adjust the page table to a kernel level one. This would allow for increased security, preventing reading or writing to kernel memory.

An issue we had was definitely trying to get the MMU to start probably. Since any small issue when initializing the page tables would cause the processor to hang indefinitely it was hard to debug what we might have been doing wrong. Memory alignment was usually our issue when setting up our page table since ARM has specific rules about how memory must be aligned when setting the TTBR.

3 PROCESSES

Our operating system's implementation of a process is simple. This was done because it was the most expeditious option. In effect, it would have to be changed in the future as it is an overly restrictive model if the project were to continue development.

A process is represented by a `struct`. This struct is labelled as a `struct pcb` [1] and contains a few key elements.

Listing 1: src/proc.h:pcb

```
struct pcb {
    struct context context;
  volatile unsigned int pgtbl[NUM_SECTIONS]
    __attribute__((aligned(SECTION_ALIGNMENT)));
    unsigned int pid;
    enum proc_state state;
    struct pcb *sender_queue[NUM_PROCS];
    unsigned int s_q_head;
    unsigned int s_q_tail;
    char msg_buf[MSG_SIZE];
    unsigned int msg_len;
};
```

- The process context, which is key to process scheduling [4] and is explored in more detail in section [5].
- The process page table, which is key to the MMU memory protection, and is explored in more detail in section [2].
- The process PID, which is used to uniquely identify a process.
- Process state, used to manage whether a process is the currently running process, is runnable, is blocked, or hit an exception from which it can't recover. This is shown in the code below [2].
- Several fields relating to messages. These are explored in more detail in section [7].

Listing 2: src/proc.h:proc_state

```
enum proc_state {
    RUNNING,
    READY,
    BLOCKED,
    ZOMBIE
```

```
};
```

Although our operating system does have support for dynamic memory allocation, we did not take advantage of it for our process implementation. Instead, PCBs are allocated globally on the stack [3]. This greatly simplified how we allocated process' memory on the stack.

Listing 3: src/proc.c:procs

```
struct pcb procs[NUM_PROCS];
```

Process PCB information is initialized in the function `proc_init` [4][5].

Listing 4: src/proc.c:proc_init

```
user_stack_top = (unsigned int)&sys_stack_top;
i = 0;
pid = 0;
pcb = &procs[i];
pcb->state = READY;
pcb->pid = pid++;
pcb->context.ret_pc = (unsigned int) &null_proc;
pcb->context.sp = user_stack_top - (i * STACK_SIZE);
pcb->context.spsr = 0x1F;
```

Listing 5: src/proc.c:proc_init

```
for (i = 1; i < NUM_PROCS; i++) {
    pcb = &procs[i];
    pcb->state = READY;
    pcb->pid = pid++;
    pcb->s_q_head = 0;
    pcb->s_q_tail = 0;
    pcb->msg_len = 0;
    bzero(pcb->msg_buf, MSG_SIZE);

    /* ... */

    pcb->context.sp = user_stack_top - (i * STACK_SIZE);
    pcb->context.spsr = 0x10;
    init_user_pgtbl((unsigned int*)pcb->pgtbl);
```

Because there is a predefined and static number of process which can be active, processes

can have their stack memory allocated on startup. This is exactly what was done. In [5], process context stack pointers are set to an offset from a predefined position. As shown in [4], this is the `sys_stack_top`, an address which is specified in src/entry.S. In this way, every process is pre-allocated a portion of the stack in which to execute. Unfortunately, our operating system does not support *real* user applications — all processes have their entry point defined in the kernel binary and those entry points exists in the kernel binary as well. This makes the current iteration of this operating system unsuitable for any general purpose use.

4 SCHEDULING

Scheduling processes implies a certain degree of complexity already implemented in the operating system. If more than one process can exist to be scheduled, then we have already:

1. implemented the concept of a process,
2. have the ability to manage memory such that processes are either swapped in and out fully or they share the memory via partitioning, and
3. have developed a method by which processes can either yield the CPU to the scheduler or by which they may be interrupted.

In our operating system, all three of these conditions were met. By implementing a process context struct [3], context switching [5], and system calls [6], we have the foundation upon which process scheduling can be implemented.

We implemented a round robin scheduling algorithm. Essentially, this revolves around a for loop that iterates through the global list of processes [3]:

Listing 6: src/proc.c:scheduler

```
while (1) {
  i = (i + 1) % NUM_PROCS;
  count++;

  if ((procs[i].pid != 0 && procs[i].state == READY) ||
      (procs[i].pid == 0 && count > NUM_PROCS)) {
    /* Always enter for the null proc since it's always running or
       ready, but only after every other process has been checked first
          */
    if (curr_proc->state == RUNNING) {
      /* Reset to ready only if its status has not changed */
      curr_proc->state = READY;
    }

    curr_proc = &procs[i];
    curr_proc->state = RUNNING;
    swtch();
  }
```

```
}
```

As this loop enters, `i` has already been set as the PID of the current process. Remember, these PIDs are assigned sequentially during process initialization [3]. `i` is incremented (to give the next PID of the next process in sequence) and we check that either:

1. `i` is not the `NULL` process and is in a ready state.
2. If the `NULL` process is running and we have already checked the status of every other PID (`count` is initialized to 0 upon function entry).

If either of these conditions are met, we can change the state of the current process to `READY` from `RUNNING`, assign process `i` as the current process, and call `swtch`, which will switch process contexts and continue the new process' execution [5]. In the first case, this means passing along execution to the next `READY` process in the PCB. In the latter case, it means either continuing or starting execution of the `NULL` process, which will run until the timer interrupt fires. If neither of these conditions are met, we simply iterate through PIDs until one of them is met. As such, this implementation therefore acheives a very simple round-robin scheduler.

The scheduler is called in several places:

- In the interrupt handler for the timer:

Listing 7: src/timer.c

```
if (count % 10000 == 0) {
  HWREG(CM_PER_GPIO1_CLKCTL) |= 0x00000002;
  HWREG(GPIO1_OE) &= ~0x00200000;
  HWREG(GPIO1_DATAOUT) ^= 0x00200000;
  if (preempt)
    scheduler();
}
```

- by the `yield` [3] system call:

Listing 8: src/syscalls/yield.c

```
void sys_yield() {
  scheduler();
}
```

- By the `send` [5] system call:

Listing 9: src/syscalls/send.c

```
/* Block until message is received */
```

```
curr_proc->state = BLOCKED;
scheduler();
```

- As well as in the data abort interrupt handler:

Listing 10: src/interrupt.c

```
void data_aborted(void) {
    uart0_string_put_polling((unsigned char*)"[KERN] Aborted\r\n",
       16);
    curr_proc->state = ZOMBIE;
    scheduler();
    while(1);
}
```

This pattern of invocation leads to conventional behaviour. When a process issues a system call that would result in it waiting on I/O, or if the the system call is `yield` specifically, the process ceases execution in favour of the next-in-line. Otherwise, processes are rescheduled with the predictability of their time-slice — determined by the timer interrupt.

Improving the scheduling algorithm to increase fairness or efficiency any more is dependent on an increased complexity in process implementation. Without support for process priority (even two or more priority levels), process scheduling queues can't be implemented. This is the next step with regards to improved scheduling.

5 Context Switch

Members Responsible: Erik Andvaag & Rowan MacLachlan The context switching method used is very simplistic. On entry into the kernel (either through a system call or exception) all of the user registers are saved, as well as the saved program status register (spsr) and link register of the newly entered mode (either irq_lr or svc_lr). The link register contains the value that the program counter should be set to in order to return to the user process. All of this state is saved within the process control block of the running process. Therefore, the work needed to switch out the currently running process is already done by the time execution reaches the swtch() routine.

When the scheduler (proc.c) selects a new process to run, it calls `swtch` (swtch.S) to switch in the new process. `swtch` simply loads the user registers with the values stored in the new process's PCB and then loads the program counter with the saved link register value. This causes execution to jump directly from the swtch() routine to the new user process. No kernel state is ever saved. The 'kernel' stack (either the supervisor stack or the IRQ handler stack) is effectively abandoned, without unwinding. Therefore on entry into the kernel, the stack pointer (supervisor stack pointer or IRQ stack pointer) must be reset to point to the top of the appropriate stack.

There is one additional complication to the context switch. As described in the exception handling section, when a user process is preempted by a device interrupt, the irq handler must

acknowledge the interrupt before returning to user space, in order to allow new interrupts to be raised. However, if an IRQ exception causes the currently running process to be switched out (as when a timer interrupt occurs), the process will never return to the irq handler routine. The interrupt controller will never receive the acknowledgment it needs. Hence, it is the job of the `swtch` routine to perform this acknowledgment before jumping to the new user process. This is accomplished through the use of a global variable called was_irq. This value is set on entry into the kernel. The swtch() routine checks the value of this variable; if it sees that the switched out process entered through the irq handler, it will perform the acknowledgment. This approach works, but is inelegant.

In general, although the context switching design is adequate for switching between user processes, it is fairly limited. The reason for this is due to the fact that the context switcher jumps directly to where the switched-in user process last left off. This is a restrictive design, because it is often the case that the kernel needs to perform more work on behalf of a switched-in process before returning to user space. For example, IPC which is blocking for both `send` and `receive` would be difficult to implement under the current design. The most natural implementation of blocking `send` and `receive` is to have the receiver copy the sender's data into its address space after it has been woken up and before returning to user space. This is not possible with the current implementation. A more flexible scheme would save the kernel context and call chain in addition to the user's values. This would likely necessitate having a separate kernel stack for each process.

Although the final implementation was not very complicated, it was challenging to determine what steps needed to be taken in the `swtch` routine. This was partly due to the order of development. The system call and IRQ handlers had already been written, but they were dissimilar from each other. It quickly became clear that in order to be compatible with a single context switching routine, the two handlers would need to be compatible with each other. Thus introducing context switching to the system required a re-design of these handlers. It was also unclear at first where the user state should be saved - in the PCB, or on a dedicated kernel stack. Ultimately, the PCB was chosen for simplicity.

6 System Calls

Members Responsible: Erik Andvaag & Rowan MacLachlan The implementation of system calls can best be described by tracing through the flow of execution which occurs when a system call is invoked from a user process. A system call is initiated when a user process calls a function from the system call API. The system call API functions (src/sysapi.c) are stubs which call a generic `syscall` routine. Each API function passes a variable number of arguments to `syscall`. The first argument is always the system call number (defined in include/syscall.h); the remaining arguments are the system call arguments provided by the user.

The `syscall` routine (src/syscall.S) performs the software interrupt instruction (swi) which causes execution to trap to the kernel. The number given to the swi instruction is irrelevant; the system call number has been stored in the r0 register, since it was the first argument provided to `syscall`.

The software interrupt causes a jump to the `svc_handler` (exceptionhandler.S). Additionally,

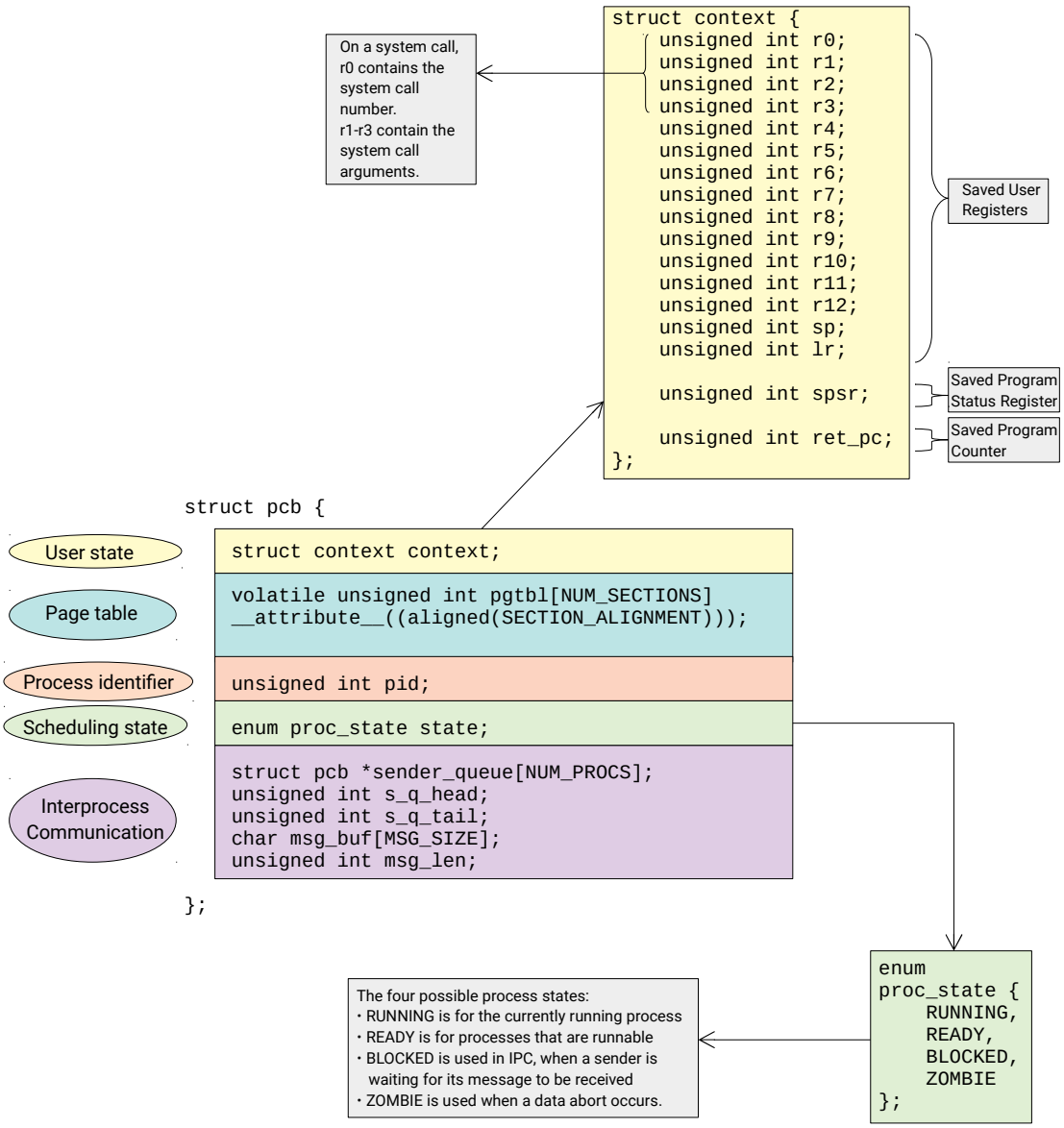

Figure 5: Process Control Block Layout

this instruction causes the processor to switch from user to supervisor mode. The handler stores all of the user register values into the context field of the currently running process's process control block. Thus, since registers r0-r3 have been loaded with the system call number and arguments, these values will be stored in the PCB for later retrieval. After storing the user registers (along with the svc link register and spsr), the `svc_handler` will jump to the sysjump routine.

The `sysjump` function (syscall.c) has the responsibility of jumping to the appropriate system call routine. It does this by consulting the system call number stored in the PCB. Sysjump uses the system call number to index the syscalls jump table. This structure is an array of function pointers containing the addresses of the various system call routines.

Execution has now reached the system call routine within the kernel. Although system call arguments can be retrieved directly from the PCB, care must be taken when handling user values, especially pointers which may point outside of the user's address space. The functions argptr and argint (args.c), are intended to retrieve system call arguments from the PCB, while also checking their validity. However, because the current version of our system does not provide each process with its own address space, these functions do not actually perform any checks.

A system call is able to return a value to the user by storing the value in the r0 field of the PCB's context structure. On return from the system call, the `svc_handler` will load the user registers with the context values. This return value will then be passed along through the syscall routine and API function back to the user.

In total, seven system calls were implemented: write(), uptime(), yield(), getpid(), send(), receive(), and acquire_syscall(). A brief description of each follows.

1. **write()**: The write system call allows the user to send characters through the UART. write() takes three arguments: a file descriptor, a pointer to a character buffer, and a count of the number of bytes the user wishes to output. The file descriptor is unused in the current implementation; write() is only used for serial transmission. The write system call calls the uart0_string_put_polling() routine to output the character string provided by the user.

2. **uptime()**: The uptime system call returns the number of ticks that have occurred since start-up. Ticks is a global variable updated in the timer interrupt service routine.

3. **yield()**: The yield system call allows the currently running user process to cede control of the CPU to another process. It achieves this by simply calling the scheduler.

4. **getpid()**: Returns the PID of the currently running process. The PID is another field of the process control block; getpid() returns this value to the user process.

5. **send()**: Sends a message to another process. The implementation is further described in the IPC section [7].

6. **receive()**: Receives a message from the another process. The implementation is further described in the IPC section [7].

7. **acquire_syscall()**: Acquires a spinlock in an atomic fashion. The implementation is further described in the IPC section [7].

If the operating system was further developed, many additional system calls could be added. Process-related system calls, such as fork(), exec(), and exit() would be good candidates. File I/O calls, such as open(), close(), read(), and write(), would also be useful additions. Although write() was implemented, it can currently only be used for serial output. A more complete implementation would allow the user to write to a specified file.

In general, system calls were easier to implement than other parts of the system. Two primary sources were referenced. The Simple Little Operating System from the ARM System Developer's Guide provided inspiration for the generic syscall routine and the system call handler. xv6 was referenced for the jump table and argument retrieval routines. Although system calls were functional relatively early on, it later became apparent that the design was needlessly complicated and some unnecessary steps were removed.

7 Inter-Process Communication

Members Responsible: Rowan MacLachlan

7.1 Send and Receive

Send() and receive() system calls were added to allow for basic communication between user processes. As described earlier, the current context switching design does not allow both send() and receive() to be blocking operations. It was therefore decided to make send() blocking and receive() non-blocking. The send() system call copies the arguments it receives into the process control block of the currently running process (the sender). Two fields were added to the PCB struct: a message buffer and a message length. After storing these values, the sender enqueues itself onto the receiver's sender_queue. (This was another field added to the PCB, a statically allocated array of size NUM_PROCS). Next, the sender's state is set to BLOCKED, and the scheduler() is invoked.

Receive() checks if any process has been added to the calling process's sender_queue. If it finds the queue to be empty, it exits immediately. Thus, receive() is non-blocking. If it finds a waiting sender, it will copy the sender's message and message length into the addresses specified by the caller of receive(). Then it will set the sender's state to READY – now that the receiver has received the message, the sender is allowed to run again.

7.2 Locks and Semaphores

Basic spinlock and semaphore structures were implemented for user space synchronization and protection. Initially, the spinlock used exclusive load and store instructions (ldrex and strex) to achieve atomicity. However, when MMU support was added, the exclusive load instruction caused a data abort. As a result, the lock implementation was re-designed. A new system call was introduced: acquire_syscall(). Acquiring a lock thus requires entering the kernel. The kernel can guarantee that the acquire operation is atomic, since all interrupts are disabled within the kernel.

The semaphore structure was built on top of the spinlock. The semaphore makes use of a lock to achieve mutual exclusion within the up_sem() and down_sem() operations. The semaphore queue is implemented as a statically allocated array of pointers to process control blocks.

7.3 Future Expansion

It would be better to implement lock acquisition using the exclusive load and store instructions, since they are more efficient than making a kernel call. Additionally, if the kernel was made preemptive in the future, the current implementation would not guarantee atomicity.

The semaphore implementation could be improved if dynamic memory allocation was added. With dynamic memory allocation, the semaphore queue could be a linked list rather than a static array. The current implementation is adequate if there are only a small number of processes, but it is not scalable.

7.4 Pitfalls

The primary difficulty in the development of IPC was caused by an unclear idea of which part of the system the IPC was intended for: the user processes or the kernel. It eventually became apparent that the answer to this question would affect the implementation. It was finally decided that the locks and semaphores would be for user processes. This was chosen for two reasons: (1) the kernel had no need for critical section protection since it always disabled interrupts, and (2) it was easier to demonstrate and test the locking mechanisms from a user process.

8 INTERRUPTS

Members Responsible: Erik Andvaag & Jian Su. The primary uses of interrupts within our operating system will be for preemptive scheduling and an interrupt-driven UART driver. The steps required for servicing interrupts intersects with those needed for system call handling, so the design and development of these two areas will be need to be done in tandem.

We present several options for the design of our interrupt handler:

- A non-nested interrupt handler. This would be the simplest and least-capable interrupt handling design to implement. Interrupts are disabled from the very start of the interrupt servicing routine. They are not re-enabled until the routine is finished and control is returned back to the interrupted task, at which point other interrupts may be serviced.
- A prioritized interrupt handler. This would be a slightly more complicated interrupt handler based on the premise that interrupts of higher priority should be serviced before those of lower priority. Interrupts would need to be prioritized. After returning from an interrupt servicing routine, pending interrupts would be scanned for that highest priority interrupt which would be handled.

- A nested interrupt handler. This would be significantly more complicated to implement, although it would greatly reduce interrupt servicing latency. Interrupts from different classes would need to be prioritized. Higher priority interrupts would interrupt the service routines of lower priority interrupts, and interrupts would only be disabled for the brief periods in which important data was being handled.

There are other more specific schemes available too. One of the difficulties of designing the interrupt servicing mechanisms of our operating system is their relation to the interrupted processes and process contexts. Interrupts and nested interrupts all impose a greater level of complexity which may be difficult to first handle without the risk of destabilizing or losing process state.

For this reason, the preliminary targets of our operating system should probably include the simplest interrupt handling design: that of a non-nested interrupt handler.

The implementation of this non-nested interrupt handler is in src/exceptionhandler.S. The handling process can be divided into 3 phases. The first phase is for context saving. In this phase, the handler first saves the user state including user registers r0 to r14 in the process control block (PCB) of the current process. This design decision of using PCB instead of the IRQ stack to store user context is made for the convenience of context switching. The spsr register and the return address of this handler (stored in the banked register `lr_irq`) are also saved in the PCB. Additionally, a flag `was_irq` is set, which is also used in the context switching. The corresponding interrupt serveice routine (ISR) runs in the second phase. It runs in IRQ mode using the IRQ stack. The active interrupt number is obtained from the `INTC_SIQ_IRQ` register. Execution jumps to the ISR by indexing the interrupt vector table with the active interrupt number. After the ISR returns, the user context is restored in the third phase. This is accomplished simply by calling the swtch() routine. A new process has not been chosen to run, but the swtch() routine can still be used to return to the currently running user process. swtch() will load user registers r0 to r14 from the PCB first. Then, by setting the pc to the stored return address and cpsr to spsr, execution will return back to the point in user space before the interrupt happened.

Interrupt related data structures and routines are defined and implemented in src/interrupt.c; the corresponding header file is include/interrupt.h. An interrupt table `irq_handlers` is used to containe all registered ISRs. In our system, two interrupt sources, the UART0 module and the DMTimer2 clock module, have registered their ISRs. In the ARM Cortex A8 processor, each interrupt source has a predefined number. The UART0 module uses 72 and the DMTimer2 uses 68. Interrupt realated routines include:

- `int_AINTC_init`: initializes the interrupt controller on the processor. The interrupt priority threshold mechanism is disabled here. By doing this, no priority masking is provided by the interrupt controller. Because our system is using a non-nested interrupt handler, it actually does not matter if this mechanism is disabled. If a nested interrupt scheme is implemented, this mechanism can be used to enable fast processing of high priority interrupts. This function also registers a default handler for all interrupts.

- `int_register`: registers an ISR in the interrupt vector table. The index used by each ISR should be consistent with the predefined number of the interrupt.

- `int_priority_set`: assigns a priority to an interrupt and routes it to IRQ or FIQ. The priorities of interrupts are used in the priority sorting mechanism provided by the interrupt controller. When multiple interrupts with different priorities and the same type (IRQ or FIQ, in our system, only IRQ is supported) occur simultaneously, the interrupt with the highest priority is serviced first.

In the future it would be possible to expand the interrupt handling to a two tier model with a reentrant kernel. The two tier model would use a upper and lower half to separate processing the ISR between different priorities. Having a priority based queue in the lower half for handling less time sensitive tasks and a upper level for disabled interrupt handling would allow for a faster response time to multiple interrupts being triggered concurrently. This model would also make it easy to prioritize between different devices.

The first challenge in adding interrupts to the system was deciding what components were needed (vector table, general IRQ handler, table of device handlers, and initialization and service routines). Next, it was necessary to determine which steps needed to be performed in the generic IRQ handler. The StarterWare and Technical Reference Manual provided by Texas Instruments were referenced here. It gradually became apparent that it was possible to avoid certain features, such as interrupt prioritization. An additional pitfall introduced itself when the vector table needed to be relocated: absolute addressing was needed instead of relative addressing. Finally, as mentioned earlier, many aspects of the IRQ handler required changing once context switching was introduced.

9 I/O

9.1 MMC

Responsible: Braunson Mazoka & Lizzie Adams The MultiMediaCard (MMC) driver is an integral part of the bootloader and therefore was one of the first features our team aimed to develop. The MMC driver was also one of the features that took the longest to implement, as we encountered several challenges during its development that significantly delayed its completion. Nonetheless, the driver was implemented fully in that it can be successfully used to read the kernel from the SD card.

The entirety of the MMC driver is located in `/bootloader/src/drivers/mmc.c` and `/include/mmc.h`. We began by studying section 18 of the AM335x Technical Reference Manual which is entirely devoted to the MMC of the device. Section 18.3 specifies the low-level programming models involved, including the global initialization of surrounding modules, the software initialization flow of the MMC controller, and the further configuration of optional modes. At first, we struggled a lot to understand what exactly we needed to implement considering that it was likely that U-Boot implements at least some, if not all, of the global initialization of surrounding models mentioned. Similarly, although the manual does provide flow charts illustrating the procedures to be implemented for various methods of card configuration, the instructions were not always immediately clear to us.

It wasn't until several weeks later and much reading up on MMC data transfer that we finally discovered a very well-written SD Card Specification document that we made significant progress regarding functional code. The specification provides an excellent description of the initialization procedures that need to be implemented, and additionally provides a complete reference of MMC commands used and their associated response types. The MMC driver works by issuing commands that are sent to the SD device. Each command has a unique command number as well as an associated response format. Some commands make use of an argument, and read/write commands utilize a data buffer provided by the MMC controller. In order to send a command these parameters are specified and we wait until there is a change to the status register, which indicates either command completion or an error. Upon receiving a properly specified command, the MMC device replies to nearly all command types with a response that is stored in a set of controller registers and whose format depends on the command sent. We began to truly make good implementation progress once we learned to properly check the status register for errors after issuing a command, and investigating the response packet to ensure the device was operating as expected.

The initialization procedure implemented can be broken down into four steps: (1) initializing the MMC controller of the processor, (2) applying a soft reset, (3) identifying all connected cards, and (4) selecting a card to communicate with. The controller initialization involves setting internal processor registers in order to select supported voltages, power on the controller, configure the clock and power management, and finally enable SD card interrupts. Once the host controller is initialized, a soft reset must be applied as the first step of card initialization. The host then issues a broadcast command CMD8 that requests that all connected cards indicate their presence. The ACMD41 command is used to communicate the card capacity of all devices that are available, and the controller will drop communication with any devices that are not compatible. All remaining compatible devices will receive the command CMD2, requesting that they send the data contained in their Card Identification (CID) register. The CMD3 command is sent to request that all cards indicate the Relative Card Address (RCA), which can be be used to issue commands to specific cards rather than broadcasting to all devices. The controller requests Card Specific Data (CSD) using CMD9 after which it finally selects the card for data communication via the use of CMD7.

Once a card has been selected, data transfer commands may be utilized. Our system only needs to read from the MMC, so only the read single block command CMD17 is implemented as an API. Perform the write is quite trivial compared to the card initialization progress. A CMD17 is issued and the controller waits for data to arrive at the buffer from which it is then read out. More data is automatically streamed into the buffer as it is read out, until an entire block has been transferred. Although not the most efficient option, the single block read command is simply used in repetition as to successfully read the kernel from MMC memory.

9.2 Serial

Members Responsible: Jian Su. There are six UART modules (UART0 - UART5) available on the processor; only UART0 is used in this project. UART0 works in the UART 16x mode with a baud rate of 115200 and a data format of 8-bit data, 0-bit parity and 1-bit stop. Both the polling mode and interrupt-driven mode are supported for UART0.

The uart driver is implemented in /src/uart.c, and the corresponding header file is /include/uart.h. The following interfaces to other parts of the kernel are provided by the uart driver.

- `uart0_init`: initializes UART0 to polling mode or interrupt-driven mode. A parameter is used to indicate which mode to use.
- `uart0_interrupt_enable`: configures and enables UART0 interrupt.
- `uart0_char_put`: prints out a byte as a character using polling mode.
- `uart0_char_get`: gets a character using polling mode.
- `uart0_string_put_polling`: prints out a string using polling mode.
- `print_bytes_as_hex`: prints out a byte as a hexadecimal number using polling mode.
- `print_bytes_as_bits`: prints out a byte as a binary number using polling mode.
- `uart0_string_put`: prints out a string using interrupt driven mode.
- `uart0_line_get`: gets a line using interrupt driven mode.

To initialize UART0, the following operations are performed for both polling and interrupt-driven modes:

- Enables the clocks for UART0 module.
- Sets up the pin multiplex for UART0 module.
- Resets UART0 module.
- Sets up FIFO configuration: polling and interrupt-driven modes use different configurations here.
- Sets up the baud rate of 115200.
- Sets up the UART data format of 8-bit data, 0-bit parity and 1-bit stop.
- Switches to UART16x operating mode.

`uart0_char_put` and `uart0_char_get` are the two fundamental functions for the polling mode. All other interfaces for the polling mode are built based on them. `uart0_char_put` checks the `LSR` register to wait until the transmit hold and shift registers are empty and then puts a character into the `THR` register. `uart0_char_get` also checks the `LSR` register to wait until at least one byte arrives in the `RHR` register and then gets it.

To enable UART0 interrupts, we need to implement an interrupt service routine (ISR), register it and set up its priority. We need to clear the corresponding mask bit for UART0 interrupt in the interrupt controller register `INTC_MIR_CLEAR` and we also need to enable `THR` and `RHR` interrupts in the uart register `IER`.

The two interface functions for the interrupt-driven mode `uart0_string_put` and `uart0_line_get` work together with the ISR. Two global variables `tx_empty_flag` and `rx_flag` are used. `uart0_string_put` divides the given string into multiple chunks, and waits for the ISR to set the `tx_empty_flag` to send a chunk to the transmitter FIFO. `uart0_line_get` sets `rx_flag` to be 1. The ISR echoes every character it receives, and stores each character to a given location if `rx_flag` is set. Whenever the ISR gets a newline character, it resets `rx_flag` to be 0 to signal `uart0_line_get` of the ending of a line.

Setting up the UART was a non-trivial task that required considerable knowledge of the module and its associated registers. Once again, the StarterWare and Technical Reference Manual were used extensively. There are many registers that need to be initialized correctly and in proper sequence, leaving little room for creativity. Thus, although we were able to understand the steps needed to initialize the UART by carefully tracing the StarterWare code and referencing the register descriptions in the TRM, it remains one of the more unoriginal parts of the system.

REFERENCES

[1] R. Love, *Linux Kernel Development: Linux Kernel Development*. Pearson Education, 2010.

[2] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: design and implementation*. Prentice Hall Englewood Cliffs, 1997, vol. 68.

[3] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Elsevier, 2004.

[4] *AM335x and AMIC110 Sitara Processors Technical Reference Manual*, 10 2011, literature number SPRUH73P, revised March 2017. [Online]. Available: https://www.ti.com/lit/ug/spruh73p/spruh73p.pdf

[5] *ARM Architecture Reference Manual*, 110 Fulbourn Road, Cambridge, England CB1 9NJ, 2018, aRM DDI 0406C.d (ID040418). [Online]. Available: https://static.docs.arm.com/ddi0406/cd/DDI0406C_d_armv7ar_arm.pdf

[6] Linux kernel. [Online]. Available: https://github.com/torvalds/linux

[7] Minix boot documentation. [Online]. Available: www.os-forum.com/minix/boot/

[8] Xv6 for x86. [Online]. Available: https://github.com/mit-pdos/xv6-public

[9] houcheng. Xv6 for armv7. [Online]. Available: https://github.com/houcheng/xv6-armv7

[10] (2017, 08) Beaglebone black boot process. Slideshow for BBB boot process. [Online]. Available: https://www.slideshare.net/sysplay/beaglebone-black-booting-process-78730088

[11] allexoll. Bbb-baremetal. Basic bare metal examples for the BeagleBone Black. [Online]. Available: https://github.com/allexoll/BBB-BareMetal

[12] U-boot process. Overview of U-Boot process. [Online]. Available: http://omappedia.org/wiki/Bootloader_Project

[13] bootloader example. [Online]. Available: https://sourceforge.net/p/starterwarefree/code/ci/master/tree/bootloader/src/

[14] ChaN. Fatfs. An embedded FAT Filesystem Implementation. [Online]. Available: http://elm-chan.org/fsw/ff/00index_e.html

[15] Ti am335x starterware mmcsdlib. [Online]. Available: http://software-dl.ti.com/dsps/dsps_public_sw/am_bu/starterware/latest/index_FDS.html

[16] Ti am335x starterware user guide. [Online]. Available: http://processors.wiki.ti.com/index.php/StarterWare_02.00.01.01_User_Guide

[17] Uart example code. [Online]. Available: https://android.googlesource.com/kernel/lk/+/upstream-master/platform/am335x/ti/

[18] A. F. M. Abdelrazek. (2006) An overview of how arm handles interrupts. [Online]. Available: http://www.iti.uni-stuttgart.de/~radetzki/Seminar06/08_report.pdf