BUDDY OS

"IT'S HARD TO BE BUDDIES WITHOUT COMMUNICATION"

Author(s) Names and NSID:
Marwan Mostafa (mam024)
Noah Phonsavath (nop028)
Aaron DSouza (aad921)
Alexei Doell (cka067)
Diego Cabrera Pell (efm473)
Abdullah Abdullah (kan728)
Emily Hartz-Kuzmicz (job346)
Kevin Zhang (zbk618)

Date: April 9, 2025

CMPT432: Advanced Operating System Concepts
Team 00

Introduction

BuddyOS: It's hard to be buddies without communication! Our focus was to build a kernel with a foundation of IPC and a working network stack. Along the way, we developed a working virtual file system and shell as well.

MVP:

- Booting from our own bootloader
- Basic interrupt handling
- Bare bones memory management
- Physical memory management
- Allocated with a slab allocator
- Single-level FAT12 filesystem
- FCFS scheduler
- At least two running processes
- Shell interface
- IPC

NTH:

- Preemptive RR scheduler
- Ethernet driver & Network stack (UDP)
- VFS
- Variable number of processes

Table of Contents

Bootloader

Overview

BuddyOS utilizes a single-stage bootloader that is loaded directly from a FAT12 formatted SD card during the execution of the SoC's internal ROM bootloader. The designed bootloader is responsible for the following:

- Initializing critical hardware such as clocks, memory, as well as various peripherals that are required by the OS
- Initializing the interrupt controller and enabling some basic interrupts
- Loading the kernel binary from a FAT12 formatted SD card into memory
- Jumping to the kernel's entry point and handing off control to the kernel

Initialization Steps

This section highlights the overall flow of the bootloader as well as the order in which various system-critical hardware components are initialized.

1) **Clock Setup**
    - The following PLLs are initialized:
        i) CORE PLL
        ii) MPU PLL
        iii) PERIPHERAL PLL
        iv) DDR PLL
2) **Interrupts**
    - A simple interrupt handler is initialized that executes various ISRs based on the interrupt that has been triggered.
    - Currently, only the following hardware components are configured with ISRs:
        - TIMER0: Triggered every 1 second
        - UART0: Triggered on every character that is received
3) **DDR Memory**
    - The bootloader configures the DDR3 memory interface on the BeagleBone Black by initializing the EMIF controller and the DDR PHY
    - According to the AM335x Memory Map, the starting address of SDRAM is 0x80000000
4) **UART Setup**
    - UART0 is initialized with a baud rate of 115200 and acts as the primary method of user interaction with the OS
5) **SD Card Driver Setup**
    - The bootloader configures the MMC interface which provides the following functions to read from and write to the SD card
        - *MMCreadblock(uint32_t block, volatile uint32_t* buf)*
        - *MMCwriteblock(uint32_t block, volatile uint32_t* buf)*
6) **Loading Kernel into Memory**
    - A FAT12 driver is initialized which enables reading from and writing to a FAT12-formatted SD card that is inserted into the BeagleBone Black

- The kernel is loaded into memory by reading the kernel binary (KERNEL.BIN) from the FAT12 SD card to a buffer at the start of RAM (0x80000000)

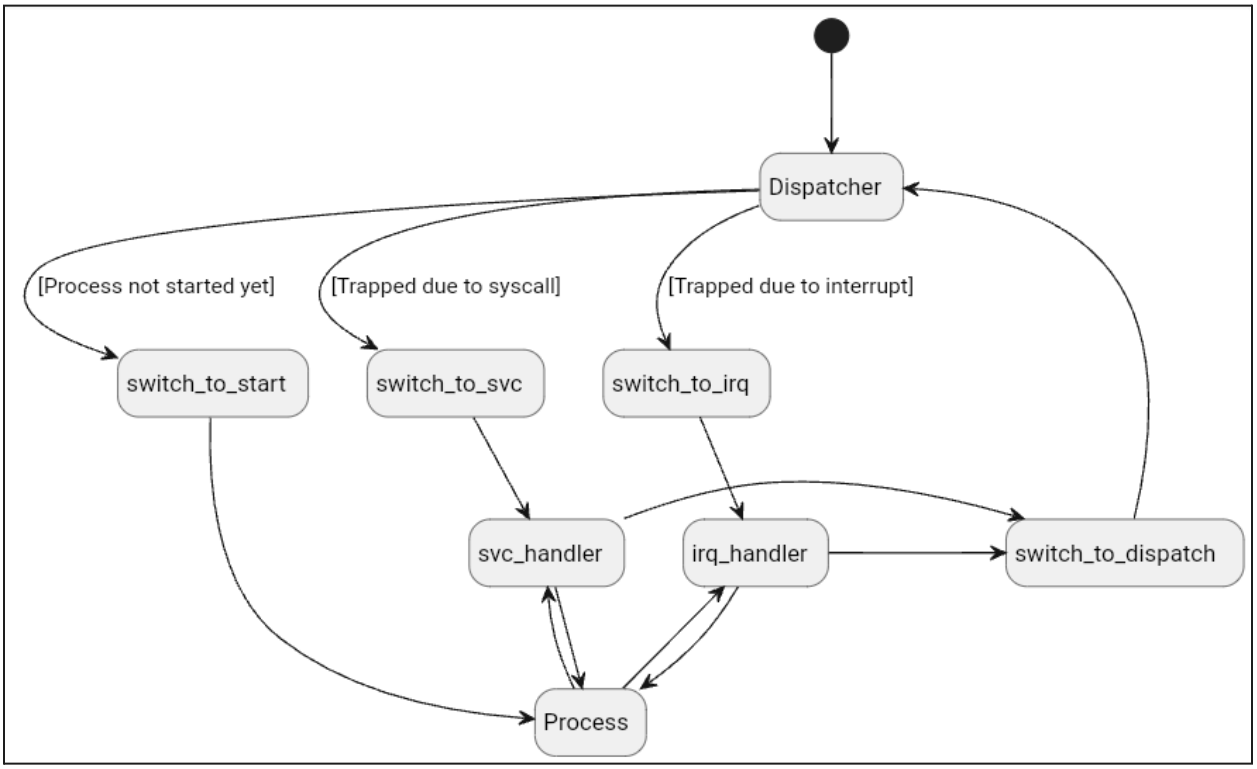
7) **Jump to Kernel**
   - Once the kernel binary is completely loaded, the CPU jumps to the start of RAM and begins executing the kernel

Kernel Overview

Once the bootloader jumps to the kernel, we initialize memory and mount our filesystem. After all initialization is complete, we jump to the dispatcher.

Mode Switching Flow

While in the kernel, the Beaglebone runs in system mode, with a stack located at 0x85000000. When a process is chosen to run, the dispatcher will call a procedure that switches into user mode, and begins executing the process. A process will only ever switch back into the kernel (and therefore system mode) when calling a system call or being interrupted, using one of the specific procedures below to properly switch modes.



User Mode Programs

BuddyOS provides the ability to load binaries from our VFS and create a process to run the code provided that the binary is linked with our system call library. We provide an API for user programs to use and

achieve semi-POSIX compliance with some socket abstractions as seen in one of our test user programs: schat.c.

Memory

BuddyOS provides a memory allocator for dynamic memory in the kernel and user space. The API provides implementations of *malloc(), free(), memcpy(),* and *memset()*. The allocator draws from a resizable pool of memory blocks that contains blocks of different sizes depending on the allocation needs in order to reduce fragmentation on a block-by-block basis. Each size is separated into its own free list.

Memory Map

The kernel memory map is split into chunks and implemented in *include/memory.h*:

- *KERNEL_START* refers to addresses *0x80000000* to *0x80400000*. It is reserved for the kernel the moment it is placed into memory by the bootloader
- *KERNEL_RESERVED* refers to addresses *0x8040000* to *0x80800000*. It is reserved for kernel data structures that need to be quickly accessed and will persist across the kernel's lifetime. Further details regarding the data structures stored here will be provided in the respective sections.
- *KERNEL_DYNAMIC* refers to addresses *0x80800000* to *0x84800000*. It is reserved for the memory block pool used by the allocator. All 64 Mbs are in use, but this can be adjusted to increase the amount of allocatable memory if necessary.

High Level API Overview

The macros defined can be used to change the amount of allocatable memory as well as the size of the largest and smallest block.

- *MAX_BLOCK* is used to define the largest block size available in the memory pool.
- *MIN_BLOCK* is used to define the minimum block size available in the memory pool.
- *MAX_ORDER* is the number of free lists where each list is a different size from min to max.
- *BLOCK_NUM* is the number of blocks in each free list. This is calculated automatically.

Four API calls can be made to allocate, free, and manipulate the dynamic memory:

(1) *kmalloc(uint32_t size)* allocates a block of memory from the pool. The size of the block given will be the nearest power of two that is greater than or equal to the parameter *size*. Block size is limited based on the *MAX_BLOCK* and *MIN_BLOCK* macros; any *size* larger than the max will cause an error, and no block will be smaller than the min, e.g., if *MIN_BLOCK* is 64 and *size* is 10, the user is given a block of size 64. The allocation is performed in *O(1)* time as the block is simply removed from the free list and given to the user

(2) *kfree(void *ptr)* frees the block of memory pointed to by *ptr*. The block is appended to the back of the free list, which is performed in *O(1)* time.

(3) *kmemcpy(void *src, void *dst, uint32_t size)* takes the two pointers which define the source and destination blocks that memory should be copied to and from. The memory is copied

byte-by-byte based on the *size* parameter. This API call is not limited to dynamic memory blocks, which causes some limitations that will be described in the implementation details section.

(4) *kmemset32(void *ptr, uint32_t value, uint32_t size)* takes a memory address and sets the memory of *size* to the value defined in *value.*

The API depends on a free list of each block size, and stores the metadata corresponding to each block within the free block itself. This allows the entire allocator to use *O(1)* memory. These details will be further elaborated upon under the implementation details section.

Implementation Details

The memory is initialized at the start of the kernel by *init_alloc()*. This sets up all of the free lists and pre-allocates every block in the pool to allow for constant time operations for all other calls. This section will walk through everything that occurs during the initialization, which will also describe how the memory is laid out and how the implementations work cohesively to produce the API.

Free Lists

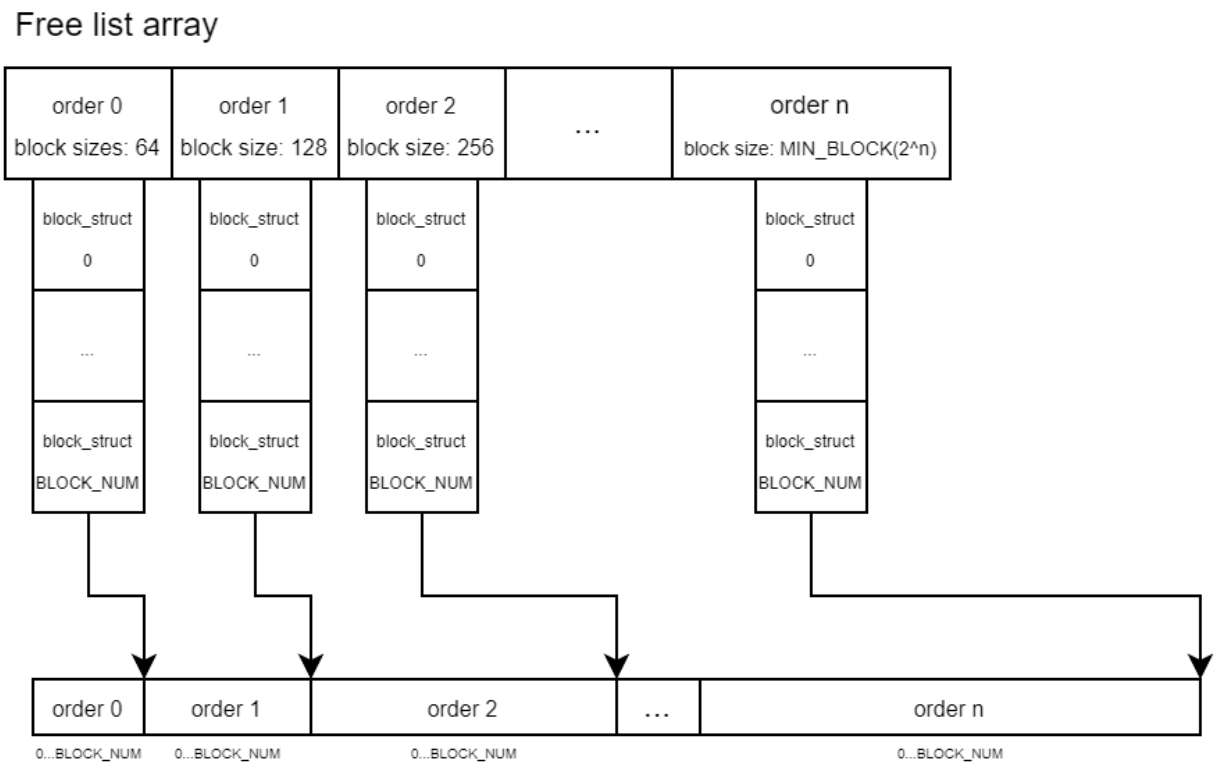
An array is allocated in a static location in memory at 0x80004000 with the number of elements being the number of free lists, aka *orders*. Each element is a *mem_list_struct* with fields *head* and *tail* which are simply memory blocks, and *num_free* which is the number of free blocks remaining in the order. Each memory block struct contains data regarding the size, address, and next block in the list. This way each block acts as a node in the linked list. This data is stored within the free block itself, and *kmemset32()* is used to clear it before giving it to the user; this allows for *O(1)* memory usage.

Initialization

init_alloc() calls a helper function *init_order_arr()*. This function will allocate all the blocks for the *order* param. The blocks are allocated from the *create_mem_block()* function which creates a block for the given order. The size of a block can easily be calculated using *MIN_BLOCK << order*. During *init_alloc()*, the entire memory pool is allocated and the data corresponding to each block is placed in the first few memory addresses of the block. Once the memory is initialized, *kmalloc()* pops from the corresponding order free list, and *kfree()* pushes it.

Calculations

Memory addresses and offsets are all calculated using bitwise operators. This is because all values are powers of 2 which maintain memory alignment and allow for quick calculations. Overall, this allows for modularity in the amount of blocks per order, the smallest and largest block size, number of orders, and total dynamic memory. No values are hardcoded, and all of these can be easily changed by adjusting the macros in *memory.c*.



This figure shows the relationship between the list of order arrays (free lists) and the way the blocks are actually placed in physical memory. Each element is the head of a linked list, which stores all the blocks of that size. The blocks are allocated smallest to largest in memory, and each order has the same number of blocks.

System Calls and Interrupt Handling

First, we will discuss the helper functions that enable system calls and interrupts then list and explain the individual system calls.

Assembly procedures:

supervisor_call:

- Purpose: Saves system call arguments to exception stack then jump to svc_handler.
- Procedure:
    1) Acquire address of current process PCB and load exception stack into SP.
    2) Saves registers (r0-r4 and r11) and the link register (lr) onto the current process's exception stack.
    3) Move current SP into r1, put SVC number into r0 and move SPSR into r11.
    4) Call svc_handler.
    5) Restore SPSR from r11, then restore registers and LR from stack.

6) Return to the calling user process.

irq_interrupt:

- Purpose: Saves system call arguments to exception stack then jump to interrupt_handler.
- Procedure:
    1) Acquire address of current process PCB and load exception stack into SP.
    2) Saves registers (r0-r4 and r11) and the link register (lr) onto the current process's exception stack.
    3) Move current SP into r0, and move SPSR into r11.
    4) Acknowledge interrupt by writing to interrupt controller register.
    5) Call interrupt_handler.
    6) Restore SPSR from r11, then restore registers and LR from stack.
    7) Return to the interrupted user process.

Process Duplication (fork and f_exec):

- Purpose: Duplicates the calling process to create a new child process. fork() will continue from the fork() call of the calling process, whereas f_exec() will start the new process at the beginning of the binary file given as an argument.
- Procedure:
    1) Acquire a free PCB slot by scanning for a PCB with state DEAD using create_process().
    2) If necessary (fork), copy the parent's stack to the child's stack using kmemcpy(), ensuring the same offset is maintained for the saved stack pointer.
    3) If necessary (fork), copy the parent's saved context (registers r4–r11 and the link register lr) into the child's PCB.
    4) If necessary (f_exec), allocate memory for new program and copy file data to new buffer
    5) Set the fork return values: assign 0 to the child and the child's PID to the parent.
    6) Append the child process to the ready queue.
    7) Return the child's PID to the parent process.

Other functions:

1) *interrupt_handler(uint32_t args[])* determines the interrupt source by masking the IRQ source register. This number is used to determine how to handle the current interrupt. Much of the timer interrupt logic is contained within this function as well. Our timer interrupt routine contains quantums for each process, as well as a global timer quantum that lasts one second. Both of these counters are decremented on each timer interrupt, and are only fully serviced when they go below zero.

2) *isr_switch(uint32_t isr_num, uint32_t args[])* saves the IRQ source number and arguments into the current process' PCB, marks the current process as having been interrupted, then calls switch_to_dispatch to switch into the dispatcher. This function is only called when the current process' quantum goes below zero.

3) *svc_handler(uint32_t svc_num, uint32_t args[])* saves the SVC number and arguments into the current process' PCB, marks the current process as having called a system call, then calls switch_to_dispatch to switch into the dispatcher.

4) *kmemset32(void *ptr, uint32_t value, uint32_t size)* takes a memory address and sets the memory of *size* to the value defined in *value.*

User Mode Standard Library:

The user mode standard library API alongside the system calls mentioned in other sections is as follow:

- void printf(const char* str, ...)
- char *fgets(char *str, int n, int stream)
- void* malloc(int size)
- void free(void* ptr)
- int __poll(int file_type, int file_num)
    - Allows polling UART or a socket for data

Processes

The process table itself is a static array of PCB structs. The current process (as well as the kernel process) is kept track of via a pointer to a PCB struct. The stack size of each process is 1024 bytes, the kernel stack size of each process is 512 bytes, and the default quantum given to each process is 1000 ticks to more easily showcase concurrency. The OS has a null process that is constantly running and waiting for interrupts from the kernel, different processes, etc, to give leeway to them to run. BuddyOS's shell is the first process that runs following the null process, and other processes are executed from within it.

PCB (Process Control Block)

The BuddyOS PCB holds all information pertaining to a process. Process state is stored in the PCB as an instance of the *context* struct. This structure contains the general purpose *r4-r11* registers and the link register *lr*. This is not to be confused with the other PCB attribute *state*, which contains the current state of the process–ready (to run), running, blocked (on the blocked queue), or dead. There also exists process priority (low, medium, high) and exit status (to be set by *exit()*). A process's PID and PPID (if it is a child) will be stored in the PCB. The other things stored in the PCB include:

Of the 32-bit unsigned integer type:

- *trap_reason*: the reason for this process trapping. Accounts for a trap via either a syscall or an interrupt.
- *cpu_time*: the remaining ticks left on the process's current quantum
- *status*: SVC/IRQ number
- *saved_lr*: the process' user mode link register (while in system mode)

Of the 32-bit unsigned integer pointer type:

- *stack_base:* the base address of the allocated stack

- *stack_ptr*: the saved context (stack pointer)
- *saved_sp*: the process' user mode stack pointer (saved while in system mode)
- *exception_stack_top*: the process' SVC and IRQ stack pointer
- *r_args*: syscall arguments

Of the boolean type:
- *started*: has this process started running?
- *quantum_elapsed*: has the quantum for this process elapsed?

Of the KList (intrusive linked list) type:
- *children*: a list of this process' children
- *sched_node*: a node to be given to the scheduler's ready queue

Finally, the PCB also holds a *mailbox* attribute for SRR-style interprocess communication through mailboxes

Interprocess Communication

BuddyOS uses blocking SRR-style IPC to facilitate fixed-length FIFO send-and-wait messaging with an uncapped mailbox. To receive messages/replies, a buffer and its length must be supplied to the api functions to store the message. If the length of the buffer is not sufficient to store the full message, the message will be truncated to the length of the buffer. If the length of the buffer is 0 or any of the buffers are NULL or an error occurs, then a negative value will be returned. Otherwise, 0 is returned.

Interface

The IPC service can be accessed through the 4 system calls seen here:

1. *int __send(int pid, void *msg, uint32_t len, void* reply, uint32_t* rlen)* — Sends a message to a process specified by a PID (*pid*). The message in question is the data stored in *msg* of length *len*. Then blocks until a reply is sent back. The reply will be stored in the *reply* variable, where the length of the reply buffer is the integer pointed to by *rlen*. *rlen* will be modified (if no errors occur) to store the length of the reply message stored in *reply*.
2. *int __receive(int* author, void* msg, uint32_t* len)* — Blocks until there is a message to receive. When a message is available, the PID authoring process will be stored in the *author* pointer. The message will be stored in a buffer provided by the *msg* variable. The buffer's length is given by the value stored in *len*. If no errors occur, then the length of the received message will be stored in *len*.
3. *int __reply(int pid, void* msg, uint32_t len)* — Replies to a process specified by a PID (*pid*). The process must have sent the calling process a message (which has not been replied to yet), or a negative value will be returned (error). The message sent back is stored in the buffer *msg*. The length of the buffer is passed by the variable *len*. After replying, the process which is receiving the reply will be unblocked.
4. *int __msg_waiting()* — Simply returns a boolean indicating whether a message is waiting to be received by the calling process. Returns 0 if no messages are waiting; otherwise, 1.

Implementation Details

The mailbox and other associated structures for IPC are defined in include/kernel/srr_ipc.h. To handle an uncapped mailbox, there is a KList (defined in include/misc/list.h) in each mailbox to which MailEntry structs (also defined in srr_ipc.h) are appended to the tail and popped from the head of the list to create a queue (FIFO). When process A sends process B a message, the MailEntry struct is added to process B's mailbox and process B's mailbox count is incremented. Process A's mailbox is also modified to set a variable to show that process A sent a message to B. This variable is used to ensure that replies are only sent to processes which are expecting one. When process B receives the message, the Klist in the mailbox is popped at the head and the count is decremented. When process B replies, a reply message is stored in a special slot in process A's mailbox and the variable is set to 0 so that no repeat replies can be sent.

Scheduler

BuddyOS uses a round-robin scheduling mechanism with support for multiple process priorities. The scheduler:

- Managers a ready queue for processes in the READY state
- Utilises a set of context-switching routines to transition between user, system (SVC) and IRQ modes.
- Relies on a Process Control Block (PCB) to store each process' context

This context includes:

- Registers
- Stack pointers
- Mode-specific state of the process

Scheduler Functions

schedule():

- The schedule() function implements a round-robin policy to select the next process to run.
- Checks if the ready queue is empty; if so, the null process is run.
- If the current process is not blocked, it rotates the process to the tail of the ready queue.
- The next process in the ready queue is then selected and set as current_process.

dispatcher():

- The core scheduling routine, dispatcher(), is in charge of handling, and continuously performs process traps, including syscalls, interrupts, or normal process termination. It invokes the context switch procedures as needed.
- Workflow:
    1) The dispatcher begins by checking whether the current process has started.
        - If not, switch_to_start() is called.
    2) Depending on the trap reason (SYSCALL, INTERRUPT, or HANDLED), it calls the corresponding context switch routine:

    - For system calls, switch_to_svc() is used.
    - For interrupts, switch_to_irq() is used
3) After that process returns to the dispatcher, either their syscall is handled, or in the case of the process' quantum elapsing, the scheduler (schedule()) picks a new process to run and marks it as the current process.
4) The dispatcher prints debug information (when enabled) to trace process execution and trap reasons.
5) This loop continues indefinitely from step 1, ensuring that processes are scheduled appropriately based on their states (RUNNING, BLOCKED, etc).

proc_wrapper(void (*func)(void))

- In order to allow new processes to be safely finished, whenever a new process is started, rather than jumping immediately to its starting address, we use a wrapper function around it. This function merely calls the starting address of the process as a function, then calls the __exit() system call to ensure that the kernel is aware that the process has finished and that the process has a safe LR value to return to for further handling.

Implementation Details

- Each PCB stores not only the process's runtime context but also important data such as the saved SP, LR, and process state. The context switching and scheduler routines rely on these fields to determine where to save and restore the execution context.

Context Switching

Our kernel provides four key assembly routines that we utilize to switch execution contexts between processes. These routines are designed to:

- Save the current process's context.
- Transition the processor in the Beaglebone to the target mode.
- Restore the new process's context and transfer execution to it.

Mode Switch Procedures

switch_to_dispatch:

- Purpose: Switches from the current process (in any mode) to a new process using system mode
- Procedure:
    1) Saves registers (r4-r11) and the link register (lr) onto the current process's stack.
    2) Stores the current stack pointer in the PCB.
    3) Switch the processor to system mode (by setting the cpsr as required).
    4) Banks the user process's stack pointer and link register.
    5) Loads the new process's stack pointer and context.
    6) Branches to the new process via the restored lr.

switch_to_irq:

- Purpose: Handles context switching switching back to an interrupted process.
- Procedure:
    1) Saves the current process's registers and lr.
    2) Stores the current SP in the PCB.
    3) Preloads the user-mode SP and lr into the shared banked registers.
    4) Switches into IRQ mode by modifying cpsr appropriately.
    5) Loads the new process's SP and restores its context.
    6) Branches to the new process.

switch_to_svc:

- Purpose: Used to switch into SVC (supervisor) mode when a process returns from a system call.
- Procedure:
    1) Saves the current process's registers and lr.
    2) Stores the current SP in the PCB.
    3) Retrieves the user process's SP and lr.
    4) Switches to SVC mode via appropriate cpsr modification.
    5) Loads the new process's SP and context.
    6) Branches to the new process.

switch_to_start:

- Purpose: Initiates a process that is running for the first time.
- Procedure:
    1) Sets up the process's stack without needing to preload user specific banked registers
    2) Enables interrupts for the new process.
    3) Switches to user mode.
    4) Loads and restores the process's context.
    5) Branches to proc_wrapper.

Implementation Details

- Special attention is given to managing banked registers (SP and LR) when switching between different CPU modes (user, system, IRQ, SVC). This makes sure that each process resumes execution with the correct context.
- The scheduler ensures that each process is given a minimum quantum (PROC_QUANTUM) to avoid immediate preemption by timer interrupts and simplify control flow. This is crucial as it's used to prevent unwanted rapid context switches during the initialization of a new process before switch_to_start() is even returned from.

Filesystem

High Level Overview

BuddyOS uses a Virtual File System (VFS) as a layer of abstraction between the user and the underlying filesystem implementations. The VFS provides a consistent API to interface with different filesystems, enabling operations like file opening, reading, writing, and closing across supported storage media in a seamless fashion.

Currently, BuddyOS only supports a FAT12 file system on the SD card connected to the device. Support for additional file systems can be enabled by implementing the required functions and by mounting the file systems within the VFS.

Data Structures

vfs.h

1) Mountpoint Structure (mountpoint)
    - *mountpoint_id*: Integer representing index of the filesystem in the active mountpoints table
    - *type*: Integer representing the filesystem type
    - *fs_mountpoint*: String representing path at which the filesystem is mounted
    - *fs_ops*: Function table providing filesystem-specific operations
2) File Descriptor Structure (file_descriptor)
    - *file_name*: Name of the opened file
    - *mountpoint_id*: Mountpoint to which the opened file belongs to
    - *flags*: File open mode (O_READ, O_WRITE)
    - *read_offset*: Current read position of the opened file
    - *write_offset*: Current write position of the opened file
    - *file_size*: Size of the opened file in bytes
    - *file_buffer*: Buffer containing the entire contents of the opened file
3) Filesystem Operations Structure (fs_ops)
    - *open()*: Function to open a file
    - *close()*: Function to close an opened file
    - *read()*: Function to read from the opened file from the current read offset for a specified number of bytes
    - *write()*: Function to write to the opened file from the current write offset for a specified number of bytes
    - *seek()*: Function to move the read/write offset

fat12.h

1) FAT12 Primary Boot Sector (fat12_bs_t)
    - *bootJmp*: 3-byte instruction to jump over disk format information (unused)
    - *oemName*: Original Equipment Manufacturer identifier
    - *bytesPerSector*: Number of bytes per sector
    - *sectorsPerCluster*: Number of sectors per cluster
    - *reservedSectorCount*: Number of reserved sectors, including the boot sector

- *FATTableCount*: Number of File Allocation Tables (FATs)
- *rootEntryCount*: Number of root directory entries
- *totalSectorCount*: Total number of sectors in the volume
- *mediaType*: Media descriptor byte
- *sectorsPerFATTable*: Number of sectors allocated for each FAT table
- *sectorsPerTrack*: Number of sectors per track (unused)
- *headCountOnMedia*: Number of heads on the storage media (unused)
- *hiddenSectorCount*: LBA address of the first sector of the partition (unused)
- *largeSectorCount*: Used when *totalSectorCount* exceeds 65535
- *extendedBootRecord*: 54-byte space reserved for extended boot parameters

2) FAT12 Extended Boot Sector (fat12_ebs_t)
   - *driverNumber*: BIOS drive number
   - *reserved1*: Reserved byte
   - *bootSignature*: Signature indicating presence of an extended boot record
   - *volumeID*: Unique identifier for the volume
   - *volumeLabel*: 11-character ASCII label for the volume
   - *FATTypeLabel*: 8-character ASCII string represent the FAT fs type
3) FAT12 Directory Entry (DirEntry)
   - *name*: 8-character file name (padded with spaces if shorter)
   - *ext*: 3-character file extension (padded with spaces if shorter)
   - *attrib*: Byte indicating file attribute
   - *reserved*: Reserved file (unused)
   - *createTime*: Time of file creation
   - *createDate*: Date of file creation
   - *lastAccessDate*: Date of last file access
   - *firstClusterHigh*: High 16 bits of the first cluster number (unused in FAT12)
   - *modifyTime*: Time of last modification
   - *modifyDate*: Date of last modification
   - *firstClusterLow*: Low 16 bits of the first cluster number
   - *fileSize*: Size of the file in bytes

Filesystem Functions

vfs.c - Contains VFS API that the user can utilize to interact with the file on mounted storage media

(1) uint32_t vfs_mount(char target, int type)*

- **Description**: Mounts a new file system of the specified type at the specified target path. It assigns the appropriate file system operations based on the passed *type* (eg. *FAT12*) and stores the mount point in the global mount point table.
- **Return Values:**
  - *0:* Success
  - *MAX_REACHED:* Maximum number of mount points reached
  - *MEM_ERROR:* Memory allocation failure

(2) uint32_t vfs_open(char path, int flags)*

- **Description**: Attempts to open a file at the given absolute path with the specified flags (*O_READ, O_WRITE)*. This function is responsible for calling the *open* function associated with the underlying file system.
- **Return Values:**
    - *>= 0:* File descriptor of opened file (Success)
    - *MAX_REACHED:* Maximum number of files already opened
    - *ALREADY_OPEN*: File is already open
    - *NOT_FOUND*: FIle not found
    - *INVALID_MOUNTPOINT*: Mount point not found

(3) uint32_t vfs_close(int fd)

- **Description**: Closes the file associated with the passed file descriptor. This function is responsible for invoking the *close* function associated with the underlying file system.
- **Return Values:**
    - *0:* Success
    - *INVALID_FD:* Invalid file descriptor
    - *CLOSE_ERROR*: Error closing the file
    - *NOT_OPEN*: FIle is not open

(4) uint32_t vfs_read(int fd, char read_buffer, int bytes)*

- **Description**: Reads a specified number of bytes from an open file into the passed buffer. This function is responsible for calling *read* from the underlying file system.
- **Return Values:**
    - *>= 0:* Number of bytes read (Success)
    - *INVALID_FD:* Invalid file descriptor
    - *INCORRECT_MODE*: File is not opened in read mode
    - *NOT_OPEN*: File is not open

(5) uint32_t vfs_write(int fd, char write_buffer, int bytes)*

- **Description**: Writes a specified number of bytes from the passed buffer to an open file. This function is responsible for calling *write* from the underlying file system.
- **Return Values:**
    - *>= 0:* Number of bytes written (Success)
    - *INVALID_FD:* Invalid file descriptor
    - *INCORRECT_MODE*: File is not opened in write mode
    - *NOT_OPEN*: File is not open

(6) uint32_t vfs_seek(int fd, int offset, int mode)

- **Description**: Moves the appropriate file pointer as specified by *mode* (*O_READ, O_WRITE)* to the passed offset within the file. This function is responsible for calling *seek* from the underlying file system.
- **Return Values:**
    - *>= 0:* New file offset (Success)
    - *INVALID_FD:* Invalid file descriptor

- *NOT_OPEN*: File is not open

(7) uint32_t vfs_getFileSize(int fd)

- **Description**: Retrieves the size of an open file based on the passed file descriptor.
- **Return Values:**
  - *>= 0:* File size in bytes (Success)
  - *INVALID_FD:* Invalid file descriptor
  - *NOT_OPEN*: File is not open

<u>fs.c</u> - Contains FAT12 specific implementations of VFS operations

(1) file_descriptor fat12_open(const char* path, int flags)*

- **Description**: Attempts to open a file in the FAT12 file system. If the file exists, it loads its metadata and allocates a buffer to store its contents into RAM. If the file does not exist and the file is open with O_WRITE, a new file is created.
- **Return Values:**
  - *Pointer to file_descriptor:* Success
  - *NULL*: File not found, memory allocation failure or file creation failed

(2) uint32_t fat12_close(file_descriptor fd)*

- **Description**: Closes an open file by writing the contents of its buffer back into the disk where it is stored. It ensures that the file's contents are saved back to the disk before freeing resources.
- **Return Values:**
  - *0:* Success

*(3) uint32_t fat12_read(file_descriptor *fd, char* read_buffer, int bytes)*

- **Description**: Reads the specified number of bytes from the opened file to the passed buffer beginning from the current position of the read pointer.
- **Return Values:**
  - *>= 0:* Number of bytes read (Success)

*(4) uint32_t fat12_write(file_descriptor *fd, char* write_buffer, int bytes)*

- **Description**: Writes the specified number of bytes from the passed buffer to the opened file. The write begins in the opened file at the current position of the write pointer.
- **Return Values:**
  - *>= 0:* Number of bytes written (Success)

*(5) uint32_t fat12_seek(file_descriptor *fd, int offset, int mode)*

- **Description**:Moves the read and/or write offset within an open file. The *mode* parameter determines whether the seek operation applies to the read or write pointer.
- **Return Values:**
  - *>= 0:* New offset position (Success)

<u>fat12.c</u> - Lower level fat12 driver

(1) void fat12_init(unsigned int startSector, uint32_t buffer)*

- **Description**: Initializes the FAT12 file system by reading and parsing the boot sector on a FAT12 formatted SD card
- **Parameters:**
  - *startSector*: Sector number where the FAT12 boot sector is located
  - *buffer*: Buffer to be able to read sectors from the SD card
- **Return Values*:***
  - None

(2) int fat12_find(const char filename, uint32_t* buffer, uint32_t* entryIndex)*

- **Description**: Searches for a file in the FAT12 root directory and returns the sector containing the directory entry if it is found
- **Parameters:**
  - *filename*: Name of the file being searched for
  - *buffer*: Buffer to be able to read sectors from the SD card
  - **entryIndex*: Pointer to store the index of the file entry within the sector
- **Return Values:**
  - *>0:* Sector number containing the file entry (Success)
  - *0*: File not found

(3) uint32_t fat12_read_file(const char filename, uint32_t* buffer, uint32_t tempBuffer)*

- **Description**: Reads the contents of a file from the FAT12 filesystem based on the provided filename and stores the read data in the passed buffer
- **Parameters:**
  - *filename*: Name of the file being read from
  - *buffer*: Buffer where the contents of the read file will be stored
  - *tempBuffer*: Buffer to be able to read sectors from the SD card
- **Return Values:**
  - *>= 0:* Number of bytes read (Success)
  - *-1*: File not found

(4) uint32_t fat12_create_dir_entry(const char filename, uint8_t attributes, uint32_t* buffer)*

- **Description**: Creates a new directory entry for a file in the FAT12 filesystem with the passed in file attributes
- **Parameters:**
  - *filename*: Name of the file to create a directory entry for
  - *attributes*: File attributes (*R_ONLY, HIDDEN, SYSTEM, SUBDIR)*
  - *buffer*: Buffer to be able to read sectors from the SD card
- **Return Values:**
  - *>= 0:* First cluster of the created file (Success)
  - *-1*: No available space for the new directory entry

(5) uint32_t fat12_write_file(const char filename, char* data, uint32_t size, uint32_t* tempBuffer)*

- **Description**: Writes data to a file in the FAT12 filesystem. If the file to be written to does not exist, a new directory entry is created before writing
- **Parameters:**
  - *filename*: Name of the file to create a directory entry for
  - *attributes*: File attributes (*R_ONLY, HIDDEN, SYSTEM, SUBDIR)*
  - *buffer*: Buffer to be able to read sectors from the SD card
- **Return Values:**
  - *>= 0:* Number of bytes written (Success)
  - *-1*: File not found

(6) void list_dir(uint32_t buffer, uint32_t allFlag)*

- **Description**: Lists the contents of the root directory in the FAT12 filesystem. Filters out system and hidden files unless *allFlag* is set. Converts filenames and extensions to lowercase for consistency
- **Parameters:**
  - *buffer*: Buffer to be able to read sectors from the SD card
  - *allFlag*: If not zero, includes system and hidden files
- **Return Values:**
  - None

Implementation Details

- The VFS API provides a layer of abstraction for the kernel to interact with the underlying file system
- A typical flow of execution for the vfs is as follows
  1. Kernel calls vfs function
  2. VFS function performs various argument checks
  3. VFS function calls underlying FS specific function, passing it an empty fd struct
  4. FS specific function appropriate initializes the fd struct
  5. FS specific function call file system driver specific functions
- Ex: vfs_read() -> fat12_read() -> fat12_read_file()
- When a file is opened, its contents are loaded into memory in their entirety
- Subsequent reads and writes to the opened file act on the version of the file that is in the kernel memory
- The file is only synced back to the disk when the file is closed

Limitations

- At its current state, all files are stored in the root directory of the file system. There is no support for subdirectories.
- When a file is opened by the VFS, its entire contents are read into the file_buffer that is contained within the file_descriptor struct. This buffer is dynamically allocated using *kmalloc()*.This works under the assumption that the file is small enough to fit within the kernel's memory space.
- A file is only synced back to disk when it is closed. A system crash will result in the file contents not being written back to disk

Testing

Initial testing of the lower level FAT12 driver (fat12.c) included reading text files from the SD card and observing the output of the read operations. If the contents of the files were displayed in their entirety and without corruption, this proved that the read operation was functional. Following this, the create and write operations were tested in a similar fashion. Files were created and written to, after which their contents were read back. If the data that was read from the file reflected what was written to it, create and write operations were confirmed to be functional.

Once the read and write operations were sufficiently tested, the kernel binary was loaded into memory using the developed driver. The kernel being successfully loaded into RAM further validated the functionality and reliability of the lower level FAT12 driver.

The VFS API was tested using shell commands such as *cat* and *write*, which internally invoke the VFS. If these commands correctly read from and wrote to files on the SD card, the functionality of the VFS was confirmed.

UART

High Level Overview

The UART0 interface provides serial communication to assist with debugging, logging, and interacting with the system. There are initialization functions as well as basic and complex text formatting such as uart0_printf(), uart0_fgets(), and uart0_puts(), to name a few.

UART Functions

(1) void uart0_init()

- **Description**: Initializes the UART0 for serial communication, this includes clocks, registers, and modes.
- **Parameters:**
  - *None*
- **Return Values*:***
  - *None*

(2) void uart0_putch(char c)

- **Description**: Transmits a single character c over UART0.
- **Parameters:**
  - *c:* Character to be transmitted.
- **Return Values*:***
  - *None*

(3) void uart0_puts(const char str)*

- **Description**: Sends a null-terminated string over UART0, utilizing uart0_putch() to transmit a sequence of characters.
- **Parameters:**
  - *str:* The array of characters to print
- **Return Values*:***

- *None*

(4) void uart0_putsln(const char str)*

- **Description**: Prints a null-terminated string over UART0, along with a carriage return and new line.
- **Parameters:**
  - *str:* The array of characters to print
- **Return Values*:***
  - *None*

(5) void uart0_printHex(uint32_t number)

- **Description**: Prints a 32-bit unsigned integer in hexadecimal format.
- **Parameters:**
  - *number:* The 32-bit unsigned integer to print in hexadecimal format
- **Return Values*:***
  - *None*

(6) void uart0_printitoa(int num)

- **Description**: Prints an integer as a null-terminated string in base 10.
- **Parameters:**
  - *num:* The integer to print as a string
- **Return Values*:***
  - *None*

(7) void uart0_printf(const char str, ...)*

- **Description**: Prints the string with format specifier (subsequences beginning with %), the additional arguments are formatted and inserted into the resulting string replacing their respective specifiers.
- **Parameters:**
  - *str:* The array of characters to print
  - *...:* These additional arguments represent format specifiers to replace the % in the the string
- **Return Values*:***
  - *None*

(8) char uart0_getch()

- **Description**: This function returns the single character received in the receiver buffer register.
- **Parameters:**
  - *None*
- **Return Values*:***
  - *char:* represents the read character

(9) int uart0_poll()

- **Description**: This function returns the status of polling UART0.
- **Parameters:**
  - *None*
- **Return Values*:***
  - *0:* There is no data waiting in the buffer
  - *1:* Indicates there is data in the buffer

(10) char uart0_fgets(char* str, int n, int stream)*

- **Description**: Reads input from UART0 into str until Enter key or exceed the size of n
- **Parameters:**
  - *str:* Pointer to buffer where string input is stored
  - *n:*Maximum number of characters to read (including null terminator)
  - *stream:* Unused currently but will be expanded in the future
- **Return Values*:***
  - *char*:* Pointer to str, containing the user input

Implementation Details

- The baud rate is 115200
- These functions take inspiration from stdlib.h

Limitations

- The current uart0_printf() only supports a handful of format specifiers such as %c, %d, %p, %s, %x, and %%.
- There are limited debug and verbose modes for UART0 and the print functions.
- At the moment, UART0 functions are not robust, lacking error handling and buffer checking.

User Interface

High Level Overview

BuddyOS provides an interactive shell that communicates with the system through Universal Asynchronous Receiver-Transmitter interface (UART). This allows the user to provide inputs to the system to execute commands and interact with the system, and view outputs on the shell. Combining these two aspects provides an excellent user experience and the ability to take advantage of previously mentioned features such as the Virtual File System (VFS) and user programs.

Key Features

1) Character-by-character input processing: The shell reads user input in real-time for command processing
2) Command echoing with basic editing support: Users will have their input echoed back for visibility and the ability to edit their command
3) Formatted text: The shell utilizes ANSI escape codes to enhance user input through coloured output, easy to read formatting, and appealing visuals

Command Parsing

The shell will continue to take input unless the user inputs the Enter key or they have exceeded the shell buffer size. The command processor will then break this user input into tokens and execute the following command.

Built-in Commands

1) exit - terminates the shell session
2) help - displays a list of available commands
3) clear - clears the shell display
4) echo <text>- prints the provided text to the shell
5) ls - display directory contents
6) cat <filename> - display file contents
7) write <filename> <text> - writes text to file
8) ping <ip> sends icmp echo request to ip

Shell Functions

(1) int drawInitialized()

- **Description**: Initializes the shell with a Goku and Vegeta ascii art
- **Parameters:**
  - *None*
- **Return Values*:***
  - *0:* Successful execution

(2) void cat(char filename)*

- **Description**: Prints the contents of a file in the vfs
- **Parameters:**
  - *Filename*: Name of file to display contents from
- **Return Values:**
  - *None*

(3) void exec(char filename)*

- **Description**: Executes a binary file in the vfs
- **Parameters:**
  - *Filename*: Name of file to execute
- **Return Values:**
  - *None*

(4) int write(char filename)*

- **Description**: Prepares the file in the vfs for writing
- **Parameters:**
  - *Filename*: Name of file to write to
- **Return Values:**
  - *None*

*(5) int parseShellCommands(char** tokens)*

- **Description**: Parses the shell command received as an array of tokens, and performs the corresponding system function
- **Parameters:**
  - *tokens:* Represents an array of strings where the first token is the command and remaining tokens are the arguments, all of which are passed through user input
- **Return Values:**
  - *0*: To terminate the token parsing

    - *1:* Indicate shell should continue parsing tokens

(6) int shell()

- **Description**: Initializes and runs barebones shell to take user input through UART. This input is then tokenized and executes the appropriate shell command based on the tokens.
- **Parameters:**
    - *None*
- **Return Values:**
    - *0: Shell successfully terminated*

Implementation Details

- As mentioned, the shell interface integrates multiple subsystems such as UART, VFS, and executing user programs.
- Justification for macros and magic numbers
    - #define GREEN "\033[0;92m"
        - ANSI escape code to change text colour for standard prompt root>/home$
    - #define RESET "\e[0m"
        - Reset ASNI formatting
    - #define ENTER 13
        - ASCII character code for carriage return \r, to detect the Enter key
    - #define BACKSPACE 8
        - ASCII character code for backspace, to edit user input
    - #define SHELL_BUFFER_SIZE 256
        - For the beagle bone black, 256 bytes was sufficient for the command needs, but small enough for the constrained system
    - #define TOKEN_BUFFER_SIZE 64
        - Supports 64 tokens for longer commands
- The shell will parse user inputs and tokenize them. The appropriate user commands are then executed. Unless the user inputs the exit command, the shell will continue to consume user inputs.

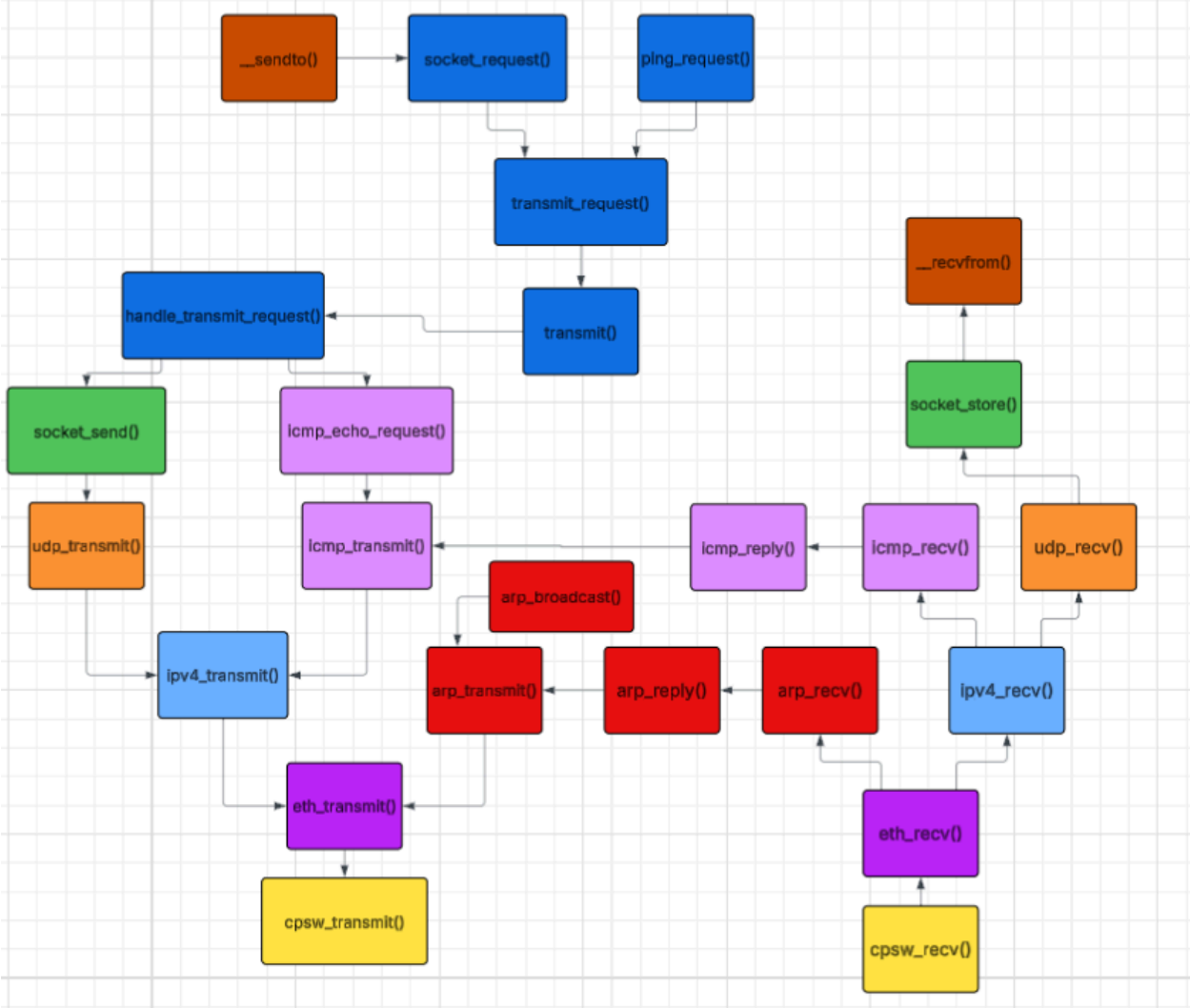
Limitations

- Currently, if the user enters an invalid command or incomplete argument, the shell has a lack of error handling. Implementing greater error handling would assist in providing useful error messages and feedback.
- The shell only supports a handful of basic commands such as exit, help, clear, echo, ls, cat, and write. Integrating more commands would create a more functional shell.
- Building on the previous point, supporting redirecting and piping commands would allow for a more efficient workflow.
- At the moment, there is no history support. Therefore, users cannot utilize the up and down arrow keys to cycle through previously entered commands.

Network Stack

High Level Overview

BuddyOS Provides a Network stack for processes to use, processes can use the socket system call api to create, bind, and use sockets for transmitting and receiving UDP packets. The network stack supports various protocols such as Ethernet II, ARP, IPV4, ICMP, and UDP. The stack uses the Beaglebone Black's Common Platform Switch (CPSW) to send and receive frames over the ethernet cable.



- *Common Platform Switch - Yellow*
- *Ethernet II - Purple*
- *Address Resolution Protocol - Red*
- *Internet Protocol version 4 - Light Blue*
- *Internet Control Message Protocol - Light Purple*
- *User Datagram Protocol - Orange*
- *Upper Half Network Driver - Dark Blue*
- *Sockets - Green | User Socket Syscalls - Brown*

Network Stack Layers

The Network Stack works in a layered fashion which can be seen in the above figure, On frame reception the frame is passed up the stack starting in the cpsw_recv() function. The received frame is passed up to eth_recv() where the frames ethernet header is extracted, from here the frames protocol is checked if it is ARP it is passed to arp_recv(), if it is IPV4 it is passed on to ipv4_recv(). In arp_recv() the ARP header is extracted if the frame is an ARP request for our IP we send a reply. In ipv4_recv() the IPV4 header is extracted and the protocol is checked if it is an ICMP frame it is passed to icmp_recv(), if it is UDP it is passed to udp_recv(). In icmp_recv() the ICMP header is extracted, if the frame is an ICMP echo request we reply to it. In udp_recv() the UDP header is extracted and the destination port is used to check to see if any socket is bound on the port, if there is a socket bound to the port the frame is sent to socket_store() alongside the socket number. socket_store() stores the payload of the UDP frame in the sockets packet queue. User programs can retrieve the payload by calling the __recvfrom() system call with the binded sockets number, the system call returns the first pending packet in the queue.

On packet transmission User programs call the __sendto() with the socket number and frame to be transmitted. The__sendto() system communicates with the upper half network driver and creates a socket transmit request, socket_request() calls transmit_request() and places a socket transmit request on the transmit queue. When transmit() is called it checks the queue for any requests, if there is a request the request is handed over to handle_transmit_request() which checks the request type and calls the appropriate function. On socket requests the request is handed to socket_send() which calls udp_transmit(), the function appends the UDP header to the packet and passes it to ipv4_transmit(). The IPV4 header is appended to the packet and passed down to eth_transmit() which appends the ethernet frame header. The fully crafted frame is passed to cpsw_transmit() which takes the frame from memory and transmits it.

Common Platform Switch

The Beaglebone Black contains a TI Common Platform Ethernet Switch (CPSW) which is a layer 2 three port switch. Its main functionality is to place received frames in memory and transmit frames from memory using CPDMA queues. In order to access the functionality of the ethernet switch, BuddyOS provides a driver for the CPSW.

CPSW Driver API

The Kernel has the following functions available to it to use the CPSW:

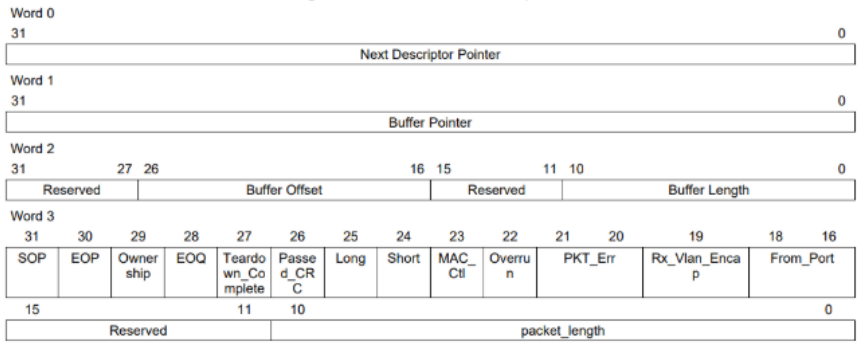
1) *void cpsw_init()*
   Description: *Initializes the CPSW for packet reception and transmission*
2) *int cpsw_recv()*
   Description: *Processes all packets in the cpdma receive queue*
3) *int cpsw_transmit(uint32_t* packet, uint32_t size)*
   Description: *Adds packet to cpdma transmit queue and transmits it*

CPSW Initialization

- Select interface
    - Choose between Reduced Media Independent Interface (RMII) and Media Independent Interface (MII).
    - The Media Independent Interface facilitates the transfer of Ethernet frames between the MAC and PHY layers
    - RMII has less pins then MII
    - Select MII interface by setting bits 0-3 to 0 in the config gmii_sel register
- Pin Muxing
    - Setting the functionality of the multiplexed pins of the CPSW
    - Each pin has a conf_ register where its functionality can be set by setting the specific bits
    - Each pin is set to a fast slew rate by setting bit 6 to 0
    - Each pin pullup/pulldown is enabled by setting bit 3 to 0
    - Each pin is set to pulldown by setting bit 4 to 0 except mdio pins
    - Pins that require input set bit 5 to 1 to enable receiving
    - Mdio pins are set to pull up by setting bit 4 to 1
- Enabling clocks
    - Enabling the 2 clocks the CPSW relies on
    - Set bit 0-1 to 0x2 in CM_PER_CPGMAC0_CLKCTRL register
    - Wait until module is functional by reading bit 16-17 and waiting for it to equal 0
    - Set bit 0-1 to 0x2 in CM_PER_CPSW_CLKSTCTRL register
    - Wait for clocks to be active by reading bit 4 until it equals 1
- Software reset
    - Software reset each module, this is achieved by setting bit 0 to 1 in the modules reset register. Reset is done when bit 0 is read back as 0 from the same register
    - CPDMA_SOFT_RESET register
    - CSPW_SL SOFT_RESET registers (2 of them)
    - CPSW_SS SOFT_RESET register
    - CPSW_WR SOFT_RESET register
- Initialize cpdma descriptors
    - Set CPDMA descriptor registers to 0, this must be done after cpdma reset
    - Set all TX (0-7) and RX (0-7) channels Header Descriptor Pointer (HDP) registers and Completion Pointers (CP) registers to 0
    - Write 0 to TXn_HDP and TXn_CP registers (n = 0-7)
    - Write 0 to RXn_HDP and RXn_CP registers (n = 0-7)
- Configuring control registers
    - Address Lookup Engine (ALE) control register
        - Set bit 31 and 30 to 1 to enable ALE and clear ALE table
    - Management Data Input/Output (MDIO) control register
        - MDIO is how we communicate with the PHY
        - Set bit 30 to 1 to enable MDIO
        - Disable preamble by setting bit 20 to 1
        - Enable fault detection by setting bit 18 to 1
        - Set MDIO clock divisor to 124 by writing it to bits 0-15

    - Statistics control register
        - Disable statistic module by setting all bits to 0
- Read cpsw MAC address
    - MAC_ID0_LO and MAC_ID0_HI contain the cpsw MAC address
- Setting Port States and Mac Addresses
    - Set port states in PORTCTLn registers
    - Set ports 0 and 1 to forward by setting bits 0-1 to 3
    - Setting port 2 to forward causes multicast packets to not work so we do not set port 2s state to forward
    - We also set the ports MAC address to the MAC address of the cpsw by writing to Pn_SA_LO and Pn_SA_HI
- Setting up Address Lookup Engine
    - Create unicast and multicast entries in the address lookup engine table
    - One entry for our MAC address and one for broadcast
    - If an incoming packets destination MAC is not found in the ALE table then the packet is dropped
    - Entries are created by adding each word of the entry (3 words) in the corresponding TBLWn registers
    - Then writing the index in the table to bits 0-9 and 1 to bit 31 in the TBLCTL register will cause the entry to be written to the table
    - The main things to add in the entry are the entry type (unicast or multicast) and the MAC address
- Setup CPDMA queues
    - Setup the RX CPDMA queue in CPPI ram
    - The queue consists of a specific linked list like data structure shown below
    - We setup a queue of these descriptors in CPPI ram with each descriptor pointing to the next
    - We also kmalloc a buffer for the descriptor, set buffer length and set the ownership flag, the cpdma control clears this flag once it has written a packet to this descriptor buffer
    - The buffer size is 1520 bytes the max size of an ethernet frame
    - We do the same for TX, however the queue only has one descriptor and we allocate the buffer at transmission time

Figure 14-8. Rx Buffer Descriptor Format



- Enable cpdma controller
    - We enable the cpdma controller by writing 1 to bit 0 of TX_CONTROL and RX_CONTROL registers
- Start reception
    - To start reception we write the address of the start of the descriptor queue we setup to the RX0_HDP register
    - CPDMA controller will use this queue for writing packets to memory

CPSW Reception

The way packet reception is intended to happen is when the cpdma controller is done writing a packet it signals an interrupt then packet is handled via interrupt service routine. Due to a silicon errata interrupts are masked and do not get signaled to the CPU however if you read the INTSTATUS_RAW register for RX you can see when an interrupt is supposed to be signaled but is masked. With this discovery we were able to bypass the silicon errata and when cpsw_recv() is called it polls the INTSTATUS_RAW register to see if a packet has been received, we call cpsw_recv() on every timer interrupt so that packet reception is responsive.

cpsw_recv()

To receive packets we call this function on every timer interrupt, the function reads the RX_INT_STATUS_RAW register to see if packets have been written to memory by the cpdma controller. If there are packets it will start proccesing all descriptors in the RX queue starting from the channel's free descriptor that has the ownership flag unset (meaning the cpdma controller is done using the descriptor). A descriptor’s buffer that contains the packet data is passed into the process_packet() function alongside the packet size. After process_packet() returns we reset the descriptor flags, write the address of the descriptor to RX0_CP and move on to the next. If we reach the end of the RX queue the descriptor's End Of Queue flag will be set, after processing the final descriptor in the queue we rewrite the start of the descriptor chain's address to the RX0_HDP register and set the channels free descriptor to the start of queue to refresh packet processing. After descriptor processing ends we write to CPDMA_EOI_VECTOR register to clear the RX interrupt.

CPSW Transmission

To transmit a packet we write the TX descriptor queue address to the TX0_HDP register, before doing so we write the packet's address to the buffer pointer, set Start of Packet flag, End of Packet flag, and set the packet's length. After transmission an interrupt is supposed to occur however as previously mentioned the interrupt is masked so we poll INTSTATUS_RAW register for TX to see when transmission is done.

cpsw_transmit()

This function is called when we want to transmit our constructed packet, it takes a pointer to the buffer containing the packet and the size of the packet. The packet's pointer is written to our single TX descriptor in the TX queue and the descriptor's flags are set. We then write the descriptor address to the TX0_HDP register to start transmission. After we start transmission we poll the TX_INT_STATUS_RAW register as an interrupt is supposed to be signaled when TX DMA operation is completed. Once

transmission has been completed we write the descriptor's address to TX0_CP and write to CPDMA_EOI_VECTOR register to clear the TX interrupt.

PHY

The PHY is responsible for communicating with the physical transmission medium and auto negotiating the link parameters between 2 machines connected by a medium. Packets are Transferred from the CPSW to the MII then to the PHY for transmission over the medium.

PHY Driver

The driver for the PHY resets the PHY, auto negotiates link parameters then sets the link parameters. Due to a hardware bug the PHY doesn't power up on board startup and needs to be reset, the PHY can be reset using GPIO pin 8. This information is not listed anywhere and was found looking at the linux device trees for the beaglebone black.

```
ethphy0: ethernet-phy@0 {
        reg = <0>;
        /* Support GPIO reset on revision C3 boards */
        reset-gpios = <&gpio1 8 GPIO_ACTIVE_LOW>;
        reset-assert-us = <300>;
        reset-deassert-us = <50000>;
};
```

After resetting the PHY we need to autonegotiate link parameters, all communication with the PHY is done using the MDIO. We start Auto Negotiating by writing to the enable bit in the PHYs BCR register, after auto negotiation has been completed we read the PHYs Partner Capabilities register to check what link parameters we can use. We then set the link parameters based on the partners capabilities by writing to PORT1_MACCONTROL register.

Note on Packet Processing

For transmission of packets, the memory for the entire packet is allocated at the start of the stack. The packet is passed down the stack and each protocol's transmit function sets the bytes for its respective header byte by byte. For reception of a packet, the packet is passed up the stack and each protocol's recv function extracts the protocol header byte by byte. This design choice was made due to the Beaglebone Black not allowing unaligned memory access, which went against my initial plan of casting parts of the packet to a struct of the protocol header. Currently need to include destination mac in all transmit functions, would need an ARP table to be able to abstract this.

Ethernet II

eth_transmit(uint8_t* frame, int size, uint8_t* dest, uint16_t type)

- frame
  - Pointer to allocated memory for frame with upper layers headers set
- size
  - Size of frame
- dest
  - Destination MAC address
- type
  - Packet/Frame type (ARP or IPV4)
- Description
  - Sets the Ethernet II header for the frame and sends it to cpsw_transmit() for transmission
  - Sets the Destination MAC to dest and Source MAC to the Beaglebone Blacks MAC address
  - Sets the Frame type to the passed in value of ARP or IPV4

eth_recv(uint32_t* frame, int size)

- frame
  - Pointer to received frame from cpsw_recv()
- size
  - Size of entire frame
- Description
  - Extracts Ethernet II header from frame and stores it in ethernet frame header struct
  - Checks Packet/Frame type (ARP or IPV4)
  - Subtracts the size of ethernet header from frame size
  - Passes the rest of the frame (not including ethernet header) and ethernet header struct up the stack to the respective protocol recv function

Address Resolution Protocol

arp_transmit(uint8_t* frame, int size, uint8_t* dest_mac, uint8_t* dether_mac, uint32_t dest_ip, uint16_t opcode)

- frame
  - Pointer to allocated memory for frame with upper layers headers set
- size
  - Size of frame
- dest_mac
  - Destination MAC address for ARP header
- dether_mac
  - Destination MAC for Ethernet II layer
- dest_ip
  - Destination IP for ARP header
- opcode
  - ARP header OP code (Request or Reply)

- Description
    - Sets the ARP header for the frame and sends it to eth_transmit()

arp_recv(ethernet_header frame_header, uint32_t* frame, int size)

- frame_header
    - Ethernet II header struct
- frame
    - Pointer to frame from ARP header and onwards
- size
    - Size of frame from ARP header and onwards
- Description
    - Extracts ARP header from frame and checks ARP opcode
    - If the opcode is an ARP request call arp_reply() and pass the extracted ARP header
    - ARP replies aren't supported (would store the replies in an ARP table)

arp_reply(arp_header arp_request)

- arp_request
    - Struct of ARP header
- Description
    - Checks if the arp request is for our IP
    - If it is for our IP, uses arp_transmit() to transmit an arp reply

arp_garp()

- Description
    - Abstraction function that uses arp_transmit() to transmit an ARP gratuitous packet

arp_anounce()

- Description
    - Abstraction function that uses arp_transmit() to transmit and ARP broadcast packet

Internet Protocol Version 4

ipv4_transmit(uint8_t* frame, uint16_t size, uint8_t protocol, uint32_t dest_ip, uint8_t* dest_mac)

- frame
    - Pointer to allocated memory for frame with upper layers headers set
- size
    - Size of frame
- protocol
    - Ipv4 protocol (ICMP, UDP, TCP) TCP not supported
- dest_ip
    - Destination ip for IPV4 header
- dest_mac
    - Destination MAC for Ethernet II header
- Description
    - Sets IPV4 header for the packet and sends it to eth_transmit() alongside destination mac

ipv4_recv(ethernet_header frame_header, uint32_t* frame, int size, uint8_t* frame_ptr)

- frame_header
  - Ethernet II header struct
- frame
  - Pointer to frame from IPV4 header and onwards
- size
  - Size of frame from IPV4 header and onwards
- frame_ptr
  - Pointer to start of full frame, used for very specific case when we re use the buffer of the received frame for transmission (ICMP echo reply)
- Description
  - Extracts IPV4 header from frame
  - Checks the protocol type (UDP or ICMP) and calls respective protocol recv function

ipv4_checksum(uint8_t* ipv4_header, int size)

- ipv4_header
  - Pointer to ipv4 header
- size
  - Size of header
- Description
  - Computes ipv4 checksum following RFC1071
- Return
  - uint16_t checksum

Internet Control Message Protocol

icmp_transmit(uint8_t* frame, int size, uint8_t type, uint8_t code, uint32_t data, uint32_t dest_ip, uint8_t* dest_mac)

- frame
  - Pointer to allocated memory for frame with upper layers headers set
- size
  - Size of frame
- type
  - Packet type (echo reply or echo request)
- code
  - Packet code (echo reply code or echo request code)
- data
  - Icmp header data field
- dest_ip
  - Packet destination ip
- dest_mac
  - Destination mac for Ethernet II header
- Description
  - Sets ICMP header for packet and sends it to ipv4_transmit()

    - Uses ipv4_checksum() with icmp header to compute icmp header checksum

icmp_recv(ethernet_header eth_header, ipv4_header ip_header, uint32_t* frame, int size, uint8_t* frame_ptr)

- frame_header
    - Ethernet II header struct
- ip_header
    - IPv4 header struct
- frame
    - Pointer to frame from ICMP header and onwards
- size
    - Size of frame from ICMP header and onwards
- frame_ptr
    - Pointer to start of full frame, used for very specific case when we re use the buffer of the received frame for transmission (ICMP echo reply)
- Description
    - Extracts ICMP header from frame
    - Checks packet type if it is a ICMP echo request it call icmp_echo_reply() reusing the requests buffer to send the reply

icmp_echo_reply(ethernet_header eth_header, ipv4_header ip_header, icmp_header icmp ,uint8_t* frame, int size)

- frame_header
    - Ethernet II header struct
- ip_header
    - IPv4 header struct
- icmp
    - Icmp header struct
- frame
    - Pointer to frame from ICMP header and onwards
- size
    - Size of frame from ICMP header and onwards
- Description
    - Sends an ICMP echo reply to the requester

icmp_echo_request(uint32_t ip, uint8_t* mac)

- ip
    - Destination ip
- mac
    - Destination mac of Ethernet II header
- Description
    - Sends an ICMP echo request to the specified ip
    - Abstraction that uses icmp_transmit()

User Datagram Protocol

udp_transmit(uint8_t* frame, uint16_t size, uint16_t src_port, uint16_t dest_port, uint32_t dest_ip, uint8_t* dest_mac)

- frame
    - Pointer to memory allocated for frame with Payload set
- size
    - Size of memory allocated for frame
- src_port
    - Source port for UDP header
- dest_port
    - Destination port for UDP header
- dest_ip
    - Destination ip for IPV4 header
- dest_mac
    - Destination MAC for Ethernet II header
- Description
    - Sets UDP header for packet and sends it to ipv4_transmit()
    - Computes UDP header checksum using pseudo header (calls udp_checksum())

udp_recv(ethernet_header eth_header, ipv4_header ip_header, uint32_t* frame, int size)

- eth_header
    - Ethernet II header struct
- ip_header
    - IPv4 header struct
- frame
    - Pointer to frame from UDP header and onwards
- size
    - Size of frame from UDP header and onwards
- Description
    - Extracts UDP header from frame
    - Checks if a socket is bound on the destination port of the packet
    - If socket is bound on the port it stores the payload in the sockets queue

udp_checksum(uint8_t* frame, uint32_t src_ip, uint32_t dest_ip, uint16_t size, uint16_t src_port, uint16_t dest_port)

- frame
    - Pointer to memory allocated for frame with Payload set
- size
    - Size of memory allocated for frame
- src_ip
    - Source IP for UDP pseudo header
- src_port
    - Source port for UDP pseudo header

- dest_port
    - Destination port for UDP pseudo header
- dest_ip
    - Destination ip for UDP pseudo header
- Description
    - Computes UDP checksum by creating a pseudo udp header and calling ipv4_checksum()
- Return
    - uint16_t checksum

Sockets

socket(uint32_t pid, uint8_t* dest_mac, uint8_t protocol)

- pid
    - Pid of process
- dest_mac
    - Destination MAC for Ethernet II layer
- protocol
    - Socket packet protocol (only UDP supported)
- Description
    - Finds a free socket in the socket table, sets up some of the basic struct attributes and returns socket number
- Return
    - int socket_num

socket_recv(int socket_num)

- socket_num
    - Socket number
- Description
    - Returns the pointer to first payload in sockets receive queue

socket_send(int socket_num, uint8_t* frame, int size)

- socket_num
    - Socket number
- frame
    - Pointer to frame with payload set
- size
    - Size of frame
- Description
    - Calls udp_transmit() using sockets stored attributes

Sockets API

__socket(int pid, uint8_t* gateway, uint8_t protocol)

- pid
    - Pid of process

- gateway
    - Destination MAC for Ethernet II layer
- protocol
    - Socket packet protocol (only UDP supported)
- Description
    - User system call to create a socket
- Return
    - int socket_num

__bind(int soc, socket_info *soc_info)

- soc
    - Socket number
- soc_info
    - Pointer to socket info struct
    - Contains bind port (what port you want to bind on)
- Description
    - User system call to bind socket to a port
    - Binds socket to port which allows packets to be stored in sockets receive queue

__closesocket(int soc)

- soc
    - Socket number
- Description
    - User system call to free socket

__recvfrom(int soc, uint8_t* buff)

- soc
    - Socket number
- buff
    - Buffer to copy payload into
- Description
    - User system call to copy first payload in sockets receive queue into the provided buffer
    - Returns number of bytes copied
- Return
    - int size

__sendto(int soc, uint8_t* payload, int size, socket_info *soc_info)

- soc
    - Socket_number
- payload
    - UDP payload
- size
    - Size of payload
- soc_info

    - Pointer to socket info struct that contains dest ip and dest port
- Description
    - User system call to create socket transmit request
    - Allocates memory for frame and copies payload into it
    - Creates a socket transmit request using upper half driver

Upper Half Driver

To multiplex the Network stack BuddyOS provides an upper half driver. The upper half driver uses the transmit() function to check the transmit request queue for any requests and handles them by calling the respective protocol transmit function. The types of requests that can be made to the upper half driver are socket requests and icmp echo requests. The transmit() function is called on every timer interrupt alongside cpsw_recv() to keep the network stack responsive.

Network Testing

To test the network stack a couple of tools and test programs were made/used. First is Wireshark, this was used to check that the transmitted packets fit the protocol standards and that the checksum were correct. To implement the network stack I would hex dump the received packets and compare them to wireshark, essentially reverse engineering the protocol headers using wireshark. Secondly, we used Packet Sender, a tool that allows you to send custom packets. Thirdly, we used *ping* which allows you to send ICMP echo requests and *arp -a* which allows you to see cached ARP tables. Finally, we recreated schat to be cross compatible with Windows and BuddyOS using socket abstraction functions and our own standard library system calls.

Lessons Learned

Noah:
Something that I never considered before operating systems was the significance of abstraction. When we began developing BuddyOS, there were numerous aspects that I had never considered before as they were hidden from the user. This can be applied outside of operating systems as well in how we can provide a more efficient, maintainable, and better experience for the user. Lastly, an idea that was further reinforced for me was how the team environment and culture can shape your experience, being able to have fun, discuss ideas freely, and build comradery makes a world of a difference.

Aaron:
An aspect of BuddyOS that I spent a considerable amount of time on was the design of the VFS. During the process of working on this, I realized the importance of abstraction and the need to make the VFS as loosely coupled to a specific file system implementation as possible. This approach would allow the kernel to support a wide variety of different file system types without heavily depending on a specific one. By going with a VFS design that was agnostic to the underlying file system implementation, I was able to increase the kernel's overall flexibility and scalability. This would make it easier to add new file system types to the kernel in the future, without disrupting the overall architecture. This lesson can be further extended to any software I have to design in the future and is an excellent example of why it is important to write code that is loosely coupled.

Diego:
A lesson I learned was about the complexity of doing anything modern with code. There is a lot of supporting code for anything meaningfully big being developed. I never thought about all the data types and functions given to us whenever we use programming languages. This project revealed how much we stand on the shoulders of developers before us by making us implement all those functions and data types ourselves. It provided more appreciation for all the tools and technology we get when developing software on a modern operating system.

Emily:
To step away from the code aspect of this class, my lesson learned centres more around the project management itself. Communication among team members is essential; Without communication, it's impossible to know who's doing what and the overall state of the project. It is important to have 1-2 people on top of project deadlines and requirements, so that they can fill the rest of the group in. Another thing I realized was that the concepts we learn in class are a lot different from how things are implemented. It is one thing to know how something is *supposed* to work, and another thing to actually *make* it work. Knowing just the concepts will only take you so far. Hands-on implementation experience and self-guided learning is what makes the difference and bridges the gap between the two.

Marwan:
One of the biggest lessons I learned is to keep your design adaptable especially when working with hardware/embedded systems. When working on the network stack and ethernet driver I had many cases where I had to change the design due to hardware limitation or a hardware bug / silicon errata. The Beaglebone Black does not allow unaligned memory access meaning the network stack design had to be changed from casting the frames to a struct and extracting header fields from the struct to having to extract each byte manually and storing it. This plus setting specific bits in registers caused me to become a lot better at byte and bit manipulation. Initially, the design for the ethernet driver was going to be interrupt based, on packet/frame reception an interrupt would occur and an ISR would be called, however due to a hardware bug the interrupts for packet reception and transmission are masked so I had to adapt the design to overcome this issue. I ended up polling the interrupt status raw register to check if packets are received in cpsw_recv() and if there was I would essentially run the isr as the manual specifies in order to clear the interrupt. This also reinforced the idea of understanding the hardware you are working on, if I didn't study all the ethernet registers I wouldn't have known about the raw vs masked interrupt status registers.

Another lesson I learned was the importance of providing an easy to use API for other developers to use. For the SD card driver it was crucial to provide an easy to use API in the form of MMCreadblock() and MMCwriteblock(), both taking the index of what block to read/write and the buffer to read/write to. This was important as other members of the team would be using this to develop file system drivers. Also for the network stack it was important to provide a way for user programs to easily use the network stack in the form of sockets and transmit requests from the upper half driver. This allows developers to use the network stack without worrying about underlying implementation and prevents developers from using the network stack in malicious ways. To conclude, abstraction is important especially when working with other developers. Also setting up an exception handler to print exceptions is super helpful.

Alexei:
The biggest lesson I learned was the importance of proper communication with the people that will be working with your code. When working with something as complicated as an operating system, with its many quirks that must be worked around, it's necessary to ensure that these quirks are documented and communicated to others. Often when someone else needed to create something that involved my own code, I would need to either watch over carefully or even take over completely in order to prevent them from making a mistake that only I would be able to recognize. If I put more effort into making these oddities of BuddyOS easier to understand, it would have been significantly easier for others to work alongside me, allowing them to finish faster, and allowing myself to move on to another feature. This connects with the overarching lesson we all seemed to learn, that being learning how to work with a larger team effectively. It was great being given an opportunity to work with other people just as passionate about BuddyOS as I was, and really taught me how to collaborate as a team and make something a lot greater than I could ever do alone without getting into each other's way. Initially, there was a lot of stepping on each other's toes, but by the end we all got into a rhythm that was not only incredible at developing together, but was also just a great deal of fun.

Abdullah:
One of the biggest lessons I was taught throughout the development of BuddyOS was that there truly are so many layers of complexity beneath what we usually take for granted in modern Operating Systems. While working closely on parts like scheduling and context switching, it made me realise how something that may seem simple in a specific environment like context switching in QEMU, or how straightforward a round-robin design may seem, there are so many layers of complexity when trying to get it to work on a real system. Beyond just the technicalities, the project also made me realise the importance of planning ahead and writing robust code that can handle the quirks of working on hardware, or integrating with IPC and system calls. While working on the project, we had to rethink and redesign a lot of components as there would be new constraints or bugs that would surface when combining modules together. This made it clear to me that a good system design isn't about perfect design or getting everything right the first time, it's about making it flexible; capable of adapting and growing. On a personal level, I also learned about how important proper collaboration is in a team project like this. Each part of the OS connects with the other parts, so writing down ideas early, documenting our later decisions, and just discussing on the go made a huge difference. Overall, the experience taught me not only about the internals of an operating system, but also about how to approach large, low-level, collaborative projects with patience and a willingness to learn with each struggle that we would go through.

Kevin:
Implementing the memory system for BuddyOS helped in learning many new things as well as reinforcing concepts from class. The major things to keep in mind when implementing something to interface with memory were:

- Memory alignment
- Reliability
- Speed & Memory efficiency
- Overall design and planning

Going over the points, memory misalignment was the cause of the majority of bugs in the early period of implementation due to small mistakes made when offsetting the memory address. A simple +1 was the

difference between a working implementation and an immediate data abort upon entering the kernel. As a callback to class, using powers of two for all block sizes made alignment significantly easier, as well as increasing speed and efficiency with the use of bitwise operations. With this use of powers of two, we were also able to effectively remove any hard coding of values outside of the defined macros, as offsets could be easily calculated using the user defined information. This all tied back to the design and planning aspect; considering all these factors beforehand made the actual coding process significantly easier, which is especially important in the early stages of development where we lacked proper debug tools.