The Guide To Operating Systems Programming

Nicholas Walker

Nicholas.Walker@Warwick.ac.uk

Department of Computer Science

University of Warwick

2022

Table of Contents

Chapter 1

Introduction to PunchOS

1.1 Purpose

PunchOS and the supporting lab guide intends to provide the support students need to bridge the gap between low level OS programming, and the abstract concepts taught in OS modules and textbooks. This guide will work alongside the content taught in lectures, providing the experience of being able to work on a strong prebuilt framework to implement what has just been learnt.

1.2 Using PunchOS

To start, clone the repository into a directory of your choice using:

```
git clone https://github.com/NickWalker1/PunchOS
```

From here, `git checkout LabX` will move you to a lab `X`.

1.2.1 DCS

Running PunchOS on the DCS is very simple due to the use of make. It's usage is as follows.

When root directory of PunchOS:

```
$ make
```

Compiles and links all PunchOS files together into a single *os.iso*

```
$ make run
```

Runs the *os.iso* produced. If *os.iso* does not exist it will compile it first then run on success.

```
$ make clean
```
Deletes any object files produced to ensure a clean start

1.2.2 Linux

PunchOS requires the use of several dependencies outlined below:

- GCC
- Grub2
- Xorriso
- QEMU-system-x86
- NASM

If you are using the apt package manager, they can be installed with the following command:
```
$ sudo apt install gcc make grub2 xorriso qemu-system-x86 nasm
```
Otherwise, you must modify this appropriately.

PunchOS then requires the modification of the Makefile as the default is setup for DCS. To do this, comment and uncomment out the appropriate QEMU and GRUB variables for the DCS and Local instances respectively.

1.2.3 Other

PunchOS does not currently support operating systems other than Linux and has not been tested on them either. Should you get PunchOS working on a different system please contact me directly.

1.3 Notes

Standard notation: `i_am_a_function()` in file *somefile.c*.

Chapter 2

Lab 1 - Getting Started

The content of this lab will not line up with the content taught in lectures; instead, it will serve as an elementary introduction to OS development and to quickly ensure your C skills are up to scratch.

2.1 Introduction

2.1.1 Virtual Memory

Virtual memory concepts aren't taught until later in the module, so for now, you will need a basic understanding to get started with these lab exercises.

The memory which we will concern ourselves with for now is the memory in RAM. This memory can be thought of as an array of bytes, each with an associated address assigned linearly from 0 to 4GiB (max 32-bit addressable range is 2^32); these are known as the physical addresses. However, in reality, there isn't only RAM, and we would also like to separate address spaces both for logical and security reasons, hence the use of virtual addresses. Throughout your university career, the likelihood is that you have only dealt with virtual addresses. When using a virtual address, its value is converted to physical addresses through a mapping system computed by hardware in the CPU. The two main approaches to map virtual to physical addresses are segmentation and paging, these will be covered in detail later on, but for now, in this preliminary OS, only segmentation is used as it is the default system when the CPU first boots.

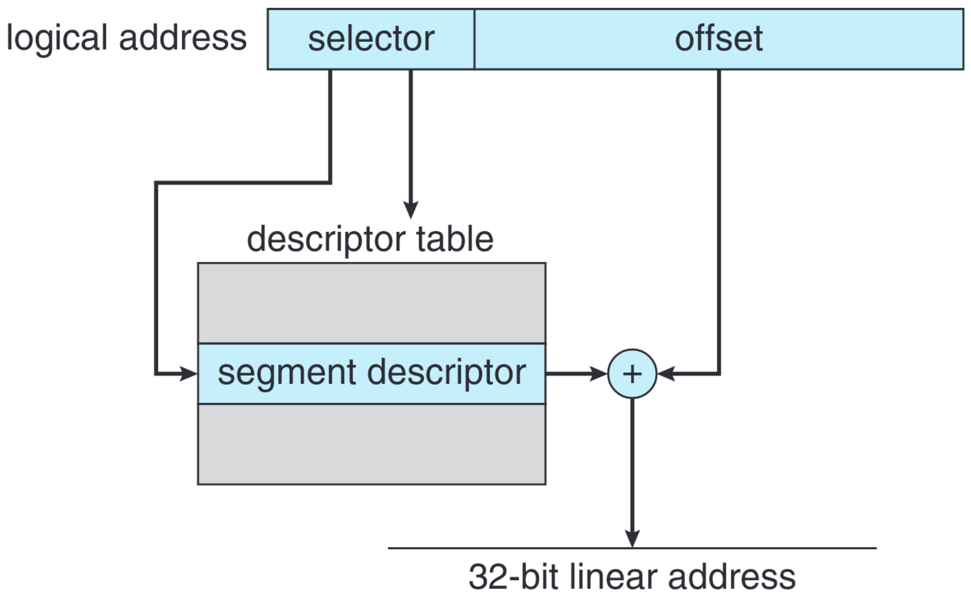


Figure 1: IA-32 Segmentation[1]. Logical and Linear addresses refer to virtual and physical addresses respectively.

Segmentation

Segmentation is the idea of breaking a virtual address space into variable length segments. Each virtual address is broken up into a segment number and an offset, as seen in Figure 1. The calculation into the physical address is performed by the CPU by using a series of segment selection registers which we will not concern ourselves with. For now however, we will just assume the entire code base is in one segment, meaning the entire virtual address is simply the offset. If our segment starts at 0(which it does) it creates an identity mapping from virtual to physical addresses for simplicity.

2.1.2 Memory Address Spaces

The information regarding which memory addresses that are free for us to use are usefully provided by a data structure produced by GRUB (GRand Unified Bootloader). There already exists code such as the function:

```
bool validMemory(uint32_t addr);
```

That parses the information provided by grub to ensure that the assumptions made about memory availability are correct. You need not concern yourself with this, just know it exists and is an important part of the process.

2.1.3 Data Types

When programming an operating system, standard libraries programmers have grown to expect are no longer available due to a potential lack or difference in the functionality available on which common standard library functions rely. Therefore this new OS must contain a bespoke standard library replacement instead. Even seemingly basic data-types may be packaged with systems requiring unavailable functionality; hence for simplicity, many commonly used data-types have been separated and included in a `typedefs.h` header file for your use.

2.2 Code Tour

Use `$ git status` to ensure you are in the Lab1 branch, if not, use:
`$ git checkout Lab1` to move over.

PunchOS requires multiple different sections to come together to be built into an executable binary file that can be found by QEMU and run. The folders, key files and their uses are described in Table 1.

Folder	**File**	**Contents**
isodir/	-	This is used by GRUB to help produce the bootloader and is of no concern to the programmer unless you wish to change grub boot configuration options, but this is not recommended.
/	*link.ld*	It is used by the linker to place code and label offsets at the correct virtual addresses. For example, GRUB by default loads our kernel code at 1MiB+. Therefore when we build the OS the code needs to know it's loaded at 1MiB, otherwise it will assume another address and erroneously jump there and break.Feel free to look inside this file for more information, but for now it is not necessary.
	Makefile	Configuration file for make command. It defines which files are necessary and how to compile, combine and run them.
boot/	*boot.asm* *gdt.asm*	The code that is jumped to when GRUB finishes. It will put the CPU into a safe state, set up and enable the initial paging system, then jump to C code.
src/kernel/	*kernel.c* *kernel.h*	Primary kernel code that is jumped to from boot code.
	multiboot.h	Header file to help with boot process validation.

Table 1 continued from previous page

	heap.c *heap.h*	Where the majority of your Lab 1 code will be written.
	heap_test.c *heap_test.h*	Heap testing functions.
src/drivers/	*low_level.c* *low_level.h*	Driver code to help the OS interact with hardware.
src/lib/	*debug.c* *debug.h*	Provides tools for debugging code such as PANIC, ASSERT and breakpoint.
	int.c *int.h*	Used for integer and ASCII conversions.
	math.c *math.h*	Basic Math functionality such as pow.
	screen.c *screen.c*	API for printing to the screen.
	string.c *string.h*	Provides some basic standard library string functions such as memcpy, strcpy and memset.
	typedefs.h	Common definitions for data types and NULL.

Table 1: Lab 1 Code Tour

2.3 Exercises

This lab involves creating the functionality necessary for allocating dynamic memory on the heap.

All code to be modified is in *src/kernel/heap.c* or *src/kernel/heap.h.*

The testing functionality is available in *src/kernel/heap_test.c* for reference.

2.3.1 Exercise 1 - Data Structure

Memory space can be logically broken down into 3 main sections: the stack, static memory and the heap. The stack you should already be familiar with, this is the place where the running process will store all their required temporary data including local variables or arguments to function calls. Static memory is used for storing global variables; information that is not lost between changes of scope and persists for the life of the program. Finally, the heap is the place for allocating dynamic memory. Dynamic memory in C is entirely managed by the programmer, and is manipulated through functions such as malloc and free.

...	Header	Data	Header	Data	...

Figure 2: Heap Diagram

When the `malloc(int b)` function is called, the OS must return back to the programmer the base address of a region of size `b` bytes. To do this, it must allocate a segment of the heap that is free to be used. The management of segments is the job of this underlying data structure to be implemented in this first exercise.

Each segment must contain some data defining its status, its size and an ability to traverse to the following or previous segments. This can be naively implemented using a header structure as seen in 2. A header can be thought of as a stamp placed onto a memory region that defines all the information for the following segment. As this stamp is always of the same format, it can be defined using a C struct with the following fields:

- A `uint32_t` definining the size of the following data.
- A `bool` noting the segments status as free or used.
- Two `MemorySegmentHeader_t` pointers, one for the next and previous segments.

The empty outline of this `MemorySegmentHeader` struct can be found in *src/kernel/heap.h*.

Task 1: Fill in the required fields of the `MemorySegmentHeader` struct.

PunchOS provides you with 4KiB to use as your heap; the base and limit addresses are stored in: `KHEAP_ADDR` and `KHEAP_ADDR_MAX` respectively. With this, you can then 'stamp' the required headers directly onto memory as needed. Note the difference between this approach to more common C use cases with system calls. We have complete control of memory; there is no malloc here! This stamping process involves setting a struct pointer to point to the region of chosen memory, then filling in the fields of the struct will fill out the memory as expected.

To initialise this heap, there must be at least one segment created. This segment can be set as free, and the size will cover the entire remaining range of the heap, minus the header size of course. In addition, to improve the usability of the heap, a global variable `firstSegment` can be used that will point to the first segment header.

Task 2: Implement the `void intialiseHeap(void* base, void* limit)` function that will initialise the 4KiB address space given. This includes creating the global variable `firstSegement` and initialising it along with the heap.

With this, you are now ready to implement the `malloc` function.

2.3.2 Exercise 2 - Malloc

There are many methods to implement `malloc`, the method given here involves a ”first fit” approach due to its simplicity. When `malloc` is called it must traverse the heap until it finds a segment that satisfies its requirements. Once found, it can then adjust the headers, perform the necessary updates to the heap, and return the base address of the data to the caller.

The explicit **logic** for `malloc` is as follows:

1. Traverse the linked list from the start until landing on a memory segment that is both big enough for the size of data requested; and is free. If not, skip. If the list has been traversed and no segment is big enough, return a `NULL` pointer to indicate failure.
2. If an appropriate segment has been found, update the fields header struct at the start of this segment with the relevant information.
   - 2.1. If there is no next segment or the next segment is taken, add a new header at the end of the current segment containing the information for the remaining free memory, and update all necessary doubly linked list pointers. Remember, there is no guarantee the remaining memory is big enough for a new header.
   - 2.2. If the following segment is free, you must find a way to “move” that header to the new end of your current segment and update all necessary doubly linked list pointers.
3. In each block ensure to return the address of the start of the allocated memory (NOT the start of the header).

Task 3: Use the logic above to implement the `void *malloc(uint32_t size)` function. Run PunchOS to test your solution. You should naturally expect the `free` related tests to fail.

2.3.3 Exercise 3 - Free

As the good programmers we are, we will need an implementation of `free` to manage memory effectively.

This function will mark the associated memory segment for an address as free, and if possible merge consecutive free blocks together into larger ones to avoid fragmentation.

Here you may notice many ways to implement free, each with its own implementation semantics. For example, in one solution, a pointer to anywhere in a segment can identify the segment; in others, it must be the base pointer. Design decisions like these can drastically affect usage, and one solution may be no better than another; the importance of the distinction lies in the quality of the documentation.

Feel free to this implement as you wish, just ensure to document clearly for future reference any assumptions you may make and that the solution still passes the tests.

Task 4: Implement: `void free(void *addr);`.

2.3.4 Extension - Performance Improvements

The best-case run time of the `malloc` function implemented is currently $O(n)$, this isn't great. To quickly improve performance, add a pointer to the first free memory segment to reduce the best-case time complexity to $O(1)$, remember to add the appropriate logic in the rest of the function and in `free()` also.

The next task is to improve the computational efficiency further by updating the header struct to contain pointers for a quadruply linked list, as to now include pointers to the previous FREE segment and the next FREE segment also.

Rerun the tests to ensure the complex pointer arithmetic is correct!

Chapter 3

Lab 2 - Processes

Use `$ git checkout Lab2` to move into this new lab. You may need to commit your previous changes to do this. Alternatively clone a new copy with:

```
$ git clone --branch Lab2 https://github.com/NickWalker1/PunchOS
```

3.1 Introduction

3.1.1 PunchOS Processes

Address Space

As you should be aware, one can define CPU operation can as being in one of two modes, user(ring 3) or kernel(ring 0) in x86. This is known as Dual-Mode operation. Operating system code operates in a high privilege kernel environment with few limitations. In contrast, user code runs in a low privilege localised user environment with many protections and restrictions. When a user program requires something from the kernel, such as more memory, it must request it through a predefined API called a system call. System calls are, in effect, glorified interrupts. This approach is highly robust and thoroughly recommended for OS design.

However, PunchOS instead implements a hybrid approach between kernel only processes and a complete Dual-Mode system. The primary reason is to simplify the code base and make it easier for students to debug their solutions. PunchOS processes, as far as the CPU is concerned, are all kernel (ring 0) processes. However, the hybrid element allows for process virtual address space independence whilst managing multiprocessing and other complex OS systems without system calls.

PunchOS uses an OS design principle called a higher half kernel to implement this hybrid approach. A higher half kernel involves mapping kernel data and code sections to a high virtual address space (3GiB+) reserved for kernel operation. With Dual-Mode operation, user processes never have kernel mappings, hence are never able to execute kernel code. This hybrid approach will have no such protection. All processes are kernel processes with the appropriate kernel mappings. However, each process has independent virtual address mappings below the kernel, thus creating this hybrid approach.

Process Control Block

Figure 3 shows the structure of a standard process control block. This PCB shares many similarities with PunchOS PCBs, though with some key differences. The primary difference is that a PunchOS PCB always makes up exactly 4096 bytes (for reasons that will become apparent later). Hence the stack grows from max downwards towards the PCB data - causing the second difference. Due to the size of the PCB, the heap is stored elsewhere in memory; this is so it can be much larger and reduce the chance of challenging bugs to diagnose if the stack grows into the heap. The final difference is that the text section also doesn't exist. The text section is the place for the executable user code, but as the executable code is the standard kernel code, no text section is necessary.

Context Switching

Context switching is the process of switching between running processes. PunchOS employs a preemptive scheduler, meaning processes are preempted when their allotted CPU time slice runs out, and they're replaced by the next waiting process. To preempt a process, there must be an external interrupt informing the CPU of the passing of time. This is all handled by the interrupt handlers so is of no concern to you, but just know that with every tick (the `proc_tick()` function being called), 1/18th of a second has passed.

Synchronisation

When preemptively context switching, a process may be interrupted before it has finished operating on shared data. To ensure that another process doesn't begin operating on the same data in a potentially undefined state, there must be the use of synchronisation primitives. All synchronisation related functionality is available in *src/sync/*. It is worth looking at the functions available, though you are not required to use them to complete the following tasks.

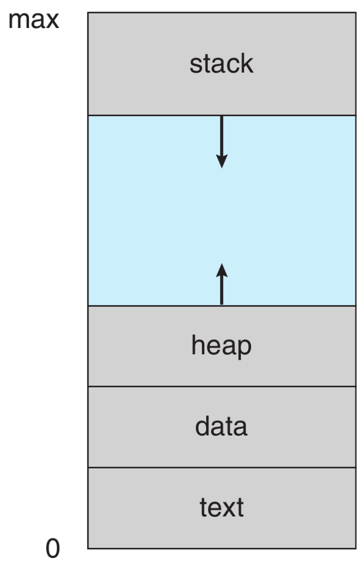


Figure 3: PCB Layout[1]

Memory

As briefly mentioned earlier, PunchOS processes share virtual memory mappings above 3GiB which defines the kernel address space. Below this address the virtual addresses are independent per process.

To allow processes to communicate for critical systems such as memory management or scheduling, the processes must make use of the shared virtual address region. In this region there is a predefined 32KiB block of memory initialised as a heap space shared between all processes. Processes are then able to use the `shr_malloc` and `shr\_free` functions to interact with this heap. Both functions already make use of the necessary synchronisation primitives for safety.

3.2 Code Tour

Folder	**File**	**Contents**
src/interrupt	*cpu_state.c* *cpu_state.h*	Used to manage the CPU state on interrupts and exceptions for context switching and debugging purposes.
	handlers.c *handlers.h*	Defines default exception handler functions.
	idt.c idt.h	Primary interrupt code. IDT setup here.
src/lib	*list.c* *list.h*	PunchOS implementation of simple doubly linked list. Uses heap to function so the memory system must be initialised before usage.
src/processes	*mq.c* *mq.h*	Message queue files that you will be implementing in this lab.
	pcb.c *pcb.h*	PCB control code and struct definitions
	process.c *process.h*	Primary processing code. Look at this in detail.
	pswap.asm	Context switching assembly code for processes.
src/shell/functions	*ps.c* *ps.h*	Basic implementation of the process status function. Used for checking and debugging processes.
src/sync	*sync.c* *sync.h*	Code for implementation of key synchronisation primitives such as semaphores, locks and conditions.
src/test/	*mq_test.c* *mq_test.h*	Test script for message queue processing, debugging and validation.

Table 2: Lab 2 Code Tour

3.3 Exercises

3.3.1 Exercise 1 - Spinning Bars

This first exercise is designed to help you become familiar with the new code base and some of the functionality available to you that will be useful in later exercises. It will also serve to demonstrate the capabilities of concurrent execution.

In the *kernel/kernel.c* file, main multi-processing code begins in the `main` function; anything before this is part of the OS bootstrap phase and should not be altered. This main function is run by the 'init' process and is free to be adapted to your choosing.

Once completed, the `void spinning_bars()` function will create 10 processes, each running the `spin(int offset)` function. The spin function will print a character to the given screen offset and sleep for 4 ticks, it will then print the next character in the sequence and repeat to give the illusion of rotation.

Processes can be created with the function:
`PCB_t* create_proc(char *name, proc_func *func, void *aux)`
The name argument is for debugging purposes, so feel free to leave it as `NULL`. The second argument `func` is the address of the function this process shall run. The final argument `aux` is the argument to the running function. Note: `aux` is a void pointer, but you may cast this inside the running function to whatever type is required!

Example usage: there is some function call you want to a new process to run concurrently instead: `work(5);` turns into: `create_proc(NULL,work,5);`

Task 1: Implement the `spinning_bars()` function. To generate the `spin` offset, you may use `rand()` to generate a random integer between 0 and $2^{15} - 1$, remember to mod this value by `SCREENSIZE`, otherwise it'll be writing to random memory locations!

Your solution may look something like Figure 4.

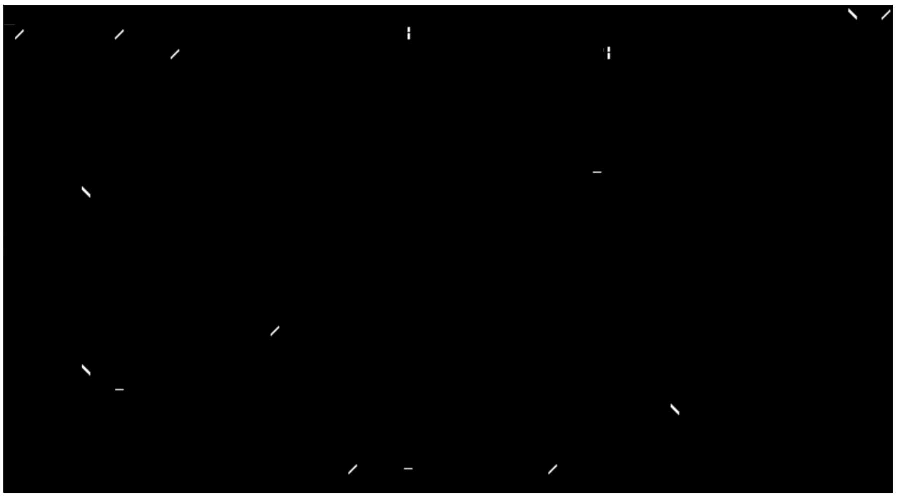


Figure 4: Spinning Bars Example

3.3.2 Exercise 2 - Inter-process Communication

NOTE: This exercise will make extensive use of lists. An explanation of PunchOS lists and their usage can be found in the tools section (Section 6). Please familiarise yourself with them if you haven't already.

All **files** relating to this exercise can be found in *processes/mq.c, mq.h* and *tests/mq_test.c.*

For this second exercise you will be creating a circular message queue implementation using the POSIX function calls as a more advanced example to base your code around.

To do this, you will need to have functionality to both create a message queue; open a preexisting message queue; close a message queue; and of course send and receive messages from a message queue.

You may find it useful to remind yourself of how a circular queue operates by simulating its use of read and write indexes on paper first.

POSIX-esque Example

Listings 1,2 show an example usage of the intended functionality.

```
//-------Client----------
char *queue_name = "test_queue";
char *send_message = "Hello, world!";
int main(){
    mqd_t *queue = mq_open(queue_name, NULL, O_CREAT);
    if(!queue){
           printf(\Cannot open %s.", queue_name);
           return 1;
      }


      //Send to server
      if(mq_send(queue, send_message, strlen(send_message)+1, 0) ==
       ↪  -1){
         printf("Failed to send message");
         Return 1;
    }

    char *in_buffer = (char*) malloc(50);

    //Receive response from server
    if(mq_receive(queue,in_buffer, 50, NULL) == -1){
        printf("Failed to receive message");
        return 1;
    }

      printf("Received:  %s",in_buffer);
      return 0;
}
```

Listing 1: Example Message Queue Client Usage

```
//-------Server----------

char *queue_name ="test_queue";
char *response="okay!";

int main(){
    mqd_t *queue = mq_open(queue_name, O_CREAT, NULL, NULL);
    if(!queue){
        printf("Cannot open queue");
        return 1;
    char *in_buffer=(char*) malloc(50);
    if(mq_receive(queue, in_buffer, 50, NULL) == -1){
        printf("Failed to receive message");
        return 1;
    }

    printf("Received: %s", in_buffer);

    mq_send(queue,response, strlen(response)+1,0);

    return 0;
}
```

Listing 2: Example Message Queue Server Usage

Much like lab 1, there is a significant testing environment to ensure the validity of your solution. However, as the message queue system is far more complicated than the heap system, there is far greater opportunity for differences in implementation and operational semantics. Therefore, **all requirements to pass the tests are listed below**, and are known as the **Key Points**, anything not stated is left up to you. Upon failing a test, you should use these key points to help check what you may have missed.

Key Points:

1. Each queue can be uniquely identified by a string name of length up to 12 characters.
2. Each message queue contains a pointer to an attribute struct which contains key values about the current state and setup of the queue.

3. Each queue uses a circular implementation with a read and write header value.
4. Each message block is of the same size. This size is defined in the attribute struct.
5. Send operation will fail if the message is too big for the message block or if the queue is full.
6. Receive will fail if the buffer is smaller than the maximum size of the message block or the queue is empty.
7. Send will return the number of bytes sent (0 on failure).
8. Receive will return the size of the buffer written to (0 on failure).
9. it is a circular queue with read/write header values zero indexed.
10. Total message queue size is at least 4096 bytes.

Opening and Closing a Message Queue

To begin we must first implement a method of creating a message queue.

The type of this open function is:
`mqd_t *mq_open(char *name, mq_attr_t *attr, uint8_t flags);`

This function will, when given the `O_CREAT` flag in the flags argument, return a pointer to a new message queue descriptor with the given attributes pointed to by the `attr` argument. If `attr` is NULL it uses the default attributes setup in the `mq_init` function. If the `O_OPEN` flag is given it will return the preexisting message queue descriptor pointer or `NULL` if it doesn't exist.

```c
/* Message queue descriptor struct */
typedef struct mqd{
    char name[12];
    mq_attr_t *attr;
    void *base; /* Base vaddr of the 4KiB mq block */
    size_t write_idx; /* Index of write header */
    size_t read_idx; /* Index of read header */
}mqd_t;
```

Listing 3: Message Queue Descriptor Struct

```c
/* Message queue attribute struct */
typedef struct mq_attr{
    size_t mq_flags; /* Flag variable */
    size_t mq_maxmsg; /* Max # of messages on queue */
    size_t mq_msgsize; /* Max msg size in bytes */
    size_t mq_curmsgs; /* # of messages currently in queue */
}mq_attr_t;
```

Listing 4: Message Queue Attribute Struct

The message queue and attribute structs are given to you already in *ipc.h*.

In order to create or open a message queue, there must be a shared data structure which can track the status of every message queue instance. One very simple and naive approach is to simply have a list pointing to each message queue that can be iterated through and check every name.

This list has already been initialised for you and the function:
`mqd_t *mq_list_contains(char *name);` can be used to iterate through the list, check the names and return the `mqd_t*` if possible or `NULL` on failure.

Writing mq_open:

To create the message queue descriptor struct for a new queue, it must first be allocated using the shared heap space with `void *shr_malloc(size_t size)`. This function performs identically to malloc, however the address returned will be from the shared heap space and addressable by every process.

As PunchOS uses a hybrid approach operation, allocating a block of memory to store the queue contents is greatly simplified. The use of:
`void *pallock_kern(size_t n, uint8_t flags)` can return a $4096 * n$ byte block of memory in shared kernel memory space. It will be sufficient for now to assume all message queues fit into a single 4096 byte block of memory.

Example palloc: `void *block = palloc_kern(1,0);`

To update the attribute struct, it cannot be assumed that the pointer to the descriptor given is in shared space, therefore you must allocate the space manually and copy the data over using `memcpy`. If no attribute struct is given, the default attributes can be found at `default_mq_attr`. When using the default attribute struct, ensure to again allocate new shared memory and copy the data over, otherwise, if two or more queues are using the default attributes they will clash.

Remember to set any default values in the descriptor struct, e.g: the name (you will need to use `strcpy` for this).

Finally, add this queue to the shared list of descriptors using `append_shared()`.

Task 2: Implement `mq_open`, and call `MQ_test()` to validate the partial solution. Details of the tests can be found in *tests/mq_test.c*.

Writing mq_close:

Now, the `mq_close` to close a message queue can be implemented.

The requirements for this function are simply to free any referenced memory including the main queue memory block. To do this you will need `palloc_free(void *addr)`.

As the primary struct and attribute structs are in shared memory, you will need to use the `shr_free(void *addr)` function to free that memory.

There are no tests for this function.

Task 3: Implement `mq_close`.

Writing mq_send:

The `mq_send` function is defined as follows:
`size_t mq_send(mqd_t *mqdes, char *msg_pointer, size_t msg_size);`

Implementing this function is a relatively simple process. For assistance, follow the structure below.

- Validate the message and the status of the queue based on the key points described earlier.
- Use the write header and the base address to calculate the address to write the memory to.
- Copy the data into the correct message block using `memcpy`.
- Update the queue attributes struct and the write header position, checking for wrap-arounds.

Ensure to follow any details set out in the **Key Points** above.

Task 4: Implement `mq_send`.

Writing mq_receive.

For this, you may simply copy and modify your `mq_send` functionality.

Task 5: Implement `mq_receive`. Ensure your code now passes all tests!

3.3.3 Extension - Alarm

This exercise will allow you to test your understanding of context switching and process scheduling.

Currently processes have the ability to block (sleep) for a given number of ticks before being appended to the ready queue (woken up), this means any processes currently in the queue will be executed before this function is actually scheduled. When the number of processes is large, this can cause significant scheduling latency.

The current implementation of sleep is to create a 'sleeper struct' allocated in the shared heap containing a pointer to the blocked process; and a counter of how many ticks are remaining.

Then on each tick, the `sleeper_tick()` function is called which will decrement every sleeper's remaining tick count. Then if any need to be woken up, their sleeper struct is freed and they are rescheduled.

In this you will implement the
`void proc_alarm(uint32_t time, uint8_t format);` function.

Implement this in any way you see fit. Use the `proc_sleep` and `sleep_tick` functions for logical reference.

Remember, you are free to edit any pre-existing files or functions should this suit your implementation.

Chapter 4

Lab 3 - Multithreading

Use `$ git checkout Lab3` to move into this new lab. You may need to commit your previous changes to do this. Alternatively clone a new copy with:

```
$ git clone --branch Lab3 https://github.com/NickWalker1/PunchOS
```

4.1 Introduction

4.1.1 Multi-threaded Processes

In modern systems, running processes are complex pieces of software that often involve many smaller pieces of work that can be run concurrently. To do this the idea of a mutli-threaded process is introduced. Multi-threaded processes are made up of a variable set of threads. Each of these threads therefore shares it's parent process' virtual address space providing far simpler and more efficient interoperability.

The nature of many modern applications involves a large number of concurrent threads, each with varying priorities that define how important it is that it is scheduled in a timely manner. The current approach from the previous lab involves a naive round robin system which has no notion of thread priority. In this lab you will be implementing a more advanced scheduler to improve system performance and reduce the scheduling latency of high priority threads.

As you can see the code base has been updated to allow for multi-threaded processing operations.

4.2 Code Base

Much in the way processes maintained a `PCB` struct, threads also use a `TCB`, taking many of the attributes from a `PCB` leaving only virtual address space info, process info and a list of all associated threads. Therefore, `TCB`s are the primary method to reference and interact with threads.

The rest of the multi-threading code is similarly taken from the processing code, with all scheduling and context switch operations being very similar, just under different names!

4.3 Advanced Scheduler

This lab involves the completion of a single exercise to improve the scheduling algorithm from a basic round robin to a fully fledged **Multilevel Feedback Queue(MLFQ)** implementation. MLFQ scheduling involves a series of queues of threads ready to be run, taking the thread from the highest priority queue first. The notion of feedback allows for threads to be moved between queues when certain conditions are met. These conditions may include if a thread has been waiting for too long, or it is using too great a share of the CPU resources.

It is also possible for each queue to define it's own time quantum(TQ) meaning threads in a higher priority queue may have a shorter time slice of the CPU to improve latency on key systems. This also applies for lower priorities having longer time quantums allowing for reduced wasteful context switch operations.

The **key points** of a MLFQ scheduler are as follows:

1. There are a series of queues, one for each priority level.
2. Each queue may have a different TQ.
3. In each queue, threads are scheduled in a FCFS manner.
4. When a thread is first run it starts in the highest priority queue.
5. If a thread fails to execute in the given TQ, it is moved to a lower priority queue if possible.
6. When choosing the next thread to run, the highest priority thread is always chosen first.
7. The running thread will be interrupted when a new thread is added to a

higher priority queue.

8. If a thread relinquishes control (`thread_yield`) it will be rescheduled in the same priority queue as it was before.
9. In the lowest priority queue, threads move around in a round robin fashion.
10. If a thread has been waiting for more than X[1] ticks it will be promoted to a higher priority.

Currently in PunchOS, threads are scheduled in a simple round robin queue. Every 1/18th of a second there is an external interrupt from a programmable interrupt timer(PIT). This interrupt is handled by the `thread_tick()` function as seen in Listing 5; this will first perform general chores such as updating the thread diagnostics information. It will then call the `sleep_tick()` function, which manages sleeping threads and allows them to be woken up if necessary.

It will then check if the current thread has exceeded it's time quantum, if it has it will yield control of the CPU and schedule another thread to be run. You need not concern yourselves with the details of the schedule function, just know it uses the `get_next_thread()` function to find a new thread to run and performs the context switch if necessary.

Given key point 7 and now being at the end of the function, the MLFQ implementation must also check that there hasn't been a new thread waiting at a higher priority. To do this it must 'peek' the next thread to be scheduled. If that thread has a higher priority, it will yield use of the CPU. However, with the round robin approach this is not necessary hence has been commented out.

4.4 Exercise

To simplify the code base, all scheduling related functionality has been separated into *src/threads/sched.c, sched.h.* In *sched.c* you will see the current implementation of the round robin queue.

Implementing a MLFQ scheduler involves the modifications of several key functions in the scheduling file.

`scheduling_init()` - Used to initialise any queues or data structures.

[1]X can be defined and tuned to your own desire. This could be a constant or a function of the current TQ for example.

```
/* called by PIT interrupt handler */
void thread_tick(){
     TCB_t *curr = current_thread();

    /* Timing analytics */
    timing_updates(curr);


    sleep_tick();


    /* Preemption */
    if(++cur_tick_count>= time_quantum){
        timeout=true;
        thread_yield();
    }

    /* Check for higher-priority thread waiting */
    /* UNCOMMENT FOR TESTING MLFQ */
    /*
    TCB_t *pn = peek_next_thread();
    if(pn && pn!=idle_thread && (pn->priority < curr->priority)){
        thread_yield();
    }
    */
}
```

Listing 5: thread_tick Implementation in src/threads/thread.c

`thread_reschedule()` - When a thread has been preempted for exceeding its time slice or is woken from a sleep, it must be rescheduled for future execution. This function will appropriately reschedule the thread, modifying it's priority if necessary on a preemption. As rescheduling operates differently between different cases, there is a global variable `timeout` of type `bool` that is set to `true` when the last running thread was preempted. You must remember to set this back to `false` once handled.

`get_next_thread()` - Returns a pointer to the next thread to be executed and pop it from the relevant queue. Based on the queue it has been popped from, it must update the global variable `time_quantum` appropriately. If no threads are ready to be executed, return the `idle_thread` pointer.

`peek_next_thread()` - Returns a pointer to the next thread to be executed, however, makes no changes to the queues or TQ. Used to check for preemption by higher priority thread.

`deschedule(TCB_t *t)` - Will remove a thread `t` from any ready queues. Used to ensure do not schedule a killed thread.

Finally, the `thread_tick()` function in *src/threads/thread.c* must be modified by uncommenting the preemption code necessary for the MLFQ.

The test for this scheduler can be run with the function `scheduling_test()`. This test involves running a synthetic workload, and will compare the running time to that of the naive scheduler on the DCS. Your solution will pass the tests if it simply runs faster than the naive scheduler. This however, may not work when operating on personal hardware, so feel free to modify the testing solutions and timing the naive scheduler first.

Task: Implement the MLFQ scheduler by modifying the above mentioned functions. Use `scheduling_test()` to test your solution.

Chapter 5

Lab 4 - Paging

Use `$ git checkout Lab4` to move into this new lab. You may need to commit your previous changes to do this. Alternatively clone a new copy with:

```
$ git clone --branch Lab4 https://github.com/NickWalker1/PunchOS
```

5.1 Introduction

Paging is the primary method of translating virtual addresses into physical addresses. All memory, virtual or physical, is broken down into chunks of either 4KiB (or 4MiB depending on the system) called pages in the virtual context, and frames in the physical. The aim of the paging algorithm is to efficiently translate virtual page addresses into physical frame addresses. Implementing a paging system is entirely optional, but one of the main reasons for it is to allow the separation between different process virtual address spaces; how exactly it does this and what that means will become clear later.

In a 32-bit addressing system, 4GiB is the maximum addressable value ($2^{32} =$ 4GiB), and paging by itself, does not allow addressing beyond this range, but using it intelligently allows you to use external storage devices far beyond this range (this also will be explained later).

In order to translate a virtual into physical address, the virtual address is broken up into two page table indexes and an offset (Figure 5). If the page size is 4KiB then the required offset is 12 bits ($2^{12} = 4096$) so to be able to fully index every byte in the page. Assuming 32-bit addressing, the remaining 20 bits can be used to index the tables.

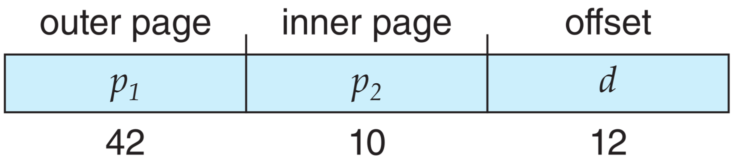


Figure 5: Virtual Address Structure with 2-level Paging [1]

Page tables are themselves pages which contain the translation to the next stage of conversion. Assuming 4KiB pages, each page table will store 1024 4 byte entries. Each entry will firstly contain the 20 most significant bits of the physical address of the next page (20 bits is sufficient to address a 4KiB aligned address as the remaining 12 bits would be 0 anyway), and the remaining 12 bits are used to store information about that page. Multi-level page tables are required to fully address the full 4GiB address space. As each table, assuming each entry points to one page can point to exactly 4Mib ($1024 * 4$KiB), therefore to address 4GiB another page table is required. You can see that if you are using 4MiB pages, then only one page table is required.

Using Figure 6, the first page table known as the page directory is indexed by the first 10 bits of the virtual address. In this 4 byte entry the first 20 bits are used to point to the bottom of the next page table. Those 20 bits must be shifted to the left by 12 bits to be a full 32-bit physical address. The process is repeated again to point to the base address of the intended physical frame. On a 64-bit system, 4 page tables are required to address the full space, and this in turn has its own efficiency drawbacks as to access a new frame requires a 4 stage lookup process.

For processes to have their own virtual address spaces, one can simply swap out the current page directory in use on a context switch, thus updating all virtual address mappings to those of the new process.

With all these pages, the kernel must track both which physical pages have been allocated and to whom(process). It is then either the job of the process or the kernel to track which virtual address are being used by that process, to ensure no overlaps. It is possible, and often common, to have multiple virtual addresses pointing to the same physical address. PunchOS manages this with a global physical pool, and virtual pools per process.

PunchOS has been using this 2-level paging system throughout the last 2 labs, and this lab will involve updating and improving it for added functionality crucial for lab 5.

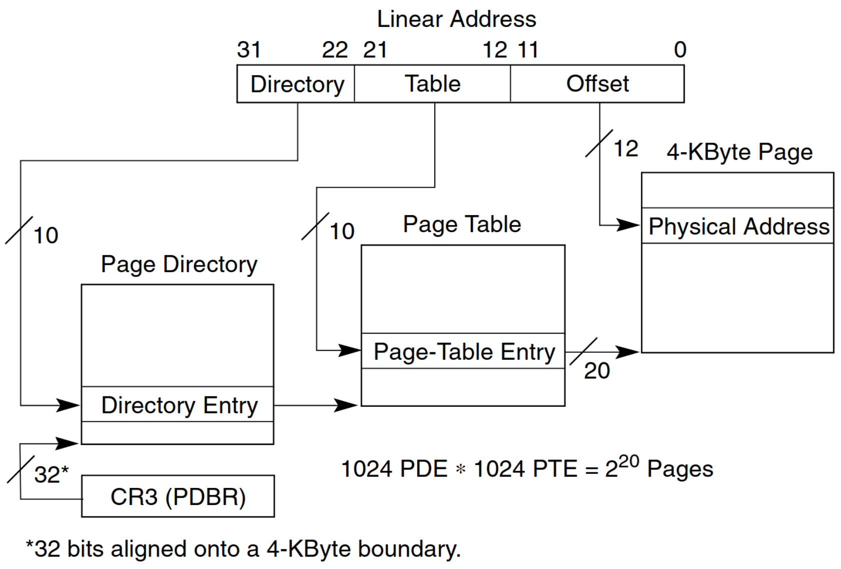


Figure 6: IA-32 Virtual to Physical Address Translation [2]

5.2 Page Replacement Algorithm

Memory must be swapped between RAM and secondary storage when a process references an address that is not currently in RAM (a page fault); this can be because RAM is full (capacity) and the page has been swapped out (conflict), or it has not yet been referenced (compulsory). In the case that RAM is full, it makes use of an overflow area called a swap file on secondary storage to be able to move a page out, thus allowing the OS to continue operating despite this “fault”.

Page replacement algorithms provide this functionality. They, in the most simple sense, function on a page fault, and are able to load the required page into RAM, potentially moving another one out to make space.

In this lab you will be implementing a **global demand page replacement algorithm**.

- **Demand**: only when a page is requested will it be loaded and never before.
- **Global**: When a page is removed from RAM it can be from any process's memory rather than the current (local) process.

To be able to perform this swap, the OS must be aware of what data the process is requesting, despite it not being mapped in the primary page table. Therefore, the OS must contain a supplementary page table that can track the ownership of pages between processes, and their states and locations in memory.

Note: A supplementary page table need not be of the same format as the primary table as it is not used directly by hardware. This allows for advanced data structures that can provide a far lower algorithmic time complexity than the naive approach introduced in this lab.

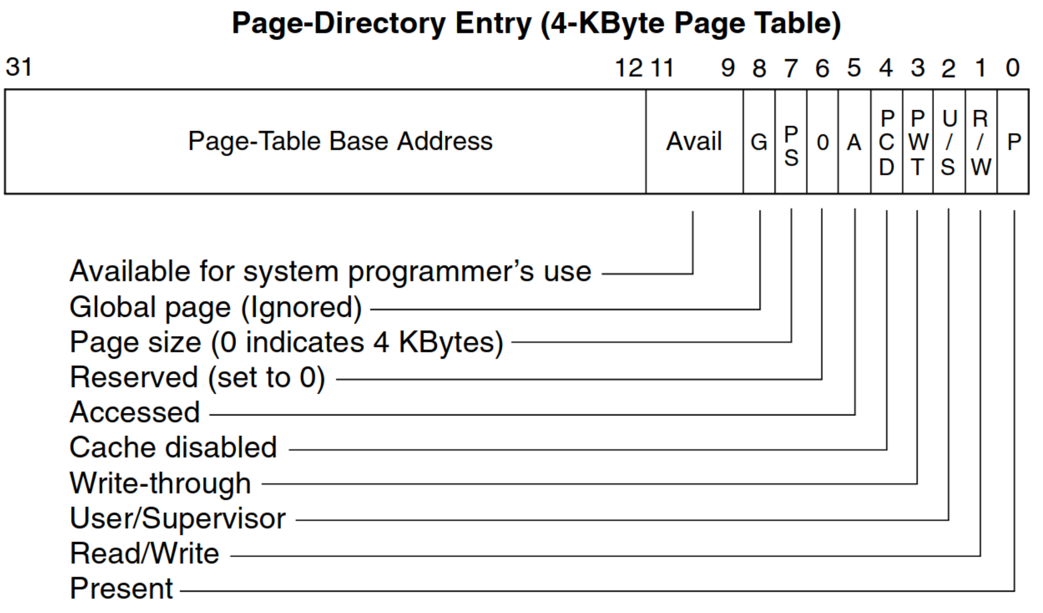
As swapping pages between RAM and secondary storage can be a very slow process due to the inherent limitations of secondary storage speed; page swaps must occur as infrequently as possible. The replacement algorithm must therefore be able to intelligently decide which pages it should swap out to minimise its own work.

For this lab you will be implementing the **Not Recently Used(NRU)** algorithm. NRU is one of the simplest page replacement algorithms. It will trivially select the first page that is of highest preference. Highest preference is defined with the **highest** number in the list below.

1. referenced, modified.
2. referenced, not modified
3. not referenced, modified
4. not referenced, not modified

In order to determine if a page has been referenced or modified, the OS is able to check the status of 2 bits in the page table entry (Figure 7). The CPU sets the access bit to 1 when the page has been accessed, and sets the dirty bit to 1 when the page has been modified. To allow for a notion of recency in the state of page references, on every timer interrupt (a tick) the `access_reset()` function is called which will (once implemented) set the access bit of every present page table entry to 0. Thus recency is defined as the time since the last tick interrupt.

Note: Preference level 3 on first inspection may seem an impossibility, however, in actuality this case can occur very frequently if a page has been written to in a previous tick but not referenced since, as the dirty bit is not reset with the timer.



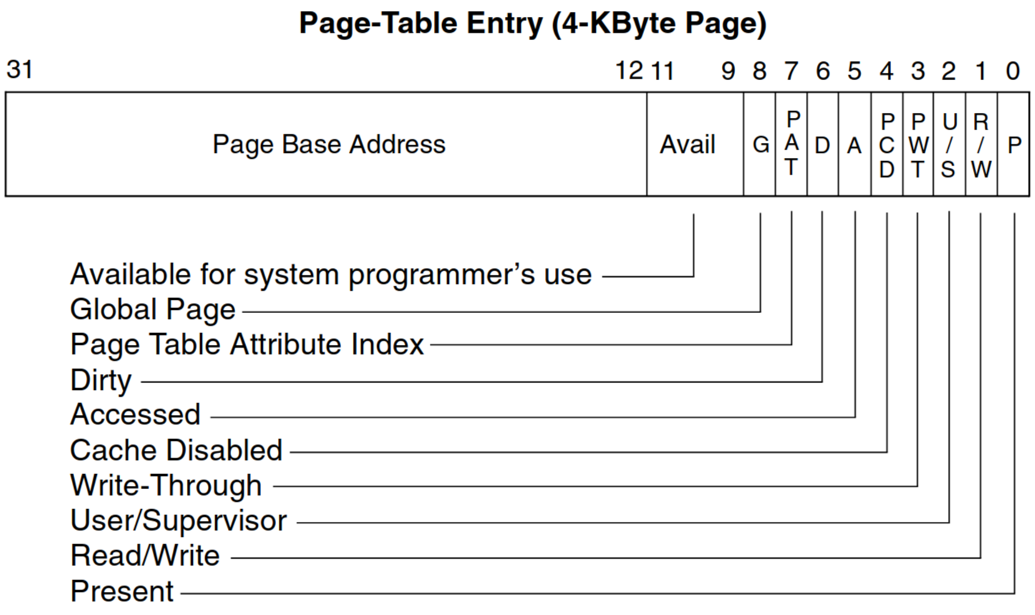


Figure 7: Page Directory and Table Entry Layout[2]

5.3 Exercise

5.3.1 PunchOS Implementation

Again, for simplicity and safety purposes, PunchOS has provided a different environment to test your algorithm. Rather than using a virtualised hard drive to store the swap contents, an area of RAM will be used instead so as to not have to deal with the asynchronous properties of IO operations. Furthermore, another area of RAM will be used to be the "fake" RAM for the test framework that will have severe size limitations, thus forcing frequent use of your replacement algorithm.

5.3.2 What can go wrong?

In a sentence: all of it.

As you will be modifying the page tables, if you make a mistake, it can, in the worst case, cause a hard crash. This may involve a bluescreen, a completely garbled screen, or potentially even a triple fault. A triple fault will cause a hard CPU reset, thus causing it to fall into a bootloop.

Furthermore, if you do not handle the page fault correctly, it will fall into an infinite loop when returning back from the interrupt handler.

If any of these problems occur, go back over your functions that edit the page table and think very carefully about what they're doing. It is often worth thinking about the size of the data you are modifying, and if that may be overwriting something else by accident. The compiler may be able to give you warnings about these potential problems.

5.3.3 Implementing NRU

All functions relating to page replacement functionality can be found in *src/memory/pg_swap.c, pg_swap.h*

Implementing this page replacement algorithm can be reduced to implementing 3 key functions.

- `page_fault_handler()`
- `swap_out_page()`
- `access_reset()`

Firstly you will need to implement the `page_fault_handler()` function. Currently it will simply PANIC with the virtual address that has caused the fault. The handler function takes an exception state struct argument, this struct contains the state of all CPU registers at the time of the interrupt, the value that will be of use to you is the value in the CR2 register. On a page fault the `CR2` register will be updated with the 32-bit virtual address that caused the page fault, thus allowing us to read that register and handle the fault accordingly.

Each process has a page directory; this page directory defines both current page mappings and page meta data. A Page directory can be thought of as an array of 1024 `page_directory_entry` structs. This struct is defined in *page.h* and it is recommended you look at this file alongside the guide. The struct contains the 20 most significant bits for a `page_table_base_addr`, therefore this must be shifted over by 12 bits to obtain the full 32-bit address. Furthermore, this is the physical address, hence it must be translated to a kernel virtual address using the `kptov()` function so it can be referenced and cast to a `page_table_entry` struct pointer much like the page directory.

Implementing: `void page_fault_handler(exception_state *state);`

- Check if the vaddr is valid in the current process' address space using `page_swap_lookup(void * vaddr)`, if it's not, use `page_fault_panic` to suspend execution.
- Next use the `RAM_status` array to check if this virtual RAM is full, if it's not, add it to `virt_RAM` using `memcpy` and updating the status.
- Otherwise, swap out a page using `swap_out_page()`, and add the page to `virt_RAM` using `memcpy` and updating the status.

Implementing: `void swap_out_page();`

- Iterate through the pages in virtual ram, checking for each preference level in NRU to find the most suitable page.
- If the page has been modified, copy the contents back to the `virt_HDD` using `memcpy`.
- Update the associated `swap_page` struct, `virt_RAM` bool, and un-map the virtual address using `invalidate_RAM_page()`.

Implementing: `void access_reset();`

- Iterate through every page directory entry, if that entry is present, iterate through the page table entries of that page, then set the accessed bits to 0.

As previously warned, you must be very careful with this, as any false modification can corrupt the page table and the results will be catastrophic. Good thing we're in an emulator!

Task: Implement the NRU page replacement system. Use the tests to validate your solution.

Chapter 6

PunchOS Tools

6.1 Debugging

Due to the unsympathetic nature of OS development, wherein debugging tools must be made yourself, PunchOS has provided some fundamental tools to assist in your development.

The file containing the majority of the tools explained below is found in *lib/debug.c*.

6.1.1 PANIC

PunchOS has a built in “blue screen” function which can be called through the `PANIC(char *msg)` function. This simply disables interrupts, makes the screen blue, and displays the message and the values of key registers. This PANIC screen is also called when there is an unhandled exception in the code.

An extra feature is the introduction of a `helper` variable, this can be set anywhere in the code, and can be used to add extra information to this crash screen to help debug problems.

Feel free to modify PANIC to your liking.

6.1.2 ASSERT

A simple helper function to check a condition variable, then either PANIC with the given message or continue.

Usage:

```
ASSERT(cond, "message on cond fail");
```

6.1.3 KERN_WARN

This function allows for improved debugging functionality through providing kernel warnings on certain failure conditions for example a failed malloc. These warnings are printed to the screen at the time, but also stored in a buffer which is then displayed in the case of a kernel panic.

Usage:
`KERN_WARN("message");`

6.1.4 HALT

This function will simply disable interrupts and halt the CPU. This has the benefit of being able to see the previous print outputs on the display, and can also be used in conjunction with QEMU's inbuilt debugging tools such as pressing 'Ctrl+Alt+2' to be able to interact with QEMU and print register values. More information is available here: `https://qemu-project.gitlab.io/qemu/system/monitor.html`

Particularly the `x/fmt addr` style commands.

Usage:
`HALT();`

6.1.5 Breakpoint

This function will throw a breakpoint exception meaning it will be handled with a kernel PANIC thus allowing you to inspect the CPU state, and any kernel warnings up until that point.

Usage: `breakpoint();`

6.1.6 GDB

This functionality isn't part of PunchOS but it can still be incredibly useful.

It is possible to connect GDB to QEMU by adding the -s AND -S options to `$QEMU_FLAGS` on boot in the Makefile. Then loading up GDB and remote connecting to the QEMU server with the command `target remote localhost:1234`.

More information is available through:

- `https://qemu-project.gitlab.io/qemu/system/gdb.html`
- `https://wiki.osdev.org/Kernel_Debugging`

6.2 Display

To output to the screen, there is a very simple way to do this. There exists a VGA text buffer at a predetermined memory address. This region is 80x25x2 bytes long, wherein every 2 bytes it contains a colour and a character value. This then repeats for 80 columns then 25 rows. Hence to write a character to a specific point in this space, one must simply do some basic maths to get the linear offset into the region.

There is also a cursor available, however programming this is slightly more complicated.

Luckily however, PunchOS already provides some basic functions to write to the screen and update the cursor for you, which you are free to look over at your discretion. The code is provided in "lib/screen.h" which can be imported as required, just remember to use relative addressing in the path as the lib/ folder has not been compiled directly as the standard library replacement in this lab.

Furthermore, the function `void println(char *msg)` is used instead of the standard C library `printf` function, this is to emphasise the differences in implementation and usage.

Examples of usage are given below:

```
/* Prints "Hello!" on a new line */
println("Hello!");

/* Prints "5" on a newline */
int x=5;
println(itoa(x,str,BASE_DEC)); /* States decimal base. BASE_HEX and
↪  BASE_BIN also exist */

/* Prints "0x1000" on a newline */
int y=0x1000;
println(itoa(y,str,BASE_HEX));
```

Note: Use of `itoa` due to the limited functionality of `println`, `itoa` (integer to ASCII) is required to convert numbers into a string representation.

You are free to look and edit the screen.c and int.c files as you so desire, there are also some more functions in there that may be useful to you also!

6.3 Lists

Lists are used extensively throughout the kernel code for system management. It is therefore important for you to understand what functionality is available to you through lists.

PunchOS lists are defined in lib/list.c. They are doubly linked lists and thus can be used for stacks and queues also with the push, pop and append functions. Each list and it's elements use dynamic heap storage allowing them to be as long or as short as necessary whilst maintaining operational efficiency. Therefore due to the hybrid processing approach described in lab2, each list must either be allocated in either shared or individual process address space. Furthermore, each list element only contains a void * to some space in memory. It is up to the programmer to ensure that this space is allocated and freed correctly.

Figures 6,7 and 8 show use an example use of these lists with a given struct pointer containing two integers x and y.

```
/* Initialising a list in unique address space */
list *lu = list_init();


struct point *p1u = (struct point *) malloc(sizeof(struct point));
struct point *p2u = (struct point *) malloc(sizeof(struct point));

p1u->x=1;
p1u->y=2;

p2u->x=4;
p2u->y=5;
println("Local List");

/* Add to the local list */
append(lu,p1u);
push(lu,p2u);

if(list_get(lu,0)->x==1){
    println("True!");
}
else{
      println("False!");
}


if(list_get(lu,1)->x==4){
    println("True!");
}
else{
      println("False!");
}

list_dump(lu);

remove(lu,p1);
free(p1u);

remove(lu,p2u);
free(p2u);
```

Listing 6: Example usage of local lists.

```
println("Shared list");


/* Initialising a list in shared address space */
list *ls = list_init_shared();


struct point *p1s = (struct point *) shr_malloc(sizeof(struct
↪   point));
struct point *p2s = (struct point *) shr_malloc(sizeof(struct
↪   point));

/* Add to the shared list */
p1s->x=2;
p1s->y=3;

p2s->x=6;
p2s->y=7;

append_shared(ls,p1s);
append_shared(ls,p2s);

list_dump(ls);


// Remove and free
remove_shared(ls,p1s);
shr_free(p1s);

remove_shared(ls,p2s);
shr_free(p2s);
```

Listing 7: Example usage of shared lists.

```
--------------------
System Out:
Local list
True!
True!
[0xb000ABCD, 0xb000EFAB]

Shared list
[0xC0007FA0, 0xC0007FAE]
```

Listing 8: Output of running Figures6,7

Chapter 7

C Support

Explicit support for C is not yet given directly in this guide. Instead, the CS241 module labs provide a very strong introduction available here: `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/labs/`.

Other resources:

- K&R - The C Programming Language[3]
- Learn-C.org[1]
- ...

[1]https://www.learn-c.org/

Chapter 8

Intro to i386

The i386, originally known as the intel 80386, is an old 32-bit x86 CPU launched in 1985 which uses the IA-32 Intel instruction set architecture. Important points about this architecture relating to PunchOS are given below.

8.1 Registers

The registers of an IA-32 ISA CPU can be broken into several key categories as shown in the following subsections.

8.1.1 General Purpose Registers

The general purpose registers include `EAX, EBX, ECX` and `EDX`. They are free to be used at a programmers discretion to assist in computation. They are typically used according to their respective names: Accumulator, Base, Count and Data.

The `E` in `E*X` stands for Extended, meaning 32-bit size rather than the 16bit of a register like `AX`.

`EAX` is typically used to store the return value of functions, however, this is naturally dependant on the data being returned and the calling conventions.

8.1.2 Control Registers

There are 4 control registers, `CR0-4`, each is used to store important system control information. For example:

- `CR0` stores information regarding the CPU state, e.g. if in protected mode or if paging is enabled.
- `CR2` stores the virtual address that caused the most recent page fault.
- `CR3` stores the physical address of the current page directory.

8.1.3 Segment Registers

There are 6 segment registers, `CS, DS, ES, FS, GS` and `SS` that operate independently. However, use of these is not significant in this guide.

8.1.4 Status Registers

Status registers unsurprisingly store the current system status information. There are 3 key registers for this.

- Extended Instruction Pointer(`EIP`):

  AKA: The Program Counter

  Stores the address of the line in memory the CPU is currently executing.
- Extended Stack Pointer(`ESP`):

  Address of the top of the stack.
- Extended Base Pointer(`EBP`):

  Base address of the current stack frame.

8.2 The Stack

Key points about the stack:

- It grows downwards from high addresses to low addresses.
- The logical top of the stack is stored in the `ESP` register.
- There is no memory protection unless self implemented. It is possible for the stack to grow large and overwrite important code or data.
- Each process/thread requires its own stack.

In order for nested function calls, the stack is used extensively. Whenever a function call is made, a new stack frame is created. A stack frame is simply a new section at the top of the stack containing all the information relating to this new function, and how to get back to the previous one. The bounds of this frame are between the `ESP`(top) and `EBP`(bottom) registers.

Each stack frame must, therefore, contain:

- The return address (old `EIP` value)
- Old `EBP` of the previous stack frame so it can be restored on return.
- State of caller/callee preserved registers (`EAX, ECX, EDX`):

  These are registers that may otherwise be clobbered by the callee function operation.
- Local variables.
- Function arguments.

8.3 Paging

i386 facilitates the use of a two level paging system when using 4KiB pages. Thus one page directory maps to one page table, then that table maps to the physical memory address. This is capable of addressing the full 4GiB virtual address space.

i386 also makes use of a Translation Lookaside Buffer(TLB). The TLB is a hardware cache of recently used virtual addresses to their physical addresses to improve performance. When unmapping a page from the page tables, the TLB must be flushed so as to not reference a previously mapped page.

The physical address of the page directory is stored in the `CR3` register.

8.4 Further Resources

For further details. Several high quality resources are available:

- Intel IA-32 ISA documentation
- Assembly usage tutorial
- Intel Assembly Introduction

Chapter 9

Bootloading

How did we get here?

Good question. It all starts with Bootloading!

9.1 What is a bootloader?

Bootloading is the notion of "pulling yourself up by your bootstraps". It is the process of starting small with limited functionality, and using it to grow into a fully functional system. Operating Systems use bootloaders to begin initial operation.

The first bootloading code is contained in the Master Boot Record (MBR). It is a 512-bit block that sits at the very start of every hard drive or SSD. This small section will contain useful information about the drive, and often contain some code to get the rest of the drive loaded into memory and jump there; beginning the bootloading process.

9.2 Setting up the CPU

Once the jump has occurred into the start of bootloader code, the CPU will be in a very low functionality mode for backwards compatibility. The first steps involve ensuring the CPU and registers are in a "safe" state, and moving from the default 16-bit mode to 32 or 64-bit operating modes. Luckily however, we need not do this ourselves because of the handy GRand Unified Bootloader(GRUB). GRUB covers the majority of the initial setup, moving the CPU into 32-bit mode, checking memory (discussed in Section 9.3), and most importantly moving the kernel code from secondary storage into RAM! The state left by GRUB is well defined (in the Multiboot specification[1]), but not safe, the first steps in *src/boot/boot.asm*[2] are to setup the stack and begin operating safely.

GRUB and Multiboot provide many more potential benefits that can be set in the Multiboot header. The header is defined in the MBR that tells the CPU that the following code is Multiboot compatible[3], along with any specification of desired functionality.

If you would like more details about what a bootloader looks like without GRUB, feel free to email me and I can send you some of my early prototypes without GRUB.

9.3 Memory Mappings

The information regarding memory address regions that are free in RAM for the OS to use are provided by SMAP(Supervisor Memory Access Protection). SMAP can be used through a BIOS interrupt[4] that will load a data structure providing basic information about the available memory (section number, start address, end address, type,...) to a predefined memory address. This can then be parsed by the OS and used to calculate it's available memory regions. However, one of the many benefits of GRUB is that it can generate this SMAP data structure for us so we only have to parse it later on.

Luckily, SMAP is one of the many features GRUB provides!

This SMAP data structure, useful as it is, provides no information regarding what data has been loaded by the bootloader, nor how much. Therefore to ensure our dynamic data allocations are not overriding any kernel code or data we can include labels in the linker file. These labels can be placed at key points in the linker file such that their addresses can be used to infer the bounds of kernel data.

[1]https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html

[2]It is best to look at this file in Lab 1, other Labs employ more advanced bootloading steps.

[3]It more complicated than this.

[4]BIOS interrupts are interrupts that are only available in 16-bit mode and are prebuilt into ROM to provide key functionality such as SMAP systems or displaying to VGA out.

9.4 Getting to C

Once having setup the CPU to a safe state. It is then only a case of jumping to a C function, you must only make it available with an `extern` instruction first.

Although, unfortunately, it's not quite this simple. Your C and assembly code will be compiled and assembled separately, therefore it must then be linked together so that this `extern` call points to the correct relative address of the intended C function and vice versa. It is possible to link these files together with `gcc`, and normally this is fine. However, as we are using GRUB and working directly with memory, we must set the virtual address offsets for the code in a linker file. By default, GRUB loads kernel code at 1MiB+, leaving the space below for information carried over from the bootloading phase including as SMAP details. Therefore, this first *boot.asm* code must be placed at that address. Unfortunately still, it is not quite this simple, so for further detail, feel free to email me individually for support.

Bibliography

[1] Avi Silberchatz. *Operating Systems Concepts, Ninth Edition*. Wiley, 2012.

[2] Intel Corporation, Denver, Colorado. *IA-32 Intel ® Architecture Software Developer's Manual*, 2004.

[3] D.M. Ritchie and B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.