Software Requirements Document for CROS

**TEAM: 16**

AUTHORS: Connor Persels, Nathan Giddings, Hunter Overstake, Kyle Clements, and Sam Lane

| **Version** | **Date** | **Author** | **Change** |
| --- | --- | --- | --- |
| 0.1 |  | SM |  |
| 1.0 | 1/29/24 | CP, SL, NG, HO, KC | Initial Draft |
| 1.1 | 2/15/24 | SL | Updates from demo 1 |
| 1.2 | 4/09/24 | CP, SL, NG, HO, KC | Updates from demo 2 |

**Table of Contents**

[**Purpose 3**](#_30j0zll)

[**Scope 3**](#_1fob9te)

[**Definitions, acronyms, abbreviations 3**](#_3znysh7)

[**Product Perspective 4**](#_1t3h5sf)

[Concept of Operations 4](#_3vobd2vw5uyd)

[Major User Interfaces 5](#_qe1661ked4rc)

[Diagram of communication from userspace to kernel space and back 5](#_hzvjidsizwgk)

[Hardware Interfaces 6](#_euz8wzk8mt6z)

[**Product functions 6**](#_4d34og8)

[Diagram kernel operations and how it handles input and output from userspace 6](#_bwdduj44s3zw)

[UC-1 7](#_le6vo1hzm30z)

[UC-2 7](#_12broc82lsiz)

[UC-3 8](#_sgvekexokja1)

[**Constraints 8**](#_17dp8vu)

[**Assumptions and Dependencies 8**](#_tp2xz9t4js1l)

[**FEATURES 9**](#_35nkun2)

[Shell Commands Table 9](#_4t8vn79khcnx)

[Kernel 10](#_gmicmc28haqv)

[Interface 10](#_pncrb5vp5ost)

[Memory Management 10](#_4xlamichzume)

[Process Management 10](#_4cdiqytsx7d6)

[Signals 11](#_7ap8fd3rx6yz)

[File I/O 11](#_g2vi5d8nhl0n)

[**Performance requirements 11**](#_1ksv4uv)

[**Software System Attributes 11**](#_2jxsxqh)

[Reliability 11](#_axs554nt9bhc)

[Useability 12](#_wkzwyjipqt9s)

[Scalability 12](#_6yg5g67i4m8v)

[Maintainability 12](#_9hi14t6dx9yr)

[Portability 13](#_ssyinuadp6r1)

[Security/Safety 13](#_16oh6pjaz691)

# 

# Purpose

The purpose of this document is to establish how the application should interact with the end user, and establish all application requirements functional, and non functional. Once finalized, this document will state what must be accomplished for the application to be considered finished.

# Scope

This SRS covers a number of potential use cases that users may encounter, as well as an overview of the project and its intended uses. It also includes information on the project’s UI sketches, but the primary purpose is to give detailed descriptions of anticipated use cases.

# Definitions, acronyms, abbreviations

| Term | Description |
| --- | --- |
| Kernelspace | Privileged operating mode with complete control over system resources, access to system registers and instructions, and ability to modify memory mapping and permissions. Only the kernel operates in this mode, allowing it to manage system resources in a controlled way. |
| Userspace | Unprivileged operating mode with no access to kernel memory, no ability to directly modify memory mapping and access permissions, and no access to (system-dependent) privileged instructions. All non-kernel software runs in userspace. |
| Process | A program managed by the kernel whose memory is isolated from all others. The kernel schedules CPU time for processes, manages the allocation of resources, and controls filesystem access. |
| Signal | An event related to a process that occurs and is handled asynchronously. The kernel executes a user-provided function each time a signal occurs in a process. |

# 

# 

# Product Perspective

“Rasbian” the official operating system for the Raspberry Pi, is based on Linux

“DietPi” a lightweight operating system specifically designed for single board computers

“Ubuntu MATE” a variant of Ubuntu created specifically for the Raspberry Pi

“Ubuntu Server” a variant of Ubuntu without a GUI and completely terminal based

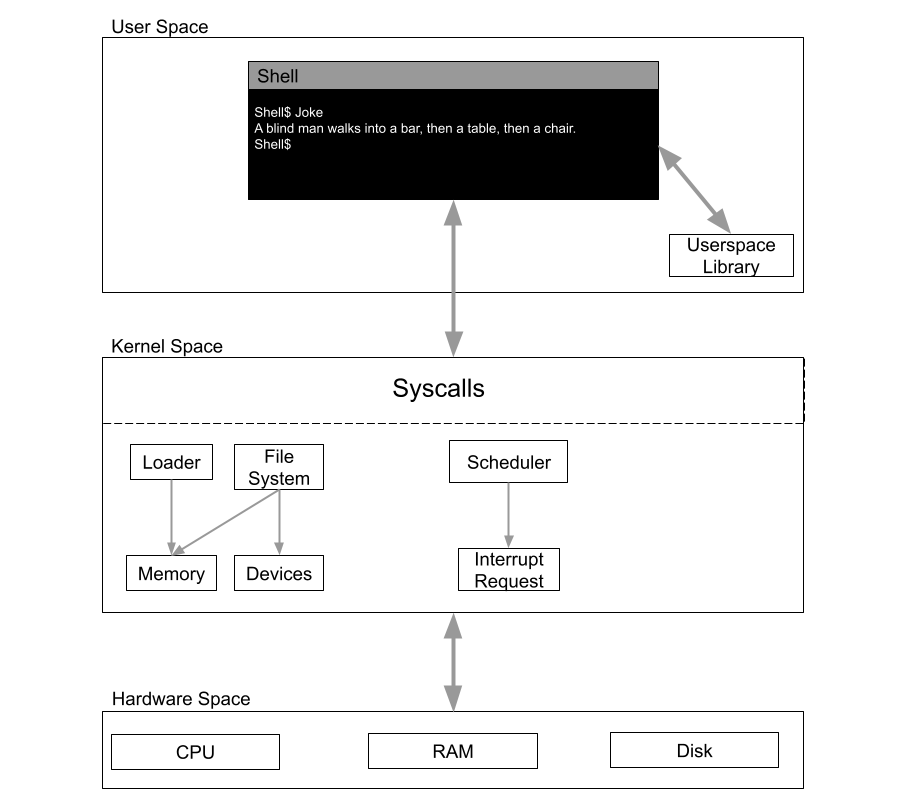
## Concept of Operations

This project will be an operating system development from scratch in C++ targeting the Raspberry Pi called CROS (Custom Raspberry Pi Operating System). The operating system will be terminal-based with no windows, displays, or GUIs. It will be able to execute user-provided programs as well as have the ability to store and retrieve data via a file system. It will also have GPIO support in order to utilize external input/output devices.

The project will have a shell where the user will interact with the operating system via a terminal. The user will be able to run a set of commands in order to execute programs, navigate the file system, and other helpful things. The project will have a kernel where user space programs run as processes on a scheduler, system calls are used to manage memory, manipulate a process, raise signals, query the file system, and more. The project will also have custom OS libraries for use in the user space as well as kernel space. These libraries will include a custom C library to act as the standard C library and a container library to house data structures / other useful tools.

This project will provide an operating system that is customizable to any clients need, such as being able to create new custom shell commands and tailor scheduler to prioritize responsiveness over fairness. This operating system will also be able to easily port to other 64-bit Pi versions such as the Pi 4 and Pi 5.

## Major User Interfaces

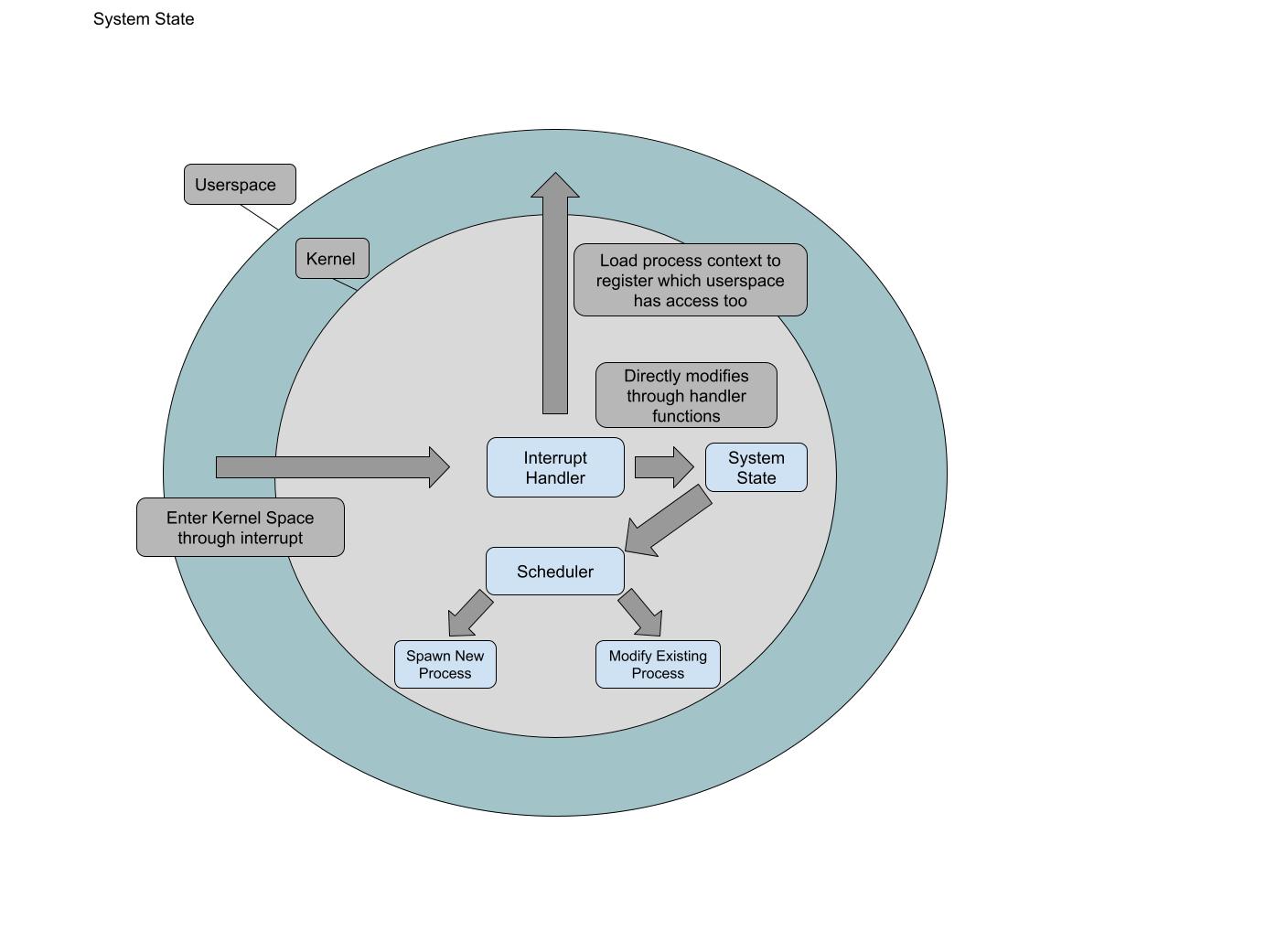


##### Diagram of communication from userspace to kernel space and back

## Hardware Interfaces

Raspberry Pi 3 and newer. This is due to the fact that Pis before the Raspberry pi 3 are not 64 bit and have different booting protocols.

## Product functions



##### Diagram kernel operations and how it handles input and output from userspace

## UC-1

*Main scenario: Using the search command on a file with shell*

* **Trigger**: The user enters the search command with a file
* **Postcondition** Every line containing the keyword in the search is outputted to the terminal
* **Main Story**
  + The search program is executed with the correct args given by the user.
  + Parse every line in the file and print if the matching token is contained.
  + Search ends and the user is given a prompt from the shell.

## UC-2

*Main scenario: Userspace program reading/writing to a file*

* **Trigger:** Userspace program issues systems calls: open(), read(), write(), close()
* **Postcondition:** Data is read from / written to the the file on disk
* **Main Story:**
  + userspace program is running and needs to read data from a file
  + program calls the system call open() to open the follow
  + program calls the system call read() to get the specified amount of data.
  + program then needs to write data to a file
  + program calls the system call write() to write the specified data to disk
  + program is done reading and writing to file
  + program calls the system call close() to close the file

## UC-3

*Main scenario: Running a new process*

* **Trigger:** User enters a shell command
* **Postcondition:** Command finishes executing and the shell is ready to run another command
* **Main Story:**
  + The shell reads a command from user input
  + It spawns a new thread using the system call clone()
  + The new thread executes the requested program as a new process using the system call exec()
    - The new process runs in a separate address space from the shell that created it
    - The new process is scheduled independently from the original shell process
  + The original thread remains intact, and waits for the child process to end.
  + Once the shell receives a signal that the child process has ended, it reads the next command.

## Constraints

Due to the nature of our project, we have to write all the libraries and functions that we want to use. We have access to the documentation for libraries like libc, but we can’t use them directly, instead, we have to write custom implementations that work with our syscalls and project structure.

## Assumptions and Dependencies

Our project has no dependencies due to the fully custom nature of our implementation. We are writing all the code that we use. We are operating under the assumption that the hardware is 64 bit and follows the boot protocol of the raspberry pi 3 or newer.

# FEATURES

## Shell Commands Table

| (stretch goal) lua | Processes the given argument as lua.  (e.x.)  **lua “x=1; print(x);”**  *1*  **lua -x file.lua**  *file.lua executes* |
| --- | --- |
| joke | Prints a random joke |
| File navigation | Manipulate the file system  Standard file system navigation:   * cd * rm * mv * cp * pwd * mkdir |
| ls | view files in working directory |
| Program execution | executes program in /path/to/some/program |
| Command piping | Chaining the output of one command into the input of another |
| wc | word count |
| search | similar linux grep functionality w/o regular expressions |
| Control operators | Standard control operators like ; && |
| echo | Echos back what is in the argument |

## Kernel

### Interface

The kernel will interface with userspace programs via a system call interface. These calls provide programs with access to kernel functionality such as memory management and filesystem access. The kernel should provide the low-level functionality required for an implementation of userspace libraries such as the C standard library.

System call table

| Memory Management | Process Management | Signals | File I/O |
| --- | --- | --- | --- |
| mmap() | clone() | sigraise() | open() |
| munmap() | terminate() | sigret() | create() |
|  | exec() | sigwait() | unlink() |
|  | yield() | sigaction() | read() |
|  |  |  | write() |
|  |  |  | close() |

### Memory Management

These calls control the allocation and mapping of memory within a process’s address space. **mmap** allocates sufficient memory and maps it to the requested region. **munmap** frees previously allocated memory and maps the requested region. A userspace program should be able to use these calls to implement more sophisticated memory management, such as **malloc** and **free** of the C Standard Library.

### Process Management

These calls control the lifespan and scheduling of userspace programs. They allow processes to spawn additional processes, load new program images, and cease operation. Using these calls, userspace programs should be able to implement library functions such as **fork** and **exit**.

### 

### Signals

These calls provide access to process signals. They allow the caller to modify the action taken when a signal occurs (if any), return to normal operation from a signal handling routine, raise a signal on a particular process, and pause until a signal occurs. They allow processes to wait for certain events without the use of polling, and facilitate a low-level form of communication between processes.

### File I/O

These calls provide filesystem access to userspace programs. They provide functionality to create and delete files, and read and write to/from files. The caller is expected to allocate buffers in its for the purpose of reading and writing data. These calls open files to prepare for I/O, perform I/O using caller-provided buffers and data, and handle the creation and deletion of files. Using these calls, a userspace library should be able to implement the I/O functionality of the C standard library.

# Performance requirements

Programs complete before the heat death of the universe

# Software System Attributes

## Reliability

* The completed OS should be able to boot and attempt to run any userspace program without crashing.
* In the event of a userspace program failing, the kernel should be capable of giving an error message to the user and kill said program gracefully.
* In the event of a userspace program breaking userspace (a.k.a. the shell), the kernel will kill the userspace program gracefully and then restart the user space in its default state.

## 

## Useability

* The user should be able to access all of the functionality of the OS and their own programs with only needing the prior knowledge of how to use a command line prompt.
* The shell will feel responsive to the user as they input commands and receive a display of accurate responses to said inputs.
* To help the user know about the OS’s functionality, the shell can display a help menu to reveal the shell’s functionality.

## Scalability

* The OS will finish in a state that will have the capability of being open sourced. This also means the OS will have numerous open ended projects a future developer can work on that includes but is not limited to:
  + driver code for various types of ports
  + Internet protocols
  + Userspace text editor
* Memory is managed appropriately and will allow future additions to be made without affecting performance times of already implemented features.

## Maintainability

* The code will be organized into a modular based style that is both neat and intuitive. This will allow future developers to gain an understanding of the code at a much faster pace and easy decision making on where to store future updates.
* Documentation will be clear and concise to help a future developer understand how the OS functions efficiently.

## 

## Portability

* The finished os should have the ability to be transferred to any Raspberry Pi 3B and newer. The host system must support the ARMv-8 architecture, which excludes the Raspberry Pi 1 and 2.

## Security/Safety

* The user will not be able to manipulate the internal functions of the kernel. The system call interface is the only means by which a userspace program can interact with the kernel.
* Memory of the kernel and userspace will be kept separate from each other. The user will not have any access to the kernel’s memory whatsoever. Attempting to access kernel memory from user space, either reading or writing, should result in the offending process being terminated with an error condition.