FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

MoKaFr OS

Raspberry Pi 4

Lukas Frey freyl@students.zcu.cz
Hana Kalivodová hanakali@students.zcu.cz
Stepan Mocjak mocjaks@students.zcu.cz

4.12.2022

Obsah

Raspberry Pi 4

BCM2711

V dokumentaci BCM2711 jsou popsány periferie, které jsou bezpečně dostupné na ARM. Mezi tyto periferie patří Timer, Interrupt controller, GPIO, USB, PCM/I2S, DMA controller, I2C masters, SPI masters, PWM a UARTs. Bohužel mnoho dalších funkcionalit, které procesor nabízí nejsou zdokumentovány.

Aplikační doména

Operační systém slouží jako základ pro budoucí vývoj chytrého zrcadla, který má za úkol předávat uživateli užitečné informace jako například aktuální čas či počasí.

Uspořádání paměti

Fyzické adresy v RAM začínají na adrese 0x000000000 a jsou omezeny velikostí instalační SD karty. Na fyzických adresách z rozsahu od 0x47C000000 do 0x7FFFFFFF jsou dostupné hlavní periferie, od fyzické adresy 0x4C0000000 do 0x4FFFFFFFF lokální periferie ARMu. Periferie jsou následně dostupné (mapované) na adrese 0xFE000000.

Vektory výjimek:

EL 0 (user mode)		EL 1 (kernel mode)	
0x000	Sync	0x200	Sync
0x080	IRQ	0x280	IRQ
0x100	FIQ	0x300	FIQ
0x180	SError	0x380	SError

Uspořádání paměti OS:

Periferie
0xFE000000
Halda
0x40000000
Kernel
0x80000
Zásobník
0x000

AArch64

Operační systém jsme psali pro 64 bitovou architekturu AArch64. Operační systém využívá označení x pro 64-bitové registry a w pro 32-bitové registry. Přehled registrů pro ARMv8 jsou uvedeny v přiloženém souboru arm64.pdf. Níže jsou uvedeny speciální registry pracující s režimy procesoru.

SPSR_EL{1..3}	stav procesu při vstupu do EL{1..3}
ELR_EL{1..3}	návratová adresa z EL{1..3} při výjimce
SP_EL{0..2}	stack pointer EL{0..2}
SPSel	výběr SP (0: SP=SP_EL0, 1: SP=SP_ELn)
CurrentEL	aktuální režim (na bitech 3..2)
DAIF	aktuální maska přerušení (na bitech 9..6)
NZCV	příznaky podmínek (na bitech 31..28)
FPCR	ovládání systému s pohyblivou desetinnou čárkou
FPSR	stav systému s pohyblivou desetinnou čárkou

Popis implementace OS

Spuštění

Celý systém se bootuje z adresy 0x80000. Na adrese se nachází kód s prefixem _start. Zde si ověříme na kterém jádře se nacházíme a pouze pro jádro 0 pokračujeme dále. Ostatní se “uspí” instrukcí WFE. Jádro 0 dále pokračuje na kód kernel_entry kde se přepne do režimu EL1 (Jádro). V dalším kroku se již pouze nastaví stackpoiner na adresu _start (0x80000), vynuluje se BSS sekce, nastaví se vektor přerušení a spustí se hlavní funkce main.

Funkce main()

Tato funkce se stará o většinu podstatných věcí při spuštění. Zde nainicializujeme všechny ovladače (UART, Video, FileSystem, Interrupty, Timer) a spustíme procesy.

TRNG

Úkolem práce bylo například zprovoznění True Random Generátoru Čísel. Po dlouhém zkoumání a přečtení mnoha fór i zdrojových kódů (Linux) jsme zjistili, že by se měl nacházet na adrese FE104000. Bohužel i když jsme použili danou adresu, tak jsme po mnoha pokusech a debugování generátor nezprovoznili.

PRNG

Místo True Random Generátoru Čísel jsme naimplementovali Pseudonáhodný Generátor Čísel. Jako seed používáme pseudo jednotku (počet ticků) které nám generuje uživatelský proces. Pokud je počet ticků stejný jako minule, využije se jako seed minulé vygenerované číslo. Pro první vygenerování se využije číslo zapsáno v hlavičkovém souboru (PRNG_SEED).

GPIO

GPIO driver slouží pro práci s GPIO piny. Umožňuje nastavení a uvolnění bitů pomocí funkcí, které se starají o bit shifting a podobně. Navíc umožňují také změnu alternativních funkcí pomocí enumerátorů.

GFX

GFX driver umožňuje vykreslování obrazců a řetězců na displej připojený k raspberry přes HDMI kabel. Implementován je přes frame buffer a videocore mailbox. Pro vykreslení je k dispozici paleta o 16 barvách a 8x8 font obsahující znaky a-Z, číslice 0-9 a několik dalších speciálních znaků pro interpunkční znaménka, závorky a podobně. Aktuálně jsou k dispozici funkce pro vybarvení jednoho pixelu, vykreslení čáry, čtverce, kruhu a vypsání textu či znaku na obrazovku. Nechybí ani funkce pro vyprázdnění obrazovky. Dále operační systém nabízí mechanismus pro vykreslování obrazců a řetězců na obrazovku pomocí rour.

Timer

V rámci vývoje jsme implementovali 3 časovače, které poskytuje BCM2711. V rámci vývoje jsme počítali s možností spuštění na QEMU, jelikož jsme ze cvičení zjistili, že pro Raspberry Pi Zero je v QEMU funkční časovač systémový, naimplementovali jsme jej jako první. Bohužel záhy jsme zjistili, že v naši verzi QEMU nefunguje. Proto jsme se rozhodli naimplementovat lokální časovač, který ovšem není příliš dobře zdokumentovaný v dokumentaci k BCM2711 a detaily jeho funkčnosti jsme hledali v dokumentaci pro Cortex-a72 (Jádro využívané v BCM2711). Zde ho lze dohledat jako Generic Timer. Ten se nám ovšem také nepodařilo zprovoznit. Rozhodli jsme se tedy pro potřeby plánovače žádný časovač nevyužívat (v QEMU) a implementovali jsme základní ARM side timer který je v dokumentaci BCM2711 popsaný dobře. Ten ve výsledku využíváme pro generování přerušení a následné plánování úloh na reálném zařízení.

Interrupt controller

Pro správu přerušení využíváme nestandardně GIC400. Jedná se o ovladač přerušení přidaný do Raspberry Pi s verzí 4. Oproti Legacy interrupt controlleru je ovšem méně zdokumentovaný (Dokumentace BCM2711) a detaily jsme hledali přímo v dokumentaci pro GIC400.

UART

Jedna z prvních věcí které se nainicializují. je UART ovladač. ten se stará o odesílání znaků a zároveň přijímání a následné ukládání do lokálního bufferu.

Interrupts

Přerušení řešíme tak, že máme tabulku přerušení která se nastaví při spouštění systému. Přerušení následně povolíme ve funkci init_interrupt_controller() a následně obsluhujeme ve funkci handle(). Zde následně buď řešíme pokud nám přišlo přerušení z timeru (64) pošleme ho dále do funkce timer_interrupt_handler(), pokud z UARTu (jediné funkční na QEMU) tak ho pošleme dále do funkce uart_handler().

Scheduler

Pro plánování jsme vytvořili preemptivní plánovač, který se spouští přes funkci schedule() buď v main() nebo ve funkci timer_tick() která se spouští při přerušení časovačem, nebo přerušení UARTem. Pokud uživatel pošle zpětné lomeno (\). UART je jediná možnost jak pracovat s přerušením v QEMU. Každý proces běží na oddělené úrovni EL0 (User Space).

Filesystem

V operačním systému je implementován jednoduchý filesystem in-memory, tedy po odpojení Raspberry Pi 4 z napájení se vytvořené soubory a adresáře smažou. Celý filesystém je ukládán do tabulky maximální velikosti 256 uzlů. Každý uzel obsahuje informaci o svém id, zda je již obsazený, adresu do tabulky stránek, ve které se nacházejí jeho data a informaci, zda z něj někdo momentálně zapisuje nebo z něj čte. Další strukturou je directory, která obsahuje pole dat (directoryEntry), která jsou v adresáři uložena. Struktura directoryEntry obsahuje název souboru, informaci, zda jde o soubor, nebo adresář a id uzlu, ve kterém jsou uložena jeho data. Soubor je reprezentován strukturou File, ve které jsou uložena data jako pole znaků. Maximální velikost souboru (adresáře) je stejná jako velikost stránky, tedy 1MB.

Terminál

Obsahuje mnoho podpůrných funkcí pro práci s filesystémem:
mkdir [název] - vytvoření nového adresáře
ls - vypsání obsahu aktuální složky
cd [cesta] - změna aktuálního adresáře
touch [název] - vytvoření souboru
pwd - vypsání cesty do aktuálního adresáře
cat [název] - vypsání obsahu souboru

A další funkce:
draw: vykreslení obrazce či řetězce na obrazovku dle zadaného vstupu (využívá roury).

Pro vykreslení čtverce:
draw rect|<x1>|<y1>|<x2>|<y2>|<color>|<fill>
např: draw rect|700|700|900|900|0x13|1

Pro vykreslení kruhu:
draw circle|<x>|<y>|<radius>|<color>|<fill>
např: draw circle|300|300|100|0x13|1

Pro vykreslení řetězce:
draw string|<x>|<y>|<text>|<color>
např: draw string|500|500|Hello world!|0x13

ticks: vypsání počtu tiknutí od spuštění operačního systému
rand: Získání náhodné hodnoty

Popis nahrávání do Raspberry Pi 4

Build:

cd ./scripts
./build.sh

Zprovoznění v QEMU:

cd ./scripts
./run.sh

Zprovoznění na reálném zařízení:

Je potřeba zařízení Raspberry Pi 4.
Zkopírovat ./build/kernel8.img do kořenového adresáře na SD kartě. Pokud se již na kartě nachází kernel8.img, musíte ho smazat.
Připojit k Raspberry Pi HDMI kabel s Monitorem.
Zapojit Raspberry Pi do napájení :)

Emulátor QEMU

Emulátor QEMU defaultně nepodporuje Raspberry Pi 4. Použili jsme tedy verzi, která je zveřejněna na adrese https://github.com/0xMirasio/qemu-patch-raspberry4. Jedná se o upravenou verzi QEMU která by měla podporovat i desku Raspi4. V emulátoru bohužel mnoho funkcí není dostupných nebo nefungují dle popisu dokumentace.

V emulátoru se nám povedlo zprovoznit terminál, filesystem, UART, přerušení klávesou '\', PRNG. Nefunguje zde ovšem časovač. Pro potřebu přepínání procesu se zde využívá UART přerušení, kdy se pošle zpětné lomeno (\).

Problémy při implementaci

Práci, pochopení a implementaci operačního systému nám komplikovala nedostatečná dokumentace a emulátor QEMU, který nepodporuje Raspberry Pi 4. Hodně času jsme strávili jenom tím, že jsme hledali použitelnou verzi QEMU a následným debugováním a zjišťováním, co na dané verzi vůbec funguje. Dále jsme neměli k dispozici žádný debugger, který bychom připojili k reálnému zařízení. Během vývoje jsme se také snažili zprovoznit JTAG debugger založený na BluePill vývojové desce a openOCD.

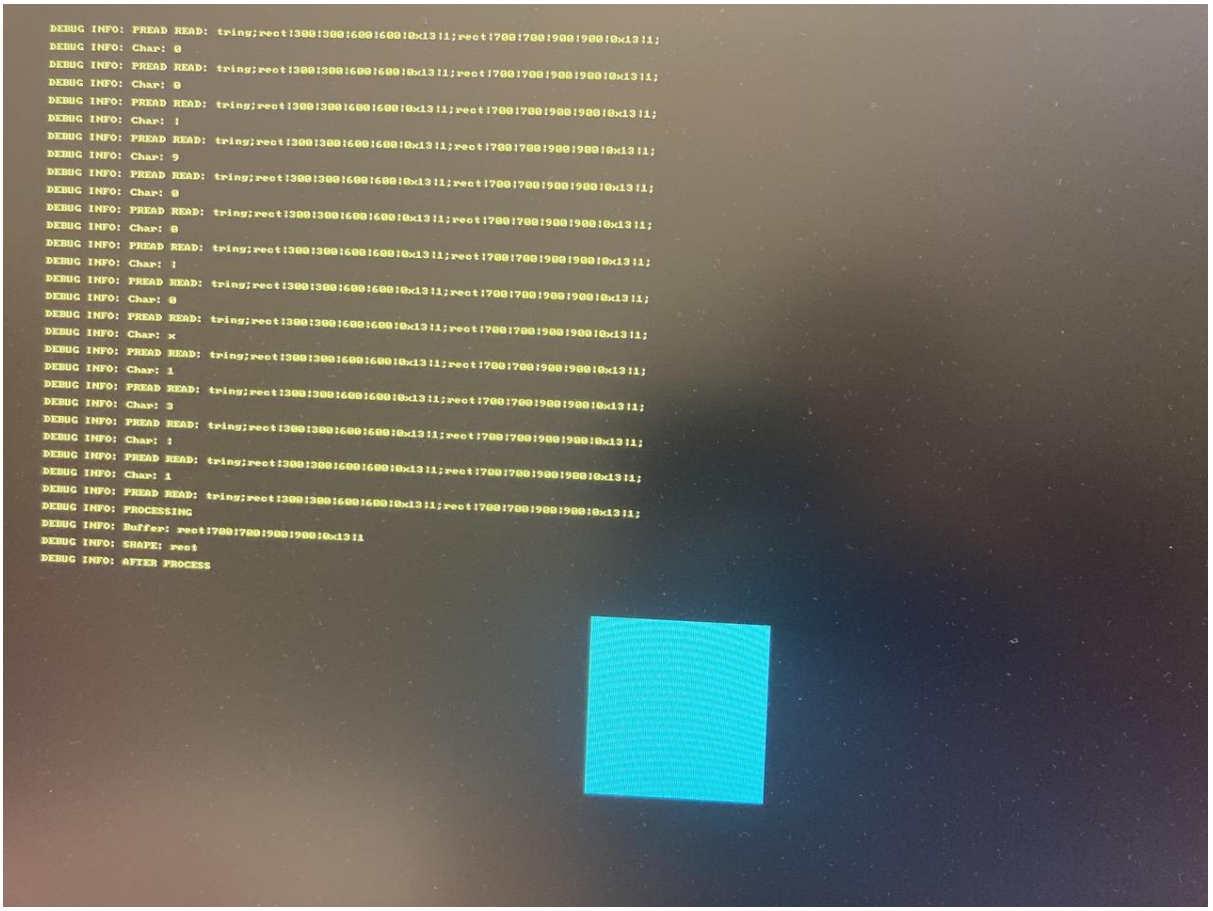
V přiloženém dokumentu můžete vidět rozpracovanou branch, kde jsme se snažili zprovoznit MMU. To se nám bohužel nepovedlo hlavně z časových důvodů. Z části kterou jsme naimplementovali jsme zjistili, že naše sektory se nastavovali jenom jako read-only a kvůli tomu jsme nemohli spustit dolní část kernelu. Problem se nám nepodařilo vyřešit a nepomohla nám ani konzultace se cvičícím.

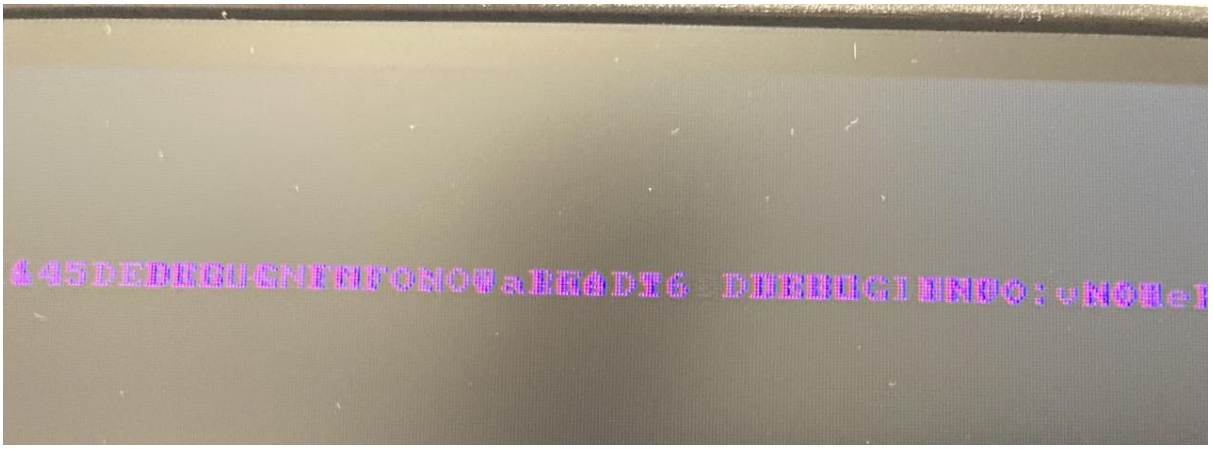
Snažili jsme se dále také zprovoznit Bluetooth driver, který se nám ovšem nepovedl zprovoznit, kromě vybuildění firmware do kernelu našeho OS.

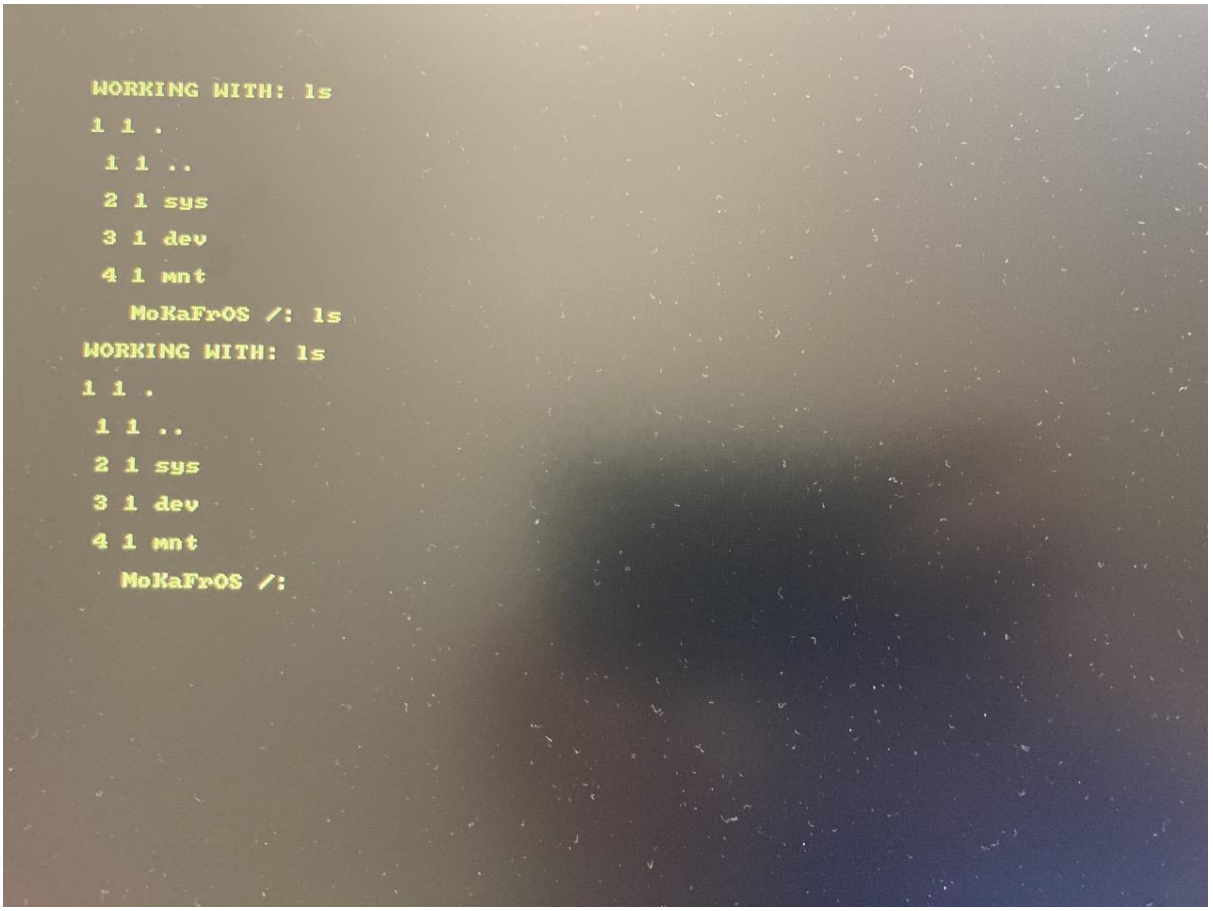
Také jsme se snažili zprovoznit TRNG, a to více způsoby, žádný však nevedl ke správnému výsledku. Průzkum nám obzvlášť ztížila chybějící dokumentace. Je tedy možné, že princip generátoru je správný, ale registrům, se kterými generátor pracuje, nastavujeme nesprávné hodnoty.

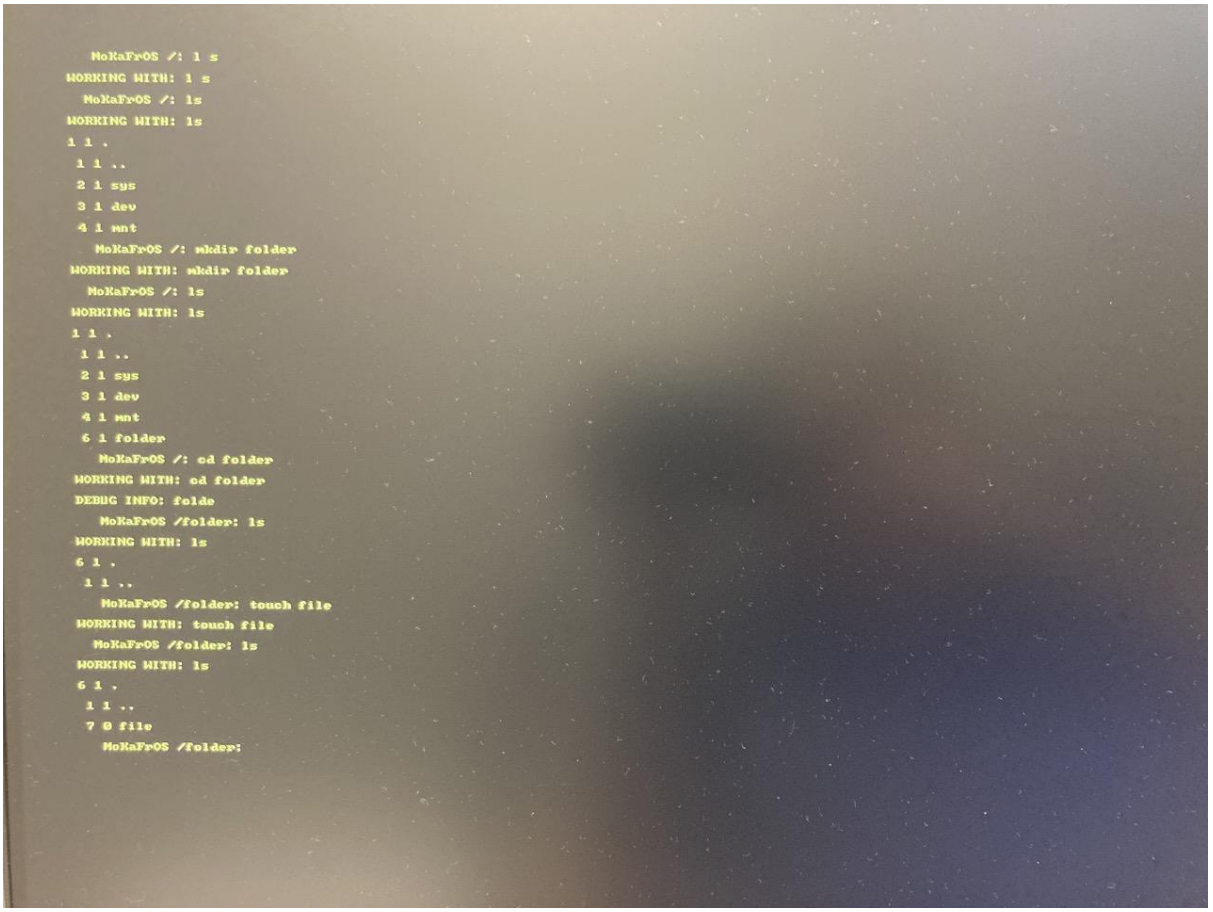
Výklad na cvičení pro nás byl velmi přínosný, proto nám nepomohlo, že nám tři cvičení odpadla a všechny cvičení se posunula. Největším problémem pro nás byl čas.

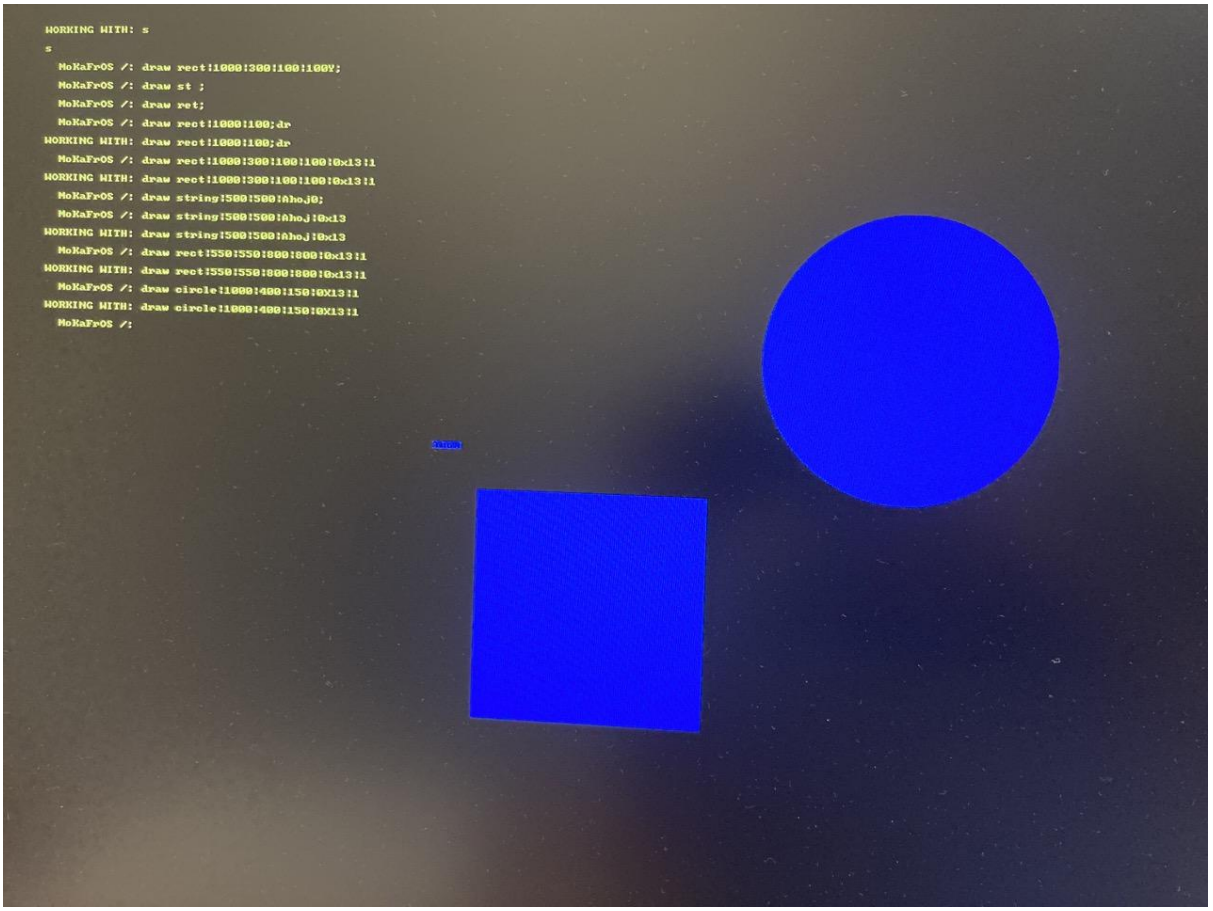
Screenshots






WORKING WITH: ls
1 1 .
1 1 ..
2 1 sys
3 1 dev
4 1 mnt
MoKaFrOS /: ls
WORKING WITH: ls
1 1 .
1 1 ..
2 1 sys
3 1 dev
4 1 mnt
MoKaFrOS /:


MoKaFrOS /: l s
WORKING WITH: l s
MoKaFrOS /: ls
WORKING WITH: ls
1 1 .
1 1 ..
2 1 sys
3 1 dev
4 1 mnt
MoKaFrOS /: mkdir folder
WORKING WITH: mkdir folder
MoKaFrOS /: ls
WORKING WITH: ls
1 1 .
1 1 ..
2 1 sys
3 1 dev
4 1 mnt
6 1 folder
MoKaFrOS /: cd folder
WORKING WITH: cd folder
DEBUG INFO: folde
MoKaFrOS /folder: ls
WORKING WITH: ls
6 1 .
1 1 ..
MoKaFrOS /folder: touch file
WORKING WITH: touch file
MoKaFrOS /folder: ls
WORKING WITH: ls
6 1 .
1 1 ..
7 0 file
MoKaFrOS /folder:


WORKING WITH: s
s
MoKaFrOS /: draw rect!1000!300!100!100Y;
MoKaFrOS /: draw st ;
MoKaFrOS /: draw ret;
MoKaFrOS /: draw rect!1000!100;dr
WORKING WITH: draw rect!1000!100;dr
MoKaFrOS /: draw rect!1000!300!100!100!0x13!1
WORKING WITH: draw rect!1000!300!100!100!0x13!1
MoKaFrOS /: draw string!500!500!AhoJ0;
MoKaFrOS /: draw string!500!500!AhoJ!0x13
WORKING WITH: draw string!500!500!AhoJ!0x13
MoKaFrOS /: draw rect!550!550!800!800!0x13!1
WORKING WITH: draw rect!550!550!800!800!0x13!1
MoKaFrOS /: draw circle!1000!400!150!0X13!1
WORKING WITH: draw circle!1000!400!150!0X13!1
MoKaFrOS /:

Závěr

Operační systém jsme psali v programovacím jazyce C a částečně v Assembleru. Semestrální práce nás stála mnoho času a úsilí, neboť to byla naše první zkušenost s psaním operačního systému. Hodně času jsme strávili debugováním programu v jazyce C, především pokud se jednalo o pointery.

Zpočátku jsme uvažovali na implementaci v jazyce C++, ale jelikož s ním nemáme zkušenosti, rozhodli jsme se implementovat OS v C. Dále jsme také přemýšleli nad implementací v jazyce Rust.

Plánovali jsme implementovat ovladač pro ARM Mailbox, jehož popis je uveden v dokumentaci BCM2711. Ovladač měl sloužit pro ovládání LED která se nachází na desce.

Při tvorbě operačního systému jsme se naučili programovat přímo pro fyzické zařízení. Dále jsme se naučili číst z technických dokumentací. Během průzkumu jsme jich prošli opravdu mnoho.

Bohužel jsme nestihli zprovoznit rozdělení kernelu na dolní a horní kernel a virtuální adresové prostory.

Implementovali jsme ovšem mnoho jiných funkcionalit. Naprogramovali jsme vlastní FileSystem, správu paměti, roury, subsystém přerušení, izolaci uživatelských procesů (režim EL0), pseudonáhodný generátor čísel, ovladač pro display.