SYKOM Projekt

Systemy komputerowe: architektura i programowanie

Kacper Średnicki

Politechnika Warszawska, Cyberbezpieczeństwo 24L

17 maja, 2024

Spis treści

1. Temat projektu

Tworzenie układów SoC z peryferiami wytworzonymi przez siebie i emulowanymi przez spersonalizowany program QEMU oraz testowanie tego systemu z wykorzystaniem tworzonej dla tego środowiska dystrybucji systemu Linux i odpowiednich sprzętowi sterowników systemowych.

2. Założenia

2.1. Opis

W ramach projektu należało stworzyć własny system operacyjny dla własnoręcznie przygotowanego zestawu sprzętowego. Zestaw ten miał zostać zaimplementowany w języku opisu sprzętu Verilog i odpowiednio modyfikować emulator QEMU. W celu umożliwienia współpracy pomiędzy pomiędzy tworzonym systemem operacyjnym a peryferiami (w szczególności z elementem sprzętowym), należało utworzyć moduł jądra systemu Linux. Poprawność działania systemu miała być weryfikowana m.in. z wykorzystaniem przygotowanej na tę potrzebę aplikacji użytkownika. Wszystkie prace wykonane zostały na maszynach z odpowiednimi narzędziami dla procesora RISC-V. Postępy tych prac natomiast były systematycznie publikowane w indywidualnym repozytorium GIT.

2.2. Specyfikacja

Otrzymano szczegółową specyfikację projektową:

— Moduł Verilog miał realizować operację wyznaczenia N'tej liczby pierwszej ($N \leq 1000$). Wpisanie wartości do rejestru A, reprezentującego argument tej operacji, miał uruchamiać automat wyznaczający N'tą liczbą pierwszą. Przez rejestr S dostępny miał być aktualny stan automatu (przyjęta koncepcja została opisana w sekcji 3.1). Wartość znalezionej liczby pierwszej miała być natomiast dostępna w 32-bitowym rejestrze W. Na wyprowadzeniu GPIO modułu gpioemu miała się również pojawiać liczba wszystkich znalezionych liczb pierwszych od włączenia systemu.
— Należało odpowiednio przygotować pliki źródłowe modułu jadra systemu Linux, komunikującego moduł sprzętowy z aplikacja użytkownika. Dane przekazywane pomiędzy tą aplikacją a modułem jądra miały być w tekstowym formacie: OCT.
— Wbudowana w docelowy filesystem aplikacja użytkownika miała testować poprawność działania całego systemu w różnych przypadkach jego użycia.

Indywidualnie przydzielone zostały także adresy przestrzeni GPIO i rejestrów (udostępnianych przez gpioemu i widocznych przez CPU):

— A - jako SYKT_GPIO_ADDR_SPACE + 0x238 (dostępny przez plik: /proc/proj4srekac/rejA),
— S - jako SYKT_GPIO_ADDR_SPACE + 0x250 (dostępny przez plik: /proc/proj4srekac/rejS),
— W - jako SYKT_GPIO_ADDR_SPACE + 0x248 (dostępny przez plik: /proc/proj4srekac/rejW).

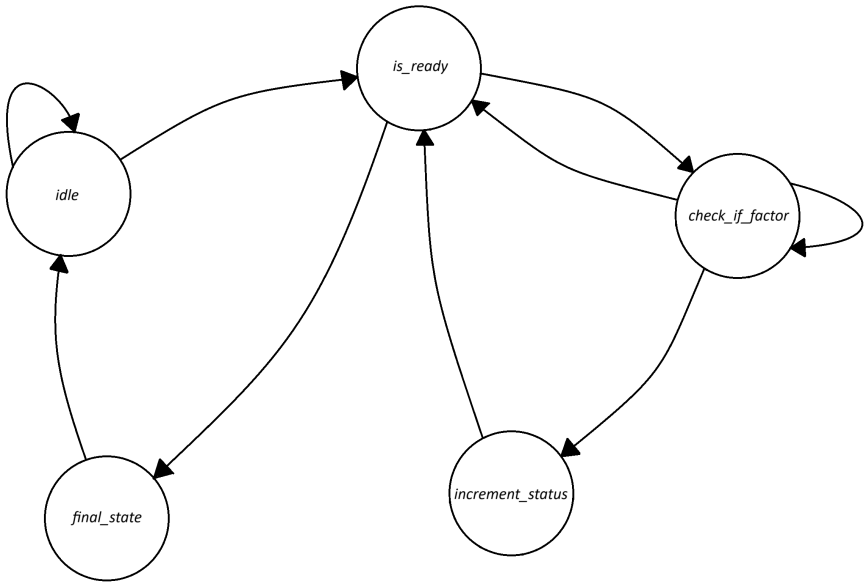
3. Moduł gpioemu

3.1. Implementacja modułu gpioemu

Przygotowany w języku Verilog moduł gpioemu realizuje operację wyznaczania N'tej liczby pierwszej. Implementuje automat (maszynę stanów), który w konkretnych stanach wykonuje ściśle określone operacje.

W opisanej w sekcji 2.2 specyfikacji zdefiniowany został rejestr S, reprezentujący aktualny stan automatu. Należało samodzielnie zaproponować jakie wartości ma on udostępniać. Zdecydowano, że status S określał będzie, ile liczb pierwszych zostało znalezionych do tej pory, od rozpoczęcia aktualnego poszukiwania (wczytania argumentu). Taka konwencja okazała się przydatna podczas implementacji aplikacji użytkownika, opisanej w sekcji 5.1. Przyjęto również założenie, że podczas poszukiwania wskazanej liczby pierwszej (czyli od momentu przypisania wartości rejestrowi `A`, do momentu osiągnięcia przez status S wartości równej A), wartość rejestru `W` wynosić będzie 0.

Wyróżnionych zostało pięć stanów: *idle*, *is_ready*, *check_if_factor*, *increment_status* oraz *final_state*. W kodzie logika przejść między stanami zrealizowana została w jednej z procedur `always`. Za przechowywanie informacji o aktualnym stanie automatu odpowiedzialny jest 3-bitowy rejestr `state_reg`. Przygotowano wizualizację tej logiki i przedstawiono ją na rysunku 1.



Rys. 1: Wizualizacja przejść między stanami automatu

Równolegle, w innej procedurze `always` realizowana jest logika mikrooperacji. To tam wykonywane są charakterystyczne dla konkretnego stanu operacje na samodzielnie zdefiniowanych do obliczeń rejestrach. Rejestry te, a także wykorzystywane parametry zostały przedstawione poniżej:

```
// wykorzystywane rejestry
reg [9:0] A; // argument
reg [9:0] S; // status
reg [31:0] W; // wynik
reg [15:0] current_number;
reg [15:0] potential_factor;
reg start;
reg [2:0] state_reg;

// adresy
localparam A_address = 16'h238;
localparam S_address = 16'h250;
localparam W_address = 16'h248;

// stany automatu
localparam [2:0] idle = 3'h0,
    is_ready = 3'h1,
    check_if_factor = 3'h2,
    increment_status = 3'h3,
    final_state = 3'h4;
```

Rys. 2: Samodzielnie zdefiniowane rejestry i parametry wykorzystywane w implementacji

Poniżej został przedstawiony opis poszczególnych stanów i wykonywanych w nich mikrooperacji:

1. *idle*:
   — Tzw. stan bezczynności.

— Sprawdza czy flaga (rejest) `start` jest ustawiona na 1 - jeśli tak, to rozpoczyna proces poszukiwania wskazanej liczby pierwszej, przechodząc do stanu *is_ready*.

2. *is_ready*:
   — Sprawdza czy proces poszukiwania wskazanej liczby pierwszej został zakończony.
   — Jeśli status S jest mniejszy niż argument A, to przechodzi do kolejnej analizowanej liczby zwiększając wartość `current_number` o 1 oraz przechodzi do stanu *check_if_factor*.
   — W przeciwnym wypadku (gdy wartość statusu S równa się wartości argumentu A) przechodzi do stanu *final_state*, ponieważ oznacza to, że poszukiwanie wskazanej liczby pierwszej zostało zakończone.
   — Przypisuje fladze start wartość 0.
3. *check_if_factor*:
   — Jeśli potencjalny dzielnik (`potential_factor`) aktualnie analizowanej liczby jest nie większy niż połowa wartości `current_number`, to następnie sprawdza czy `current_number` jest podzielny przez `potential_factor`. Jeśli tak, to rejestr `potential_factor` ustawiany jest na domyślną wartość 2, a automat powraca do stanu is_ready. Jeśli nie jest podzielny, to przechodzi do kolejnego potencjalnego dzielnika, zwiększając `potential_factor` o 1 i z nową wartością ponownie wchodzi do stanu *check_if_factor*.
   — Jeśli potencjalny dzielnik przekroczył już połowę wartości aktualnie analizowanej liczby, to przechodzi do stanu *increment_status*. Oznacza to bowiem, że aktualnie analizowana liczba jest liczbą pierwszą.
4. *increment_status*:
   — Inkrementuje wartość statusu S jako, że znaleziona została kolejna liczba pierwsza.
   — Przechodzi do stanu *is_ready* gdzie nastąpi porównanie nowej wartości statusu z wartością argumentu.
5. *final_state*:
   — Końcowy stan automatu.
   — Przypisuje `current_number` do wynikowego rejestru W.
   — Przechodzi do stanu *idle*, w którym automat będzie oczekiwał na pojawienie się kolejnego argumentu.

W pozostałych procedurach `always` zostały zrealizowane schematyczne operacje, takie jak: reset, zapis, odczyt, czy zatrzask danych. Dodatkowo, w osobnej procedurze, dokonywana jest inkrementacja wartości wyjścia GPIO w sytuacji, gdy automat osiągnął stan *final_state*. Dokonywane są także odpowiednie przypisania do sygnałów typu `output`, z wykorzystaniem konstrukcji `assign`.

Zrzut z ekranu terminala przedstawiony na rysunku 3 potwierdza, że plik `gpioemu.v` kompiluje się prawidłowo oraz pomyślnie wytwarzany jest plik `qemu-system-riscv32-sykt`.

```
sykt@deb4sykom03:~/moje_prace/Srednicki_Kacper/projekt/hardware$ makeQemuGpioEmu gpioemu.v
Checking enviroment...
Checking was done.
Compiling the Verilog file (/home/sykt/moje_prace/Srednicki_Kacper/projekt/hardware/gpioemu.v) into CPP by Verilator...
Compiling the Verilog product in CPP format into BIN...
Compiling the wraper file for gpioemu extension...
Making libgpioemu library...
Libgpioemu is ready in: /home/sykt/moje_prace/Srednicki_Kacper/projekt/hardware/libgpioemu.a
Linking the /var/local/qemu/riscv32-softmmu to final executable file with name qemu-system-riscv32-sykt...
The 'qemu-system-riscv32-sykt' is placed at '/home/sykt/moje_prace/Srednicki_Kacper/projekt/hardware' location. Bye!
```

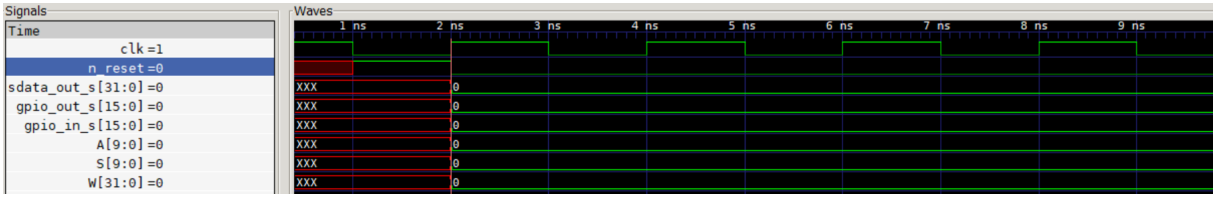
Rys. 3: Kompilacja pliku `gpioemu.v`

3.2. Testy modułu gpioemu

Na potrzebę weryfikacji poprawności działania modułu gpioemu, przygotowano dedykowany testbench. Sformułowano w nim kilka, różnorodnych przypadków testowych i z wykorzystaniem programu GTKWave analizowano uzyskane rezultaty. Kolejne testy oraz ich wyniki wraz z komentarzami zostały przedstawione poniżej.

3.2.1. Poprawność resetu

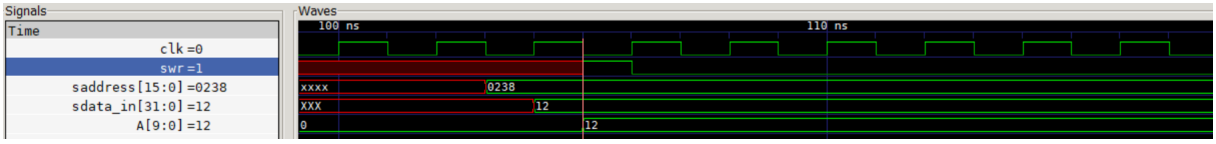
Przetestowano poprawność wykonania resetu. Zgodnie z oczekiwaniami, wraz ze zboczem opadającym `n_reset`, wartości rejestrów obsłużonych w odpowiedzialnej za reset procedurze `always` są ustawiane na 0.



Rys. 4: Początkowy reset

3.2.2. Poprawność zapisu

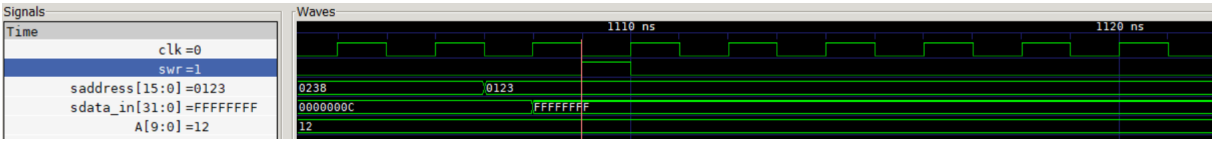
Następnie zweryfikowano poprawność zapisu. Rejestrowi `saddress` przypisano adres 0x238, powiązany (zgodnie ze specyfikacją) z argumentem A. Po pojawieniu się wartości 1 na `swr`, pomyślnie dokonany został zapis wartości 12 z `sdata_in` do rejestru `A`.



Rys. 5: Zapis argumentu

3.2.3. Podtrzymanie wartości

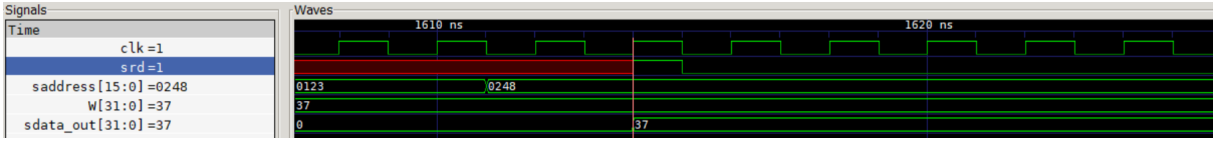
Sprawdzono również poprawność podtrzymania wartości podczas zapisu na adres, niepowiązany z żadnym peryferium. Prawidłowo, pojawienie się zbocza narastającego na `swr` nie spowodowało zapisania nowej wartości z `sdata_in` do rejestru `A`. Na rejestrze tym została podtrzymana wcześniej zapisana wartość.



Rys. 6: Zapis na niepowiązany adres

3.2.4. Poprawność odczytu

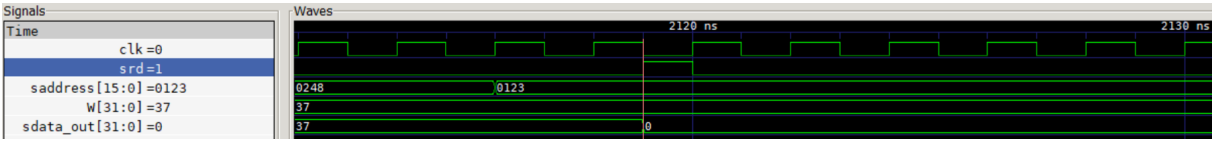
W dalszej kolejności podjęto się weryfikacji poprawności odczytu. Rejestrowi `saddress` przypisano adres 0x248, powiązany (zgodnie ze specyfikacją) z wynikiem W. Po pojawieniu się wartości 1 na `srd`, pomyślnie dokonany został zapis wartości 37 z rejestru `W` do `sdata_out`. Wykazana została również poprawność odnalezienia (zgodnie z zapisaną w sekcji 3.2.2 wartością do rejestru `A`) 12-stej liczby pierwszej: 37.



Rys. 7: Odczyt wyniku

3.2.5. Neutralność odczytu

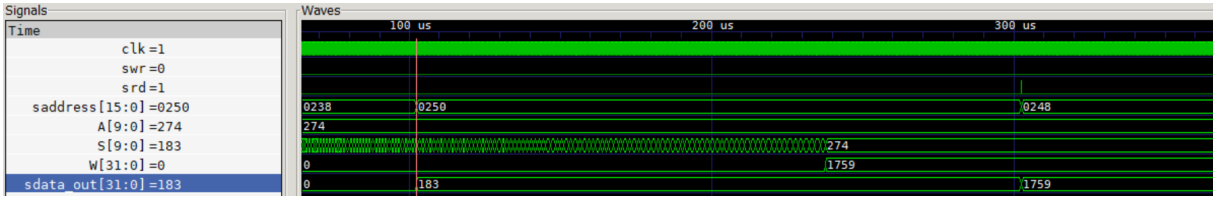
Zweryfikowano także neutralność podczas odczytu z adresu, niepowiązanego z żadnym peryferium. Prawidłowo, pojawienie się zbocza narastającego na `srd` nie spowodowało przypisania wartości rejestru `W` do `sdata_out`. Wartość `sdata_out` została za to wyzerowana.



Rys. 8: Odczyt z niepowiązanego adresu

3.2.6. Poprawność odczytu statusu i wyniku po zapisaniu kolejnego argumentu

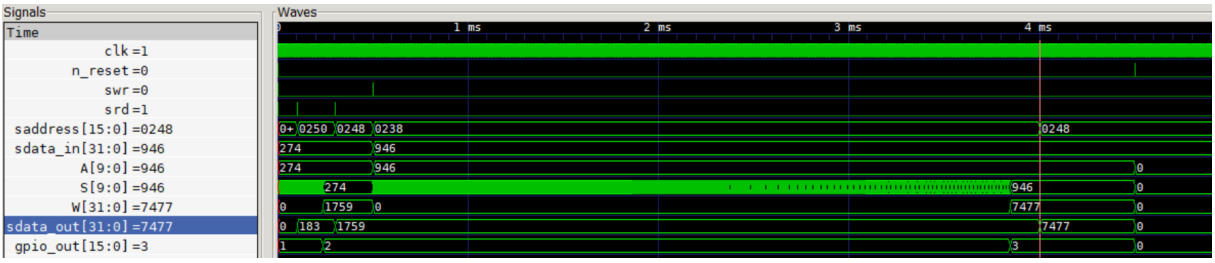
Następnie dokonano zapisu argumentu 274 i sprawdzono poprawność odczytu statusu. Rejestrowi `saddress` przypisano adres 0x250, powiązany (zgodnie ze specyfikacją) ze statusem S. Wraz z pojawieniem się wartości 1 na `srd` przed zakończeniem poszukiwania zadanej liczby pierwszej (przed osiągnięciem przez rejestr S wartości 274), do `sdata_out` została przypisana chwilowa wartość z rejestru `S` (w tym przypadku 183). W dalszej kolejności rejestrowi `saddress` przypisano adres 0x248, powiązany z wynikiem W i po odczekaniu na zakończenie poszukiwania, dokonano odczytu poprawnie znalezionej 274-tej liczby pierwszej: 1759.



Rys. 9: Odczyt statusu podczas liczenia i końcowego wyniku

3.2.7. Poprawność obliczeń dla dużego argumentu

Dokonano także analogicznej weryfikacji poprawności poszukiwania zadanej liczby pierwszej, dla dużego argumentu (946). Odczytano poprawną wartość znalezionej 946-tej liczby pierwszej: 7477. Znalezienie tej liczby zajęło modułowi niecałe $3,5ms$. Dodatkowo, na poniższym zrzucie widoczne są wartości na `gpio_out`, przez cały czas wykonywania testbench'a. Wartość na wyjściu GPIO poprawnie określa liczbę wszystkich znalezionych do tej pory liczb pierwszych, a w sytuacji resetu - zostaje wyzerowana.



Rys. 10: Odczyt wyniku dla dużego argumentu oraz przedstawienie wartości na wyjściu GPIO

4. Moduł jądra systemu Linux

4.1. Implementacja modułu jądra

Zgodnie z opisem założeń, przedstawionym w sekcji 2.1, zaimplementowano moduł jądra systemu Linux. Umożliwia on współpracę pomiędzy pomiędzy tworzonym systemem operacyjnym, a peryferiami (w szczególności z przygotowanym elementem sprzętowym opisanym w sekcji 3). Poniżej przedstawione zostały poszczególne elementy implementacyjne modułu jądra w pliku `kernel_module.c`.

4.1.1. Nagłówki, definicje, zmienne globalne

— Zaimportowane niezbędne do działania modułu nagłówki zawierające definicje wykorzystywanych funkcji oraz makra związane z przestrzenią adresową.

— Definicja modułu określająca m.in. licencję, autora czy wersję modułu.
— Definicja stałych, m.in:
  — przestrzeni adresowej (`SYKT_GPIO_ADDR_SPACE`, `SYKT_GPIO_SIZE`),
  — adresów rejestrów w przestrzeni adresowej GPIO (`A_REG`, `S_REG`, `W_REG`),
  — rozmiaru bufora wykorzystywanego w funkcjach odpowiedzialnych za odczyt i zapis (`BUF_SIZE`),
  — rozmiaru pamięci do zmapowania wykorzystywany w funkcji inicjalizującej moduł (`MEMO_SIZE`),
  — nazwy głównego katalogu w katalogu '`/proc`' (`MY_DIRECTORY`),
  — nazwy plików rejestrów w tym katalogu (`A_FILENAME`, `S_FILENAME`, `W_FILENAME`).
— Deklaracja wskaźników:
  — na początek zmapowanej pamięci dla całej przestrzeni adresowej (`baseptr`) i poszczególnych rejestrów (`baseptrA`, `baseptrS`, `baseptrW`),
  — na struktury reprezentujące katalog '`/proc`' (`directory`) i pliki w tym katalogu (`file_A`, `file_S`, `file_W`).

4.1.2. Funkcje do odczytu

Zaimplementowano osobne funkcje do odczytu wartości z poszczególnych rejestrów: `read_A()`, `read_S()` oraz `read_W()`. Jako, że każda z funkcji ma analogiczną strukturę, poniżej przedstawiona i opisana została jedynie funkcja `read_W()`.

```
static ssize_t read_W(struct file *file, char *ubuf, size_t count, loff_t *offs) {
    int not_copied, to_read, length;
    char buf[BUF_SIZE];

    to_read = readl(baseptrW);

    if(*offs > 0) {
        return 0;
    }

    snprintf(buf, BUF_SIZE, "%o\n", to_read);
    length = strlen(buf);
    not_copied = copy_to_user(ubuf, buf, length);

    if(not_copied != 0) {
        printk(KERN_ERR "Error copying to user.\n");
            return -EFAULT;
    }

    *offs = *offs + count;
    return count;
}
```

Rys. 11: Funkcja `read_W()`

Funkcja została stworzona na podstawie funkcji `lsfr_write_function()` przedstawionej w prezentacji *wprowadzenie_do_projektu.pdf*, próbując wykorzystać używane tam struktury do zrealizowania operacji odczytu. Podstawą działania przygotowanej funkcji `read_W()` jest funkcja `readl()`, pozwalająca na czytanie z określonych miejsc w pamięci. W tym przypadku odczytuje wartość z zmapowanej przestrzeni adresowej, wskazywanej przez `baseptrW`. Rozmiar bufora wykorzystywanego do przechowywania odczytanej wartości w formacie tekstowym jest ściśle zdefiniowany na 6 (definicja `BUF_SIZE` została wspomniana w sekcji 4.1.1). Wynika to z faktu, że maksymalna wartość, która może być przechowywana w rejestrach, to tysięczna liczba pierwsza: OCT 17357. Najdłuższa wartość składa się zatem z 5-ciu znaków, do których doliczyć należy tzw. string terminator, co w sumie daje potrzebny rozmiar bufora wynoszący 6. Funkcja `snprintf()` formatuje odczytaną przez funkcję `readl()` wartość jako liczbę oktalną i zapisuje ją do bufora. Zastosowano funkcję `snprintf()` ze względu na większą bezpieczeństwo w kontekście potencjalnego przepełnienia bufora, niż w przypadku funkcji `sprintf()`. Następnie dane ze wspomnianego wcześniej bufora są kopiowane do bufora użytkownika. Z wykorzystaniem offsetu realizowana jest kontrola tego,

czy dane zostały już odczytane. Obsługiwane są również podstawowe błędy podczas kopiowania danych. Funkcja `read_W()` zwraca liczbę odczytanych bajtów.

4.1.3. Funkcja do zapisu

Na potrzeby zapisu niezbędne było zaimplementowanie jednej funkcji `write_A()`. Została ona przedstawiona poniżej.

```
static ssize_t write_A(struct file *file, const char *ubuf, size_t count, loff_t
↪  *offs) {
    int to_write;
    char buf[BUF_SIZE] = {0};

    if(count > sizeof(buf)) {
            printk(KERN_ERR "Too long input.\n");
            return -EINVAL;
    }

    if((copy_from_user(buf, ubuf, count)) != 0) {
            printk(KERN_ERR "Error copying from user.\n");
            return -EFAULT;
    }

    // printk(KERN_INFO "Received: %s\n", buf);

    for (int i = 0; i < count - 1; i++) {
            if (buf[i] < '0' || buf[i] > '7') {
            printk(KERN_ERR "Wrong input, not octal.\n");
            return -EINVAL;
            }
    }

    if(sscanf(buf, "%o", &to_write) != 1) {
            printk(KERN_ERR "Wrong input format.\n");
            return -EINVAL;
    }

    buf[sizeof(buf) - 1] = '\0';
    int max_octal = 1750;

    if(to_write <= 0 || to_write > max_octal){
            printk(KERN_ERR "Octal value out of range.\n");
            return -EINVAL;
    }

    writel(to_write, baseptrA);
    return count;
}
```

Rys. 12: Funkcja `write_A()`

Podobnie jak w przypadku opisanej wyżej funkcji do odczytu, tak również w przypadku funkcji `write_A()` starano wzorować się na funkcji udostępnionej w prezentacji przez Prowadzącego. W funkcji `write_A` bufor jest inicjalizowany zerami, aby zapewnić, że dane są w nim poprawnie zakończone znakiem string terminator. Odwrotnie niż w przypaddku funkcji do odczytu, w funkjci do zzapisu funkcją `copy_from_user()` bajty z bufora użytkownika kopiowane są do bufora jądra. Funkcja `sscanf()` konwertuje łańcuch znaków w buforze jądra na liczbę oktalną i zapisuje ją do zmiennej `to_write`. W buforze dla pełnego bezpieczeństwa ustawiany jest string terminator. Wreszcie funkcja `writel()` zapisuje wartość `to_write` do zmapowanego adresu wskazywanego przez `baseptrA`. Funkcja `write_A()` zwraca

liczbę zapisanych bajtów, mogącą potwierdzić pomyślne dokonanie zapisu. W funkcji dokonywanych jest kilka istotnych weryfikacji zapisywanych wartości i obsługa ważnych błędów. Na etapie testowania po wstępnej implementacji, funkcja nie działała zgodnie z oczekiwaniami. Problemy rozwiązało wspomniane filtrowanie wartości i obsługa błędów, które zostały do tej funkcji zaczerpnięte od Adriana Zalewskiego. Wprowadzone modyfikacje poskutkowały poprawnością zapisu wartości. Szczególnie istotna jest pętla `for`, w której każdy wprowadzony znak sprawdzany jest pod kątem odpowiednich dla systemu oktalnego wartości. Na końcowym etapie testów odkryłem jeszcze jeden błąd, skutkujący brakiem możliwości zapisania liczby składającej się z jednego znaku. Powód tego błędu został wykryty dzięki wykorzystaniu zakomentowanej linii widocznej na powyższym rysunku z kodem. String terminator był dodawany przed wypełnieniem bufora danymi. Odkrycie tego błędu pozwoliło w pełni skutecznie zapisywać wartości z wykorzystaniem funkcji `write_A()`.

4.1.4. Struktury operacji na plikach

Zdefiniowane zostały trzy struktury operacji na plikach, które mogą być wykonywane na plikach wirtualnych w systemie plików '`/proc`'. Każda ze struktur dotyczy operacji możliwych dla konkretnego pliku. W przypadku pliku `rejA` ustawiane są wskaźniki do funkcji `read_A()` i `write_A()`, aby obsługiwany był zarówno odczyt, jak i zapis do tego pliku. W przypadku plików `rejS` i `rejW` wskaźnik ustawiany jest wyłącznie do funkcji `read_S()` / `read_W()`.

4.1.5. Inicjalizacja modułu

W funkcji `my_init_module()`, z wykorzystaniem `ioremap()`, mapowane są fizyczne adresy pamięci rejestrów na adresy dostępne w przestrzeni jądra. Na tak zmapowane adresy wskazywać będą `baseptr`, `baseptrA`, `baseptrS` oraz `baseptrW`. Za pomocą `proc_mkdir()` tworzony jest katalog w katalogu '`/proc`', a następnie za pomocą `proc_create()` tworzone są w nim pliki `rejA`, `rejS` oraz `rejW` o określonych prawach dostępu oraz możliwych na nich operacjach. Poprawność mapowania adresów, tworzenia katalogu '`/proc`' oraz trzech wspomnianych plików jest weryfikowana, a odpowiednie błędy obsługiwane.

4.1.6. Usuwanie modułu

W funkcji `my_cleanup_module()` następuje zapis do rejestrów sterujących emulowanego CPU i wywoływana wewnętrzna procedura zakończenia pracy emulatora QEMU. Z wykorzystaniem `remove_proc_entry()` usuwane są trzy pliki z katalogu '`/proc`' oraz sam katalog, a z wykorzystaniem `iounmap()` zwalniany zmapowany obszar pamięci.

4.2. Pozostałe pliki

Niezbędne było również odpowiednie utworzenie innych plików, których treść została udostępniona przez prowadzącego. Te pliki to:

— `kernel_module/Config.in` - opis modułu Buildroot,
— `kernel_module/kernel_module.mk` - zarządza ogólnym procesem budowania modułu jądra,
— `kernel_module/src/Makefile` - zawiera instrukcje kompilacji modułu jądra.

Dzięki odpowiedniemu stworzeniu powyższych plików wraz z modułem jądra w pliku `kernel_module.c`, narzędzie `make_busybox_kernel_module` wywołuje proces kompilacji modułu jądra oraz w przypadku powodzenia wytwarza pliki: `fw_jump.elf` (bootloader), `Image` (obraz jądra systemu) oraz `rootfs.ext2` (główny system plików).

Poprawność kompilacji modułu jądra i pomyślne wytworzenie wymienionych wyżej plików zostały udokumentowane na zrzucie ekranu widocznym na rysunku 16.

```
sykt@deb4sykom03:~/moje_prace/Srednicki_Kacper/projekt$ make_busybox_kernel_module
Changing place...
Checking contents of source files...
Removing previous compilation...
Preparing new compilation...
Module compilation in progress...
mke2fs 1.45.6 (20-Mar-2020)
Module compilation done.
Done, files: fw_jump.elf, Image, rootfs.ext2 are ready to use.
```

Rys. 13: Kompilacja modułu jądra

4.3. Testy z wykorzystaniem narzędzi systemowych

Zweryfikowano poprawność działania stworzonego modułu jądra z wykorzystaniem narzędzi systemowych `echo` oraz `cat`. Wyniki tych testów zostały przedstawione na rysunku 14. Wartości podawane i odczytywane są w systemie oktalnym.

```
# cd proc/proj4srekac/
# ls
rejA  rejS  rejW
# cat rejA
0
# cat rejS
0
# cat rejW
0
```

Rys. a: Test 1

```
# echo "5" > rejA
# cat rejA
5
# cat rejS
5
# cat rejW
13
```

Rys. b: Test 2

```
# echo "45" > rejA
# cat rejW
0
# cat rejS
25
# cat rejS
32
# cat rejS
40
# cat rejS
45
# cat rejW
235
```

Rys. c: Test 3

```
# echo "310" > rejA
# cat rejS
310
# cat rejW
2307
```

Rys. d: Test 4

```
# echo "5" > rejS
sh: write error: Input/output error
# echo "5" > rejW
sh: write error: Input/output error
# echo "-1" > rejA
[  142.315239] Wrong input, not octal.
sh: write error: Invalid argument
# echo "0" > rejA
[  147.099673] Octal value out of range.
sh: write error: Invalid argument
# echo "8" > rejA
[  152.764617] Wrong input, not octal.
sh: write error: Invalid argument
# echo "159" > rejA
[  159.616190] Wrong input, not octal.
sh: write error: Invalid argument
# echo "1751" > rejA
[  167.281321] Octal value out of range.
sh: write error: Invalid argument
```

Rys. e: Test 5

Rys. 14: Testy przeprowadzone z wykorzystaniem poleceń `echo` oraz `cat`

Poniżej przedstawiono opisy analizujące wyniki testów z rysunku 14.

1. Test 1 potwierdził poprawne utworzenie w katalogu '`/proc`' katalogu '`/proj4srekac`', a w nim plików `rejA`, `rejS` oraz `rejW`. Wartości natomiast poprawnie zostały ustawione na 0.
2. Test 2 potwierdził poprawny zapis do `rejA` oraz odczyt z `rejS` i `rejW`. Poprawnie została obliczona 05-ta liczba pierwsza: 013.
3. Test 3 potwierdził poprawność działania dla większej liczby. Z `rejW` odczytano poprawną wartość 045 liczby pierwszej: 0235.
4. Test 4 potwierdził poprawność znalezienia liczby pierwszej dla dużego argumentu (0310). Poprawnie znaleziono wartość 0310 liczby pierwszej: 02307.
5. Test 5 potwierdził poprawność pojawiania się błędów w niedozwolonych przypadkach testowych. Zapis wartości do `rejS` i `rejW` nie jest możliwy. Próba zapisania do `rejA` wartości niedodatniej lub większej niż 01750 (decymalnie 1000) również kończy się błędem. Zapisanie do `rejA` wartości zawierającej znak inny niż liczby całkowite z zakresu [0,7] (taka wartość jest niezgodna z systemem oktalnym) również zostaje blokowane i wyświetlany jest stosowny komunikat.

5. Aplikacja testująca

Na potrzeby weryfikacji poprawności działania całości systemu przygotowano testującą aplikację użytkownika.

5.1. Implementacja aplikacji

W aplikacji zaimplementowano funkcje `read_file()` oraz `write_file()`, odpowiedzialne za odpowiednio odczyt i zapis wartości. Otwierane i zamykane są odpowiednie pliki, na których przeprowadzane mogą

być operacje odczytu i zapisu. Funkcje te zostały zaimplementowane na wzór funkcji przedstawionych w instrukcji do projektu, gdzie zostały opisane (a szczególnie współpraca pomiędzy modułem jądra a aplikacją).

Oprócz wspomnianych funkcji przygotowano inne funkcje w języku C, wykorzystywane w testach (m. in. funkcja `wait_until_ready()` cyklicznie odczytująca wartość statusu z `rejS` i porównująca go z wartością zapisaną do `rejA`). Również w języku C zdefiniowano przypadki testowe, w których weryfikowana jest poprawność działania systemu. Zostały one opisane w kolejnej sekcji.

Aplikacja kompiluje się pomyślnie.

```
sykt@deb4sykom14:~/moje_prace/Srednicki_Kacper/projekt$ make_busybox_compile test/main.c
Changing place...
Checking contents of source files...
Removing previous compilation...
Preparing new compilation...
Module compilation in progress...
mke2fs 1.45.6 (20-Mar-2020)
Module compilation done.
Test application compilation...
Compiling the /home/sykt/moje_prace/Srednicki_Kacper/projekt/test/main.c file...
Install /home/sykt/moje_prace/Srednicki_Kacper/projekt/test/main file on destionation FS...
RootFS re-compilation in progress...
mke2fs 1.45.6 (20-Mar-2020)
RootFS re-compilation done (the '/home/sykt/moje_prace/Srednicki_Kacper/projekt/test/main' - product of compilation process is placed in /root of the target
 FS).
Done, files: fw_jump.elf, Image, rootfs.ext2 are ready to use.
```

Rys. 15: Kompilacja aplikacji

5.2. Testy z wykorzystaniem aplikacji

5.2.1. Test zapisu i odczytu do/z `rejA`

Przetestowano zapis wartości 07 do `rejA`, a następnie odczytano tę wartość. Wszystko przebiegło pomyślnie.

```
----------------------TEST1----------------------
Test: Writing and reading rejA.
Writing value 07 to rejA...
Reading from rejA...
Value: 07
Test PASSED!
```

Rys. 16: Poprawny zapis i odczyt do/z `rejA`

5.2.2. Testy znalezienia liczby pierwszej i odczytu z `rejW`

Następnie przetestowano znalezienie trzech różnych wartości argumentu zapisywanych do `rejA`. Następnie po zakończeniu poszukiwania, odczytywana była wartość z `rejW` i porównywana z wartością z `rejA`. Każdy z testów udokumentowanych na rysunkach 17, 18 oraz 19 przebiegł pomyślnie.

```
----------------------TEST2----------------------
Test: Searching for 017th prime number.
Writing value 017 to rejA...
Searching...
Ready, reading result from rejW...
Value: 057 (should be 057).
Test PASSED!
```

Rys. 17: Znalezienie 017 liczby pierwszej: 057

```
----------------------TEST3----------------------
Test: Searching for 034th prime number.
Writing value 034 to rejA...
Searching...
Ready, reading result from rejW...
Value: 0153 (should be 0153).
Test PASSED!
```

Rys. 18: Znalezienie 034 liczby pierwszej: 0153

```
----------------------TEST4-----------------------
Test: Searching for 071th prime number.
Writing value 071 to rejA...
Searching...
Ready, reading result from rejW...
Value: 0415 (should be 0415).
Test PASSED!
```

Rys. 19: Znalezienie 071 liczby pierwszej: 0415

5.3. Test odczytu statusu z `rejS`

Przeprowadzono także test odczytu statusu, wyświetlając jego wartość cyklicznie użytkownikowi. Test uznano za pomyślny, gdy wartość statusu zrównała się z wartością argumentu. Widoczna jest także informacja o pomyślnym ukończeniu wszystkich testów.

```
----------------------TEST5-----------------------
Test: Verifying the status.
Writing value 034 to rejA...
Searching and checking status at rejS...
Current status: 0th prime number.
Current status: 014th prime number.
Current status: 022th prime number.
Current status: 026th prime number.
Current status: 031th prime number.
Current status: 034th prime number.
Ready, status (S) is 034th prime number, same as argument (A).
Test PASSED!

----------------------SUMMARY----------------------
All tests completed: 5/5 successful.
```

Rys. 20: Cykliczny odczyt statusu

5.4. Test błędów

Przetestowano również wyrzucanie przez system błędów w niedozwolonych przypadkach testowych. System zgodnie z oczekiwaniami nie zezwolił na zapis wartości do `rejS` oraz `rejW`. Zwrócone zostały stosowne błędy, gdy podjęto próbę zapisu do `rejA` błędnej wartości (czyli takiej spoza zakresu [01,01750]).

```
Also bonus error tests in progress...

--------------------TEST_ERRORS--------------------
Writing value 04 to rejS...
Error, wrong nuber if bytes.

Writing value 04 to rejW...
Error, wrong nuber if bytes.

Writing value -1 to rejA...
[   91.743214] Octal value out of range.
Error, wrong nuber if bytes.

Writing value 0 to rejA...
[   91.751365] Octal value out of range.
Error, wrong nuber if bytes.

Writing value 01751 to rejA...
[   91.758851] Octal value out of range.
Error, wrong nuber if bytes.
Error tests FINISHED!
```

Rys. 21: Test błędów

6. Podsumowanie

Jestem zadowolony z realizacji projektu.