Politechnika Warszawska
Wydział Elektroniki I Technik Informacyjnych
Instytut Telekomunikacji

Systemy Komputerowe (SYKOM)

Projekt

Temat: 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.

Warunki wstępne:
Zapoznać się z:

- Metodami tworzenia sterowników dla systemu Linux.
- Współdziałaniem aplikacji użytkowych z sterownikami systemu Linux.
- Tworzeniem dystrybucji systemu Linux z wykorzystaniem systemu Buildroot.
- Materiałami wprowadzającymi dla laboratoriów 1, 2 i 3.

1. Przygotowanie środowiska do pracy

Dla celów zajęć projektowych przygotowano zestaw maszyn wirtualnych działających pod systemem Linux Debian. W komputerach tych zainstalowano odpowiednie narzędzia dla procesora RISC-V w wersji RV32I, odpowiednie pliki nagłówkowe oraz podsystem budowania dystrybucji systemu Linux. Aby mieć dostęp do jednej z tych maszyn należy dokonać rezerwacji z wykorzystaniem tzw. Systemu Rezerwacji Zasobów, który jest dostępny na komputerze o adresie: zsutresv.tele.pw.edu.pl.

Właściwą rezerwacje realizuje się poprzez przeglądarkę WWW zainstalowaną na swoim komputerze i używając serwowanej przez ten system rezerwacji strony (nazywany z ang. ResourceReservation):

```
http://zsutresv.tele.pw.edu.pl/ResourceReservation/
```

W przeciwieństwie do zajęć laboratoryjnych maksymalnym kwantem czasowym długości slotu są 2h. Wynika to głównie z innego charakteru wykonywanych zadań. Proszę pamiętać, że po zarezerwowanym dla siebie slocie czasowym, stan każdej z maszyn jest przywracany do stanu oryginalnego a wszystkie dane w niej składowane ulegają automatycznemu usunięciu. Aby jednak student miał możliwość pracy ze swoimi plikami (efektami swojej pracy) wytworzonymi na maszynie wirtualnej, przygotowano indywidualne i dedykowane każdemu studentowi repozytorium GIT[1][2]. Dzięki niemu tuż po rozpoczęciu pracy na maszynie wirtualnej, student powinien klonować swoje pliki z repozytorium, a przed zakończeniem pracy na maszynie wirtualnej powinien zatwierdzić w lokalnej kopii repozytorium GIT swoje poprawki (git add ; git commit...), następnie wypchnąć je na dedykowany zdalne repozytorium GIT przydzielone studentowi.

2. Utworzenie przydzielonego modułu w języku opisu sprzętu Verilog

W ramach zajęć projektowych studenci będą tworzyć swój system operacyjny dla wyspecyfikowanego i własnoręcznie zmodyfikowanego zestawu sprzętowego (hardware). Podstawą sprzętową podobnie jak w laboratorium 1 i 3 będzie emulator QEMU[3]. Szczegóły funkcjonalności a dokładniej opis budowy i funkcjonalności modułu dodawanego do tworzonego systemu (np.: adres w przestrzeni I/O, wielkość zajętej przestrzeni I/O, typ i algorytm implementowanej funkcjonalności, itp.) będą opisane w specyfikacji zadania projektowego, która to będzie przydzielona imiennie studentom. Szczegóły metody modyfikacji i tworzenia modułu oraz tworzenia emulatora QEMU zostały opisane w dokumentacji dla laboratorium 3.

3. System Linux i nowy sprzęt

3.1. Przygotowanie modułu dla jądra systemu Linux

Na zmodyfikowanym zgodnie z pkt.2 emulatorze QEMU będzie w następnym kroku uruchamiany system Linux. Aby system ten mógł współpracować z nowym sprzętem w jednym z najczęściej zalecanych podejść jest utworzenie tzw. modułów jądra.

Moduł jądra systemu Linux[4], to element oprogramowania umożliwiający m.in. współpracę z peryferiami. To dzięki niemu możliwe będzie komunikowanie się z tworzonym w Verilog elementem utworzonym w ramach pkt.2. Aby utworzyć moduł jądra systemu Linux należy utworzyć plik źródłowy w języku C o ściśle zdefiniowanej konstrukcji. Plik ten dla potrzeb używanego w dalszej części zestawu narzędzi musi mieć nazwę `kernel_module.c`:

```
1.  #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/ioport.h>
4.  #include <asm/errno.h>
5.  #include <asm/io.h>

6.  MODULE_INFO(intree, "Y");
7.  MODULE_LICENSE("GPL");
8.  MODULE_AUTHOR("Aleksander Pruszkowski");
9.  MODULE_DESCRIPTION("Simple kernel module for SYKOM lecture");
10. MODULE_VERSION("0.01");

11. #define SYKT_GPIO_BASE_ADDR  (0x00100000)
12. #define SYKT_GPIO_SIZE       (0x8000)
13. #define SYKT_EXIT            (0x3333)
14. #define SYKT_EXIT_CODE       (0x7F)

15. void __iomem *baseptr;

16. int my_init_module(void){
17.    printk(KERN_INFO "Init my sykt module.\n");
18.    baseptr=ioremap(SYKT_GPIO_BASE_ADDR, SYKT_GPIO_SIZE);
19.    return 0;
20. }

21. void my_cleanup_module(void){
22.    printk(KERN_INFO "Cleanup my sykt module.\n");
23.    writel(SYKT_EXIT | ((SYKT_EXIT_CODE)<<16), baseptr);
24.    iounmap(baseptr);
25. }

26. module_init(my_init_module)
27. module_exit(my_cleanup_module)
```

Jest to moduł, który po załadowaniu (polecenie: `modprobe kernel_module`) w logach systemu (dostępnych za pomocą polecenia: `dmesg`) wypisze komunikat „`Init my sykt module`” (linia 17). Natomiast podczas usuwania (polecenie: `rmmod kernel_module`) wypisze komunikat „`Cleanup my sykt module`” (linia 22) a następnie za pomocą linii 23 w której następuje zapis do rejestrów sterujących emulowanego CPU (patrz lab.1, implementacja `my_simulation_exit()`) nastąpi wywołanie wewnętrznej procedury zakończenia pracy emulatora QEMU – mechanizm dodany do emulatora, a tutaj stanowiący przykład komunikacji z peryferiami mapowanymi w przestrzeni We/Wy. Z punktu widzenia systemu goszczącego (tam gdzie uruchamiamy emulator QEMU) przekazany zostanie jemu kod wyjścia emulatora i będzie on równy 127 (0x7F).

W liniach 11-12 widzimy definicje informującą gdzie peryferia We/Wy zostały umieszczone w przestrzeni pamięciowej emulowanej maszyny. W linii 15 mamy deklaracje zmiennej `baseptr`, za pomocą której dzięki funkcji systemu operacyjnego `ioremap()` (linii 18), moduł jądra będzie mógł się komunikować ze zmapowanymi w przestrzeni pamięciowej funkcjonalnościami zmodyfikowanego emulatora (opisanymi w języku Verilog).

Powyższy mechanizm jest dość złożony i wymaga drobnego wyjaśnienia. W wielu systemach operacyjnych wykorzystujących mechanizmy ochrony pamięci i translacji adresów. W takich środowiskach nie jest trywialnym zadaniem odwołanie się do komórek pamięciowych pod określonym adresem. Dopiero korzystając z funkcji `ioremap()` do kodu modułu system operacyjny

przekazuje wirtualny adres pod jakim umieszczono peryferia. Dzięki tym operacjom można korzystając np.: z funkcji `writel()` i `readl()` odpowiednio pisać lub czytać z określonych miejsc w pamięci fizycznej i w szczególności odwoływać się do nowo utworzonych zasobów opisanych w module `GpioEmu`.

3.2. Kompilacja nowego modułu jądra i tworzenie finalnego systemu plików

Dla celów realizacji projektu na przygotowanych dedykowanych maszynach wirtualnych działających pod systemem Linux, utworzono wstępnie przygotowane środowisko BuildRoot[5]. Tworzenie od podstaw jądra, systemu plików, modułów oraz aplikacji użytkowych bardzo mocno wspiera system Buildroot. Niestety dla swojej pracy wymaga bardzo wielu zasobów i jego praca jest procedurą dość czasochłonną (wymagająca w zależności od sprzętu ponad 6h pracy). Dlatego przygotowując laboratoria i zadania projektowe, autor opracował dodatkowe narzędzia i wstępnie spreparował odpowiednie zestawy plików na maszynach wirtualnych działających pod systemem Linux Debian a udostępnianych studentom. Wśród tych narzędzi jest program `make_busybox_kernel_module` służący kompilacji modułów jądra, kopiowania ich binariów do docelowego systemu plików.

Dzięki temu narzędziu najbardziej pracochłonny pierwszy przebieg narzędzia BuildRoot nie jest potrzebny i czas kompilacji nowo tworzonego modułu został mocno skrócony, pozwalając użytkownikowi skupić swoją uwagę wyłącznie na tworzeniu oprogramowania modułów jądra i pracach wiązanych z jego uruchoamianiem.

Aby program mógł poprawnie pracować należy utworzyć pusty katalog np.: `/home/sykt/temp` oraz utworzyć w nim katalog np.: `kernel_module` a następnie wypełnić go następującymi elementami:

```
/home/sykt/temp/kernel_module/Config.in
/home/sykt/temp/kernel_module/kernel_module.mk
/home/sykt/temp/kernel_module/src/Makefile
/home/sykt/temp/kernel_module/src/kernel_module.c
```

Gdzie pliki mają następującą treść – a)konfiguracja - `/home/sykt/temp/kernel_module/Config.in`:

```
config BR2_PACKAGE_KERNEL_MODULE
        bool "kernel_module"
        help
                Kernel module module for sykom lecture.

                http://example.com/
```

Stanowiący opis modułu Buildroot – tu zawiera on wyłącznie kod moduł jądra.

b)dodatkowe info. dla programu MAKE - `/home/sykt/temp/kernel_module/kernel_module.mk`:

```
KERNEL_MODULE_VERSION = 1.0
KERNEL_MODULE_SITE = ./package/kernel_module/src
KERNEL_MODULE_SITE_METHOD = local

define KERNEL_MODULE_BUILD_CMDS
      #$(MAKE) -C '$(@D)' LINUX_DIR='$(LINUX_DIR)' PWD='$(@D)' CC='$(TARGET_CC)' LD='$(TARGET_LD)'
endef

define KERNEL_MODULE_INSTALL_TARGET_CMDS
      #$(INSTALL) -D -m 0755 '$(@D)/kernel_module.ko' '$(TARGET_DIR)/kernel_module.ko'
endef

$(eval $(kernel-module))
$(eval $(generic-package))
```

c)właściwy plik makefile - `/home/sykt/temp/kernel_module/src/Makefile`

```
obj-m += kernel_module.o
ccflags-y := -Wno-declaration-after-statement -std=gnu99

default: modules

modules:
        $(MAKE) -C '$(LINUX_DIR)' M='$(PWD)' modules
clean:
        $(MAKE) -C '$(LINUX_DIR)' M='$(PWD)' clean
```

d)właściwa implementacja modułu jądra: `/home/sykt/temp/kernel_module/src/kernel_module.c`, jej treść podano w pkt.3.1.
W wyniku swojej pracy narzędzie `make_busybox_kernel_module` o ile nie pojawią się błędy kompilacji, wytworzy pliki: `fw_jump.elf` - tzw. bootloader, `Image` - obraz jądra nowo utworzonego systemu, `rootfs.ext2` - tzw. root file system, a w nim umieszczony zostanie plik `kernel_module.ko` w emulowanym systemie plików: `/lib/modules/5.1.12/extra/`.

W przypadku pojawienia się problemów z działaniem powyższego programu, należy zajrzeć do tworzonego podczas jego pracy pliku: compilation_log.txt i po jego analizie próbować naprawić raportowane tam błędy.

4. Uruchomienie utworzonego systemu Linux i test utworzonego modułu jądra z użyciem emulatora QEMU

Przed przystąpieniem do następnych kroków (np.: uruchamiania aplikacji demonstracyjnej) warto sprawdzić czy nowy system będący procesorem Risc-V z nowymi peryferiami będzie poprawnie działał z nowym systemem operacyjnym. Zakłada się, że procesor z nowym otoczeniem utworzono zgodnie z opisem dla laboratorium 2 i 3 i odpowiednie binaria emulujące go nazywają się: `qemu-system-riscv32-sykt`.
Aby uruchomić emulacje na takim nowym procesorze, należy uruchomić następujące polecenie:

```
./qemu-system-riscv32-sykt -M sykt -nographic                    \
-bios fw_jump.elf                                                \
-kernel Image                                                    \
-append "root=/dev/vda ro"                                       \
-drive file=rootfs.ext2,format=raw,id=hd0                        \
-device virtio-blk-device,drive=hd0                              \
-netdev user,id=net0 -device virtio-net-device,netdev=net0
```

Proszę zwrócić uwagę w powyższym wywołaniu na każdy znak – np.: wstawienie spacji w dowolnym miejscu może uniemożliwić poprawny start emulacji. Dodatkowo uwagi wymaga znak '\' na końcach linii powyższego wywołania. Znak ten generalnie wskazuje powłoce systemu Linux, że zapis polecenia jest kontynuowany w kolejnych liniach. Co ważne po znaku '\' nie można wstawić żadnego innego znaku poza znakiem końca linii.
Po uruchomieniu zgodnie z powyższą instrukcja emulatora QEMU, nowo utworzony system Linux rozpocznie pracę. Gdy zakończy się jego tzw. proces rozruchu (tzw. boot process), użytkownikowi ukaże się komunikat o konieczności zalogowania się („buildroot login"):

```
[    1.090180] Key type dns_resolver registered
[    1.125041] EXT4-fs (vda): mounting ext2 file system using the ext4 subsystem
[    1.146275] EXT4-fs (vda): mounted filesystem without journal. Opts: (null)
[    1.147797] VFS: Mounted root (ext2 filesystem) readonly on device 254:0.
[    1.155002] devtmpfs: mounted
[    1.218951] Freeing unused kernel memory: 172K
[    1.219708] This architecture does not have kernel memory protection.
[    1.220831] Run /sbin/init as init process
[    1.471993] EXT4-fs (vda): warning: mounting unchecked fs, running e2fsck is recommended
[    1.483141] EXT4-fs (vda): re-mounted. Opts: (null)
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Initializing random number generator: OK
Saving random seed: [    2.608039] random: dd: uninitialized urandom read (512 bytes read)
OK
Starting network: udhcpc: started, v1.31.1
udhcpc: sending discover
udhcpc: sending select for 10.0.2.15
udhcpc: lease of 10.0.2.15 obtained, lease time 86400
deleting routers
adding dns 10.0.2.3
OK
Starting telnetd: OK

Welcome to Buildroot
buildroot login:
```

Na tym etapie emulowany system jest gotowy do pracy i po wpisaniu po komunikacie „Buildroot login:" słowa 'root', system zaloguje nas i pojawi się nam znak zachęty „#", po którym powłoka oczekiwać będzie na polecenia. Dla sprawdzenia czy został poprawnie skompilowany nowo utworzony moduł jądra można go załadować za pomocą polecenia:

```
modprobe kernel_module
```

Aby zobaczyć efekt uruchamiamy polecenie 'dmesg' a na końcu raportu przez niego wygenerowanego powinniśmy zobaczyć linię o podobnej treści:

```
[  274.849738] Init my sykt module!
```

Aby usunąć moduł jądra należy wykonać polecanie:

```
rmmod kernel_module
```

Po wydaniu tego polecenia zgodnie z zapisem funkcji `my_cleanup_module()` emulator QEMU zakończy swoje działanie – co w tym przypadku jest działaniem prawidłowy, na ekranie powinno ukazać się:

```
[  334.308105] Cleanup my sykt module.
qemu-system-riscv32: info: sykt_test_write(addr=0x0, val64=0x7f3333, size=0x4) - FINISHER_FAIL
qemu-system-riscv32: info: Qemu for SYKT finish (131229)
```

Po takiej operacji emulator QEMU zakończy emulowanie i powinien zwrócić kod błędu 0x7F (127) przekazany przez usuwany moduł jądra (co tutaj jest poprawne). W pewnych okolicznościach po takim zakończeniu pracy emulatora QEMU istnieje konieczność uruchomienia w linii poleceń polecenia 'reset'. Konieczność taka wynika z niewłaściwego odtworzenia ustawień terminala przez emulator QEMU.

5. Testy demonstrujące działania całościowego nowo utworzonego systemu

Po utworzeniu nowego systemu i współpracującego z nim modułu jądra, w praktycznych warunkach należy przetestować działanie całości i współdziałanie nowego systemu z aplikacjami użytkownika.

5.1. Testowanie z wykorzystaniem narzędzi systemowych

Narzędzie Buildroot (opisane w pkt.3) pomaga tworzyć aplikacje użytkowe. Są one pomocne w użytkowaniu nowo utworzonego systemu operacyjnego. Mimo, że są to generyczne narzędzia za ich pomocą można zweryfikować faktyczną poprawności działania wielu elementów systemu, w tym funkcjonowanie nowej implementacji modułu jądra.

Dla potrzeb zajęć projektowych zgodnie z specyfikacją zdania, utworzone moduły jądra muszą komunikować się z aplikacjami użytkownika. Wśród wielu różnych mechanizmów mamy odwołania z wykorzystaniem klasycznych urządzeń, których elementem styku są pliki w katalogu `/dev` systemu Linux. Można także komunikować się z aplikacjami użytkownika za pomocą systemów plików: `PROC-FS`[6] lub `SYS-FS`[7]. Szczegóły budowy modułów jądra (w tym wycinki kodów źródłowych) korzystających z tych systemów plików będą ukazane na zajęciach wykładowych.

Dla ustalenia uwagi załóżmy, że po załadowaniu modułu jądra pojawiają się nowe pliki w katalogu `/sys` obsługiwanym przez `SYS-FS` a wśród nich plik(a):

```
/sys/kernel/sykt_sysfs/sykt_gpio_out
```

Dodatkowo załóżmy, że implementacja tworzonego dla zajęć projektowych modułu jądra zapewnia funkcjonalność w której cokolwiek zostanie wpisane w postaci tekstowej (w notacji dziesiętnej) zostanie po konwersji na postać bitową poprzez odpowiednie peryferia emulowanego procesora wpisane do wyjścia GPIO i w rezultacie pokazane zostanie poprzez konsolę GpioEmuConsole (w polu „GPIO wyjściowe"). W analogiczny sposób gdy w tej konsoli zostanie zmodyfikowany w polu „GPIO wejściowe" którykolwiek bit, stan tego pola w całości zostanie przekazany do emulowanego systemu i pobrany podczas próby odczytania treści pliku(b):

```
/sys/kernel/sykt_sysfs/sykt_gpio_in
```

Za sprawą modułu jądra ta wartość zostanie pobrana z odpowiednich zmapowanych peryferii i przekazana w postaci tekstowej (podobnie w notacji dziesiętnej). Budzić się może pytanie - ale jak

przekazać treść do/z tych plików? W przypadku interakcji (tu przekazania wartości ‘123’) z plikiem (a) wystarczy wydać polecenie:

```
echo ”123” > /sys/kernel/sykt_sysfs/sykt_gpio_out
```

A dla odczytania zawartości drugiego pliku(b) - z polecenia:

```
cat < /sys/kernel/sykt_sysfs/sykt_gpio_in
```

Oba polecenia (‘`echo`’ i ‘`cat`’) są wbudowane w `rootfs.ext2`, wytworzone przez Buildroot.

5.2. Testowanie z wykorzystaniem własnej aplikacji

Testowanie opisane w poprzednim rozdziale jest poprawne, lecz ze swojej natury może być nieco uciążliwe. Szczególnie jest to dotkliwe gdy należy dla testów wykonać wiele operacji. Najwygodniej utworzyć własną aplikację w jakimś języku wyższego poziomu np.: C. Niestety utworzony prze Buildroot nowy systemu (opisany w rozdziale 3) nie posiada żadnego kompilatora ani interpretera takowego języka. Aby móc jednak tworzyć aplikacje można w systemie macierzystym (z tego systemu uruchamiamy emulator QEMU) użyć polecenie kompilacji skrośnej: `make_busybox_compile`. Polecenie to w odróżnieniu od `make_busybox_kernel_module`, oprócz kompilacji modułu jądra dokonuje także kompilacji aplikacji w języku C i wytworzy ponownie plik `rootfs.ext2`.

Aplikacja `make_busybox_compile`, wywoływana jest w katalogu w którym znajduje się kompilowany moduł jądra oraz dodatkowo oczekuje parametru jakim jest nazwa pliku z kodem nowo dodawanej aplikacji zaimplementowanej w języku C. Jeżeli takowego pliku nie znajdzie – utworzy domyślną prostą aplikacje testową. Która po uruchomieniu - polecenie `./main` w nowo utworzonym systemie Linux - wyświetli komunikat podobny do:

```
Compiled at Nov 08 2021 09:53:14
```

Aplikacje tę należy traktować jako wstęp do tworzenia własnego kodu. Proszę zwrócić uwagę że podczas swojego działania podaje ona datę kompilacji – pomaga w ten sposób zorientować się czy uruchamiamy właściwy firmware.

Nadal może pojawiać się pytanie jak aplikacja ma komunikować się z odpowiednimi urządzeniami implementowanymi przez moduły jądra zbudowanego systemu. Przykład prostej aplikacji dla nowo utworzonego systemu Linux można zapisać następująco (kod zapisano w pliku main.c):

```
1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <sys/stat.h>
4. #include <fcntl.h>
5. #include <unistd.h>
6. #include <errno.h>
7. #define MAX_BUFFER                  1024
8. #define SYSFS_FILE_NAME_IN          "/sys/kernel/sykt_sysfs/sykt_gpio_in”
9. #define SYSFS_FILE_NAME_OUT         "/sys/kernel/sykt_sysfs/sykt_gpio_out”
10. int main(void){
11.   char buffer[MAX_BUFFER];
12.   int i;
13.   int fd_in=open(SYSFS_FILE_NAME_IN, O_RDWR);    //otwarcie pliku „in” sterownika
14.   if(fd_in<0){
15.      printf("Open %s – error: %d\n", SYSFS_FILE_NAME_IN, errno);
16.      exit(1);
17.   }
18.   int fd_out=open(SYSFS_FILE_NAME_OUT, O_RD);    //otwarcie pliku „out” sterownika
19.   if(fd_out<0){
20.      printf("Open %s – error: %d\n", SYSFS_FILE_NAME_OUT, errno);
21.      close(fd_in);                  //nikt inny nie zamknie wcześniej otwartego pliku
22.      exit(2);
23.   }
24.   i=123;
25.   snprintf(buffer, MAX_BUFFER, "%d", i);         //treść do przekazania sterownikowi
26.   int n=write(fd_in, buffer, strlen(buffer));    //właściwe przekazanie danych
27.   if(n!=strlen(buffer)){
28.       //...obsluga bledu
29.   }
30.   int n=read(fd_out, buffer, MAX_BUFFER);        //odczyt danych ze sterownika
```

```
31.   if(n>0){
32.       buffer[n]='\0';
33.       printf("%s\n", buffer);
34.   }else{
35.       //...obsługa błędu
36.   }
37.   close(fd_in);
38.   close(fd_out);
39.   return 0;
40. }
```

Działanie powyższej aplikacji jest niemal identyczne jak przykłady z użyciem narzędzi systemu Linux pokazanymi w pkt. 5.1. Proszę zauważyć, że powyższy kod nie sprawdza czy moduł jądra został poprawnie załadowany i powstały odpowiednie pliki. Podczas swojej pracy kod tej aplikacji w liniach 13 i 18 otwiera zdefiniowane w liniach 8 i 9 pliki. Ich nazwy wynikają z budowy modułu jądra przedstawionego jako przykład na zajęciach wykładowych. Na tym etapie ważne jest aby po otwarciu każdego z plików współpracy z modułem jądra sprawdzić poprawność operacji (odpowiednio linie 14-17 i 20-23).

Współpraca z plikiem „wyjściowym" jest prosta i opisują ją linie 24-29. W liniach 24-25 preparowana jest tekstowa zawartość która finalnie zostanie przekazana do sterownika – za co odpowiada linia 26 – gdzie wywoływana jest funkcja systemowa `write()`. W linii 27 sprawdzana jest zwracaną przez funkcję `write()` wartość a umieszczona w zmiennej 'n' i porównywana z trzecim argumentem jej wywołania (wywołanie `strlen()` w linii 26), sprawdzając w ten sposób ile danych udało się wpisać do pliku. Gdyby liczby te były różne program powinien odpowiednio zareagować np.: wypisać właściwy komunikat - co można zapisać w linii 28.

Dla odmiany współpraca z plikiem wejściowym wymaga więcej uwagi. Po przeczytaniu danych przez funkcję `read()` – linia 30, należy koniecznie sprawdzić ile bajtów zostało faktycznie odczytanych a następnie wstawić (lina 32) we właściwym miejscu znak '\0' (kończy on łańcuch tekstowy). Jest to niezbędne gdyż jak wiadomo funkcja `printf()` (lina 33) oczekuje argumentu wpisanego do `buffer` a zakończonego właśnie takim znakiem.

Przed zakończeniem swojego działania, aplikacja powinna zwolnić wszystkie otrzymane zasoby – tutaj zamknąć wszystkie otwarte pliki. Podczas standardowej aktywności aplikacji zrealizują to wywołania funkcji `close()` w liniach 37 i 38.

Nie jest to jedyne miejsce gdzie pliki te muszą być zamknięte. Proszę zauważyć, że może zdarzyć się sytuacja gdy pomyślnie uda się otworzyć plik `SYSFS_FILE_NAME_IN` otwarty w linii 13, ale próba otwarcia pliku `SYSFS_FILE_NAME_OUT` zakończy się porażką (linia 18). W takim przypadku przed zakończeniem działania tej aplikacji trzeba oprócz reakcji na problem, także zamknąć plik `SYSFS_FILE_NAME_IN` (linia 21) a dopiero potem wyjść z błędem zwracając kod powrotu oznaczający błąd – tutaj 2 (linia 21).

Materiały dodatkowe:

1. https://git-scm.com/docs, ostatnia odsłona 2023.03.08
2. https://git-scm.com/book/pl/v2, ostatnia odsłona 2023.03.08
3. https://www.qemu.org/docs/master/, ostatnia odsłona 2023.03.08
4. Linux Device Drivers, Third Edition, https://www.oreilly.com/openbook/linuxdrive3/book/, ostatnia odsłona 2023.03.08
5. https://buildroot.org/downloads/manual/manual.html, ostatnia odsłona 2023.03.08
6. https://www.kernel.org/doc/html/latest/filesystems/proc.html, ostatnia odsłona 2023.03.08
7. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html, ostatnia odsłona 2023.03.08