

Systemy komputerowe: architektura i programowanie (SYKOM)

Wykład: System operacyjny dla systemów komputerowych

Aleksander Pruszkowski

Instytut Telekomunikacji Politechniki Warszawskiej

PLAN WYKŁADY

- Definicja, budowa i działanie systemu Linux
- Oprogramowanie w języku C/C++ dla systemu Linux
- Moduły jądra Linux - tworzenie, uruchamianie, testowanie

Definicja, budowa i działanie systemu Linux

Definicja, budowa i działanie systemu Linux

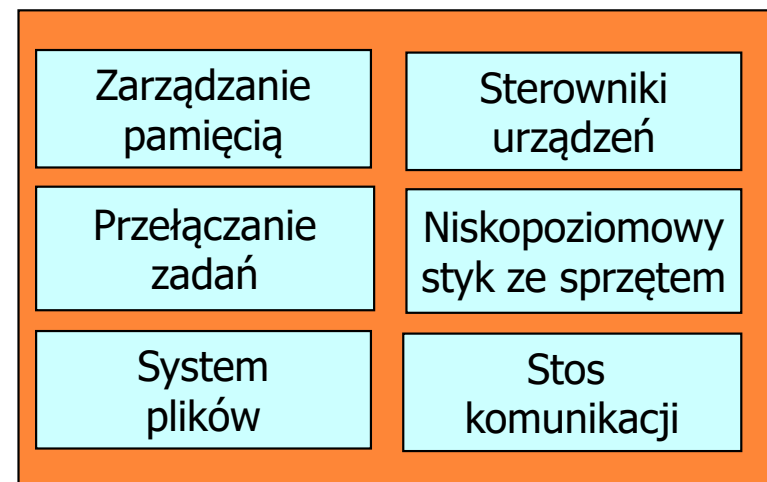
- Powstanie koncepcji otwartego oprogramowania: 1983 - Richard Stallman i jego GNU z darmowymi narzędziami m.in.: gcc, gdb, glibc, daje podstawy nowych projektów
- System operacyjny: 1991 - Linus Torvalds tworzy „Linux kernel” odpowiednik „Unix Kernel”, całkowicie darmowy^{*)} i otwarty
- System operacyjny Linux to
 - Monolityczne jądro i moduły
 - Zestaw bibliotek (np.: libgcc, glibc, ...) i narzędzi (np.: cp, mkdir, dd, ...)
 - Usługi i aplikacje (Init, Bash, ...)

^{*)} Tzw. firmware – zamknięty co do źródeł, binarny „wsad” związany ze sterownikami wymyka się otwartości

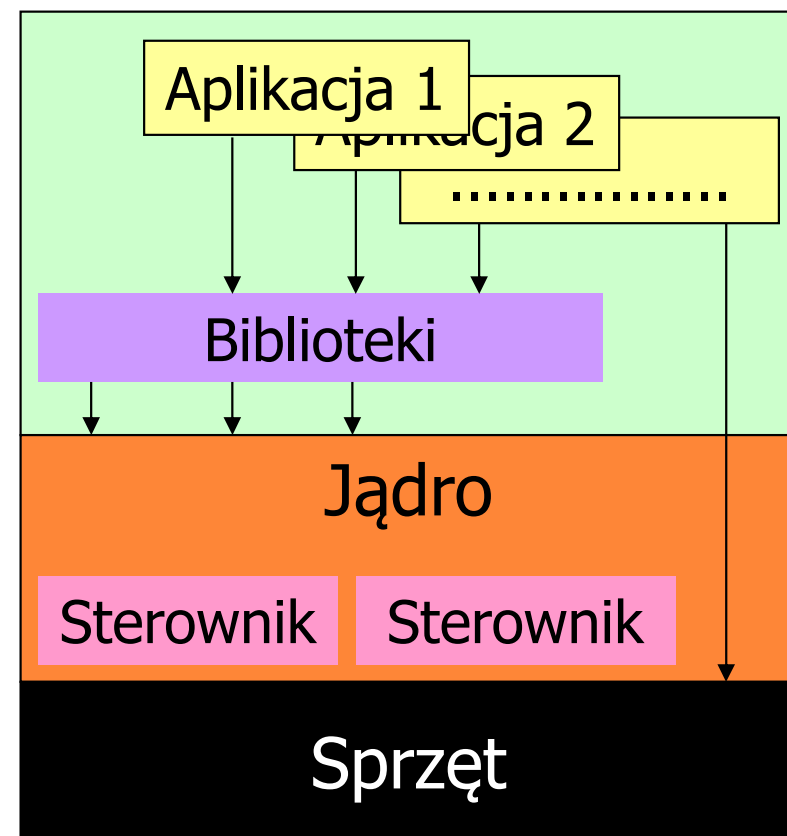
Definicja, budowa i działanie systemu Linux

■ Monolityczne jądro

- Podejście ułatwia dostosowanie jądra do własnych potrzeb - faza kompilacji
- Sterowniki czynią wyłom w monolityczności jądra
 - ułatwienie integracji systemu - zgodną z własnymi potrzebami - fazy użytkowania systemu
 - część problemów z systemem wynika z niemożności załadowania właściwych modułów jądra
 - jedno z rozwiązań - stosowane zwłaszcza w systemach wbudowanych - to skompilowanie wybranych modułów z obrazem jądra
- Możliwe jest dostosowanie jądra do własnych potrzeb
 - `make menuconfig` && `make` && `make install`
 - tworzą plik: `vmlinux`(bez kompresji)/`zImage`/`bzImage` i instalują je narzędziami programu rozruchowego



Jądro



Definicja, budowa i działanie systemu Linux

■ Jak zrobić własny „minimalistyczny Linux”

- Skonfiguruj i zbuduj obraz jądra (mając źródła jądra pobrane i rozpakowane)

```
make x86_64_defconfig
```

Ustawienie domyślnej konfiguracji kompilowanego jądra

```
cat <<EOF >.config-fragment  
CONFIG_BLK_DEV_RAM=y  
CONFIG_BLK_DEV_RAM_COUNT=16  
CONFIG_BLK_DEV_RAM_SIZE=4096  
EOF
```

Dodanie opcji związanych z tzw. ram dyskiem

```
./scripts/kconfig/merge_config.sh .config .config-fragment
```

Łączenie konfiguracji

```
make
```

Właściwa kompilacja

- Utwórz pseudo system operacyjny – dla przykładu tutaj zapisany jako jeden proces zapisując jego kod w pliku np.: main.c:

```
#include <stdio.h>  
  
int main(int argc, char *argv[]){  
    printf("Hello world (%s - %s)!\n", __DATE__, __TIME__);  
    for(;;){  
        printf("Hello world\n");sleep(1);  
    } //nie ma innych procesow - wiec nie możemy z Init „wyjść”  
    return 0;  
}
```

Definicja, budowa i działanie systemu Linux

■ Jak zrobić własny „minimalistyczny Linux”, cd.

- Plik main.c poddaj kompilacji do pliku init
 - Opcja `–static` wymagana, gdyż trzeba byłoby tę kompilację połączyć z bibliotekami nowego systemu operacyjnego, których jeszcze nie ma(!)

```
gcc -static main.c -o init
```

- Następnie utwórz główny system plików (root file system)

```
echo init | cpio -o -H newc | gzip > ram.cpio.gz
```

- Mając powyższe kroki - dla testowego sprawdzenia jak system działa można wykorzystać emulator QEMU

```
qemu-system-x86_64 -kernel bzImage -initrd ram.cpio.gz --append "root=/dev/ram0 init=/init"
```

- A otrzyma się na ekranie QEMU wynik to np.:

```
Hello world (Apr 26 2021 - 10:49:37)!
```

- System mimo trywialności pokazuje podstawy systemu Linux
 - Jeżeli potrzeba czegoś bardziej przydatnego - wystarczy tylko trochę pracy!!!

Definicja, budowa i działanie systemu Linux

■ Qemu – a co to jest?

■ Otwarto źródłowy emulator i wirtualizator maszyn

- Potrafi emulować wiele architektur - patrz obok:
- Współdziała z wieloma nadzorcami (KVM, Xen, ...)
- Zawiera wewnętrzny kompilator zamieniający instrukcje wirtualne na odpowiadający mu kod maszyny na jakiej się wykonuje
- Przykład użycia (emulacja systemu Linux dla Risc-V - przykład z zajęć projektowych, jedna długa linia)

```
./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
```

- fw_jump.elf - tzw. bootloader, umożliwia uruchomienie docelowego systemu
- Image - obraz jądra nowo utworzonego systemu
- rootfs.ext2 - tzw. root file system, z wszelkimi narzędziami systemowymi i użytkowymi

qemu-system-aarch64
qemu-system-alpha
qemu-system-arm
qemu-system-cris
qemu-system-ppc
qemu-system-i386
qemu-system-lm32
qemu-system-m68k
qemu-system-microblaze
qemu-system-mips
qemu-system-mips64
qemu-system-mipsel
qemu-system-moxie
qemu-system-nios2
qemu-system-or1k
qemu-system-ppc
qemu-system-ppc64
qemu-system-riscv32
qemu-system-riscv64
qemu-system-s390x
qemu-system-sh4
qemu-system-sh4eb
qemu-system-sparc
qemu-system-sparc64
qemu-system-tricore
qemu-system-unicore32
qemu-system-x86_64
qemu-system-xtensa
qemu-system-xtensaeb

Definicja, budowa i działanie systemu Linux

- W systemie Linux istnieje tylko jedno drzewo z plikami - „root file system”
 - Wszelkie nośniki są montowane do głównego drzewa
 - w przeciwieństwie do Windows gdzie mamy dyski: A:, B:, C:,
 - Jądro systemu po uruchomieniu dla poprawnego działania musi zamontować główny system plików a w nim znaleźć obraz głównego procesu Init (rodzica wszystkich procesów)
 - z jądrem linkowany jest (z reguły) specjalny zapasowy/awaryjny główny system plików 'initramfs'
 - można dzięki niemu zapewnić możliwość załadowania dodatkowych sterowników napędów (np.: do nietypowego sprzętu) aby z ich wykorzystaniem zamontować docelowy system plików w głównym systemie plików
 - Generalna filozofia systemów zgodnych z Unix - Linux to jedna z jego odmian - stanowi że „wszystko jest plikiem” („everything is a file”)
 - komunikacja z wszystkimi peryferiami powinna być realizowana za pomocą plików - co nie jest konsekwentnie utrzymywane

Definicja, budowa i działanie systemu Linux

- W systemie Linux istnieje tylko jedno drzewo z plikami - „root file system”, cd.
 - Struktura głównego systemu plików jest niemal standardowa
 - /bin - programy użytkowe
 - /dev - pliki urządzeń dostępnych w systemie
 - ich brak może wynikać ze złego zainicjowania jądra systemu lub braku odpowiednich sterowników
 - /etc - pliki konfiguracyjne dla całego systemu
 - /home - katalogi z plikami użytkowników
 - /lib - biblioteki i moduły jądra
 - **/proc** - wirtualny system plików (PROCFS)
 - zawiera: informacje o systemie, statystyki
 - pomaga dostrajać działanie systemu
 - np.: włączenie przekazywania pakietów, uruchamia się przez polecenie
`echo „1” > /proc/sys/net/ipv4/ip_forward`

Definicja, budowa i działanie systemu Linux

- W systemie Linux istnieje tylko jedno drzewo z plikami - „root file system”, cd.
 - Struktura głównego systemu plików jest niemal standardowa, cd.
 - /root - miejsce na katalogi i pliki dla administratora
 - /sbin - narzędzia specjalne (typowo nie dostępne dla nie administratorów)
 - **/sys** - wirtualny system plików wirtualny (sysfs)
 - nowszy konkurent PROCFS
 - umożliwia dostęp do wielu podsystemów komputera np.: za pomocą magistral I2C, 1W
 - /tmp - miejsce na dane tymczasowe
 - /usr - aplikacje usługowe – uwaga! od pewnego czasu tam trafia wiele aplikacji(!)
 - /var - dane zmienne, często istotne dla całego urządzenia (np.: strony www)
 - opcjonalnie zależnie od dystrybucji: /boot, /opt, /media, /mnt

Definicja, budowa i działanie systemu Linux

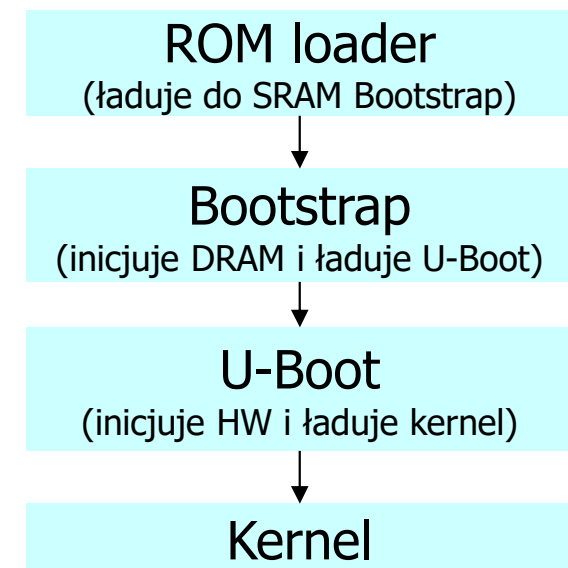
- Proces ładowania systemu Linux jest złożony i zaprojektowano go tak aby umożliwić start niemal każdej konfiguracji komputera/węzła
 - Zadanie ładowania wykonuje program rozruchowy 'Bootloader'
 - Bootloader (czasami jest to wiele mniejszych aplikacji) odpowiada za
 - inicjowanie podstawowego otoczenia sprzętowego w tym pamięci (np.: DRAM)
 - ładowanie, dekompresję i zapis obrazu jądra do pamięci danych (np.: z wbudowanej pamięci FLASH, z karty mikro-SD, poprzez sieć protokołem TFTP lub HTTP)
 - uruchomienie głównego procesu aplikacji - Init (/sbin/init)
 - aplikacja będąca rodzicem wszystkich innych aplikacji (w tym także uruchamia powłokę)
 - Znanych jest bardzo wiele programów rozruchowych
 - ogólnego przeznaczenia: GRUB, LiLo, Syslinux, EFI, OpenFirmware
 - najbardziej znany to GRUB (Grand Unified Bootloader), potrafi współpracować z wieloma systemami plików na których zapisano obraz jądra, umożliwia użytkownikowi konfigurację swojego działania - stosowany jest także w niektórych urządzeniach wbudowanych
 - dla systemów wbudowanych otwarte: U-Boot, RedBoot, CFE, zamknięte: Adam2, PSPBoot
 - Dominuje to U-Boot - umożliwia ładowanie systemu niemal z każdego nośnika

Definicja, budowa i działanie systemu Linux

■ Przebieg ładowania systemu wbudowanego - dwa podejścia

■ Ładowanie do RAM i uruchamianie z RAM

- podejście dominujące
- CPU wykonuje ze swojej pamięci ROM (o treści ustalonej przez producenta) kod ładowania obrazu standardowego programu rozruchowego np.: U-Boot, Grub
- uruchamia załadowany program rozruchowy - który kończy ładowanie systemu
- zaleta: uniwersalność podejścia, tanie komponenty (można użyć tanich i pojemnych pamięci NAND-FLASH/mikro-SD)
- wada: mała szybkość całego procesu ładowania - konieczność kopiowania obrazu



Definicja, budowa i działanie systemu Linux

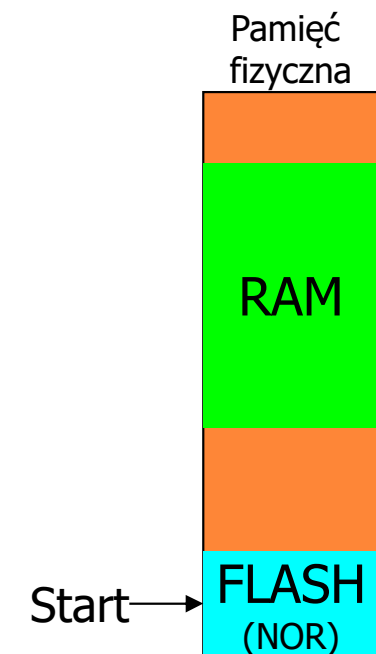
■ Przebieg ładowania systemu wbudowanego - dwa podejścia, cd.

■ Ładowanie z wykonaniem w miejscu (XIP)

- po włączeniu CPU rozpoczyna swoje działanie z określonego adresu swojej pamięci kodu - gdzie zapisano uproszczony system uruchamiania jądra
- zaleta: duża szybkość, brak kopiowania i dekompresji obrazu jądra
- wada: mała uniwersalność podejścia, wiele elementów trzeba projektować od zera, wymaga użycia jako pamięć kodu szybkich, drogich i mało

pojemnych pamięci NOR-FLASH, sprzęt musi wspierać ten mechanizm, aktualizacja obrazu jądra może być nie możliwa lub utrudniona („operacja na żywym mózgu”)

- mała pojemność pamięci FLASH uniemożliwia zapisanie w głównym systemie plików wielu aplikacji użytkowych
- podejście w którym zamontowano by dwa typy pamięci FLASH dla odpowiednio obszaru rozruchowego i dla systemu plików jest rzadko stosowane



Definicja, budowa i działanie systemu Linux

■ Konsola U-boot - przykład otrzymany po włączeniu platformy

```
U-Boot 2017.03 (March 28 2017 - 09:21:15)
```

```
CPU: SAME70
```

```
Crystal frequency: 16 MHz
```

```
CPU clock : 128 MHz
```

```
DRAM: 256 MiB
```

```
NAND: 256 MiB
```

```
In: serial
```

```
Out: serial
```

```
Err: serial
```

```
Hit any key to stop autoboot: 0
```

```
uboot> ?
```

```
?                - alias for 'help'
boot             - boot default - run command 'bootcmd'
bootd           - boot default, i.e., run 'bootcmd'
bootm           - boot application image from memory
cp              - memory copy
erase           - erase FLASH memory
md              - memory display
printenv        - print environment variables
run             - run commands in an environment variable
setenv          - set environment variables
...
```

Definicja, budowa i działanie systemu Linux

■ U-boot - rozbudowany program rozruchowy

- w źródłach jest wsparcie dla wielu platform sprzętowych
- zawiera mechanizmy inicjowania i testowania wielu typów pamięci wbudowanych w węzły
- potrafi obsługiwać wiele systemów plików przeznaczonych dla pamięci FLASH
 - Journaling Flash File System (JFFS/JFFS2)
 - Compressed ROM Filesystem (CramFS)
 - Specjalne systemy plików: Overlay File Systems (nakładka na inny system plików), Persistent RAM File system (przydatny dla zachowywania stanu pamięci RAM)
 - oraz generyczne: EXT2, FAT
- wspiera wiele interfejsów sieciowych oraz operacje sieciowe wysokiego poziomu
 - protokoły: BootP-client, Dhcp-client, HTTP-client, TFTP-client
- pozwala na definiowanie skryptów
 - są one pomocne przy automatyzowaniu procesu ładowania i eliminuje w wielu przypadkach konieczność rozszerzania kodu U-boot, np.:

```
setenv mmc-boot 'if fatload mmc 0 80000000 boot.ini; then source; else  
if fatload mmc 0 80000000 zImage; then run mmc-do-boot; fi; fi'
```


■ Rola interfejsu szeregowego w procesie uruchamiania platformy

- Dostarcza cenne informacje o przebiegu ładowania systemu
- Pozwala zmienić konfigurację rozruchu systemu (np. poprzez ustawienie zmiennych U-Boot)
- Mimo że port szeregowy jest mocno niedoceniony przez użytkowników PC, jest powszechnie stosowany w wielu urządzeniach wbudowanych
 - nawet gdy wspierają USB dla podstawowej konsoli to i tak stosują profil CDC odpowiadający komunikacji poprzez port szeregowy
- Dlaczego dostępne w Linux polecenie `'dmesg'` nie wystarczy?
 - dostarcza za mało danych - a zwłaszcza we wczesnej fazie rozruchowej, gdy jądro Linux nie zaczęło jeszcze pracy
 - ładowanie może się nie udać - wtedy nie będzie dostępu do polecenia `'dmesg'` - wtedy log przedstawiony portem szeregowym jest nieoceniony
 - należy pamiętać, że interfejs szeregowy wyprowadzony na płytach systemów wbudowanych (w tym IoT) dostarcza z reguły napięcia UART-TTL (0V...3,3V/5V) a interfejs szeregowy V.24 (RS232) akceptuje i dostarcza napięcia sygnałowe w zakresie -12V...+12V - połączenie obu typów tych sygnałów może uszkodzić platformę wbudowaną

Oprogramowanie w języku C/C++ dla systemu Linux

Oprogramowanie w języku C/C++ dla systemu Linux

■ Linux - kompilacja plików źródłowych

- Linux ma jądro zbudowane z plików C
 - do ich kompilacji z reguły stosowany jest kompilator GCC
- Istnieje silny związek między narzędziami użytymi do kompilacji jądra i aplikacji mających pracować w środowisku tego jądra
 - konsekwencje - dla każdej dystrybucji (czy jej odmiany) do wszelkich późniejszych prac programistycznych potrzeby jest komplet: kompilator (np.: GCC+Binutils), zestaw plików nagłówkowych i biblioteki dla danego jądra
 - często trudno ominąć tę zależność
 - pomocna może być treść pliku `'config'` i inspekcja działającego systemu - `'config'` dostępny w: `/proc/config.gz` lub `/boot/config...`

■ Linux - kompilacja plików źródłowych, cd.

- W systemach wbudowanych często korzysta się z kompilacji skróśnej (cross-compile)

- takie podejście przyspiesza proces kompilacji i ułatwia prace programistyczne (np.: duża przestrzeń dyskowa, wygodny edytor, ...)
- wprowadza jednak utrudnienie procesu kompilacji - wymagane jest podanie szczegółów z reguły pomijanych, np.: polecenie 'configure' wymaga:

- określenia docelowej platformy np.:

`--host=arm-linux`

- miejsca działania w wynikowym systemie plików:

`--prefix=/usr`

- wariantu używanego kompilatora języka C:

`export CC=arm-linux-gcc`

- ścieżki do używanego kompilatora języka C:

`export PATH=/usr/local/arm-linux/bin:$PATH`

- sam GCC czasami potrzebuje wskazania miejsca jego instalacji (opcja `--sysroot`)

- przykład - skróśna kompilacja jądra (nieco inne parametry)

`make ARCH=arm CROSS_COMPILE=arm-linux-`

Oprogramowanie w języku C/C++ dla systemu Linux

■ Gdzie znaleźć narzędzia kompilacji skróśnej?

- Można utworzyć je samodzielnie ze źródeł
 - proces żmudny - tylko określone wersje Binutils/GCC/LibC „chcą” się razem skompilować!
 - proces wymaga posiadania jakiegoś pakietu kompilatorów
 - dylemat: „co było najpierw jajko czy kura”
- Można pobrać „pre-kompilowane” zestawy narzędzi
 - W systemie Debian: `apt-get update && apt-get install gcc-arm-linux-gnueabi`
- Można użyć generatorów narzędzi - proces czasochłonny ale automatyczny i efektywny
 - Crosstool-ng
 - przeznaczony dla wielu architektur, wspiera używanie w docelowym systemie bibliotek: uClibc, glibc, musl
 - Buildroot
 - bazuje na plikach Makefile, wspiera wiele architektur i pozwala wytworzyć biblioteki: glibc i uClibc
 - OpenEmbedded / Yocto
 - potężny zestaw - dość zawiła procedura tworzenia nowych systemów

Oprogramowanie w języku C/C++ dla systemu Linux

■ Składniki pakietu GCC

■ Pakiet Binutils

- narzędzia dla budowania „binarów” na docelową platformę
 - as - assembler, ld - linker, ar/ranlib - manipulacja bibliotekami
 - objdump, objcopy, readelf, size, nm, strings, strip - inspekcja i manipulacja binariami
- GCC bazuje na Binutils - programista niemal nie widzi tego pakietu podczas swojej pracy

■ Kompilator - główna część pakietu GCC

- wspiera wiele języków np.: C, C++, Ada, Fortran, Java, Objective-C/C++
- produkuje optymalny kod dla wielu rodzin CPU np.: ARM, Risc-V, AVR, CRIS, MIPS, PowerPC, i386, x86_64, IA64, Xtensa(ESP8266)
- Biblioteki dla podstaw języka C z reguły nie wchodzą w skład pakietu kompilatora ale są z nim związane choć można je podmieniać

■ Komunikacja jądra z aplikacjami - pliki nagłówkowe

- Rola: udostępnienie aplikacjom informacji niezbędnych dla procesu kompilacji i ich późniejszego działania

- informacji o usługach np.: `/libc/include/sys/linux-syscalls.h`

```
#define __NR_SYSCALL_BASE 0 //lub 4000
#define __NR_exit          (__NR_SYSCALL_BASE + 1)
#define __NR_fork          (__NR_SYSCALL_BASE + 2)
#define __NR_read          (__NR_SYSCALL_BASE + 3)
#define __NR_write         (__NR_SYSCALL_BASE + 4)
#define __NR_open          (__NR_SYSCALL_BASE + 5)
#define __NR_close         (__NR_SYSCALL_BASE + 6)...
```

- informacji o stałych np.: `/include/asm-generic/fcntl.h`

```
#define O_RDONLY 00000000
#define O_WRONLY 00000001
#define O_RDWR  00000002
```

- informacji o definicjach struktur np.: `asm/stat.h, ...`

```
struct stat {
    unsigned long st_dev;
    unsigned long st_ino;
    [...]
};
```

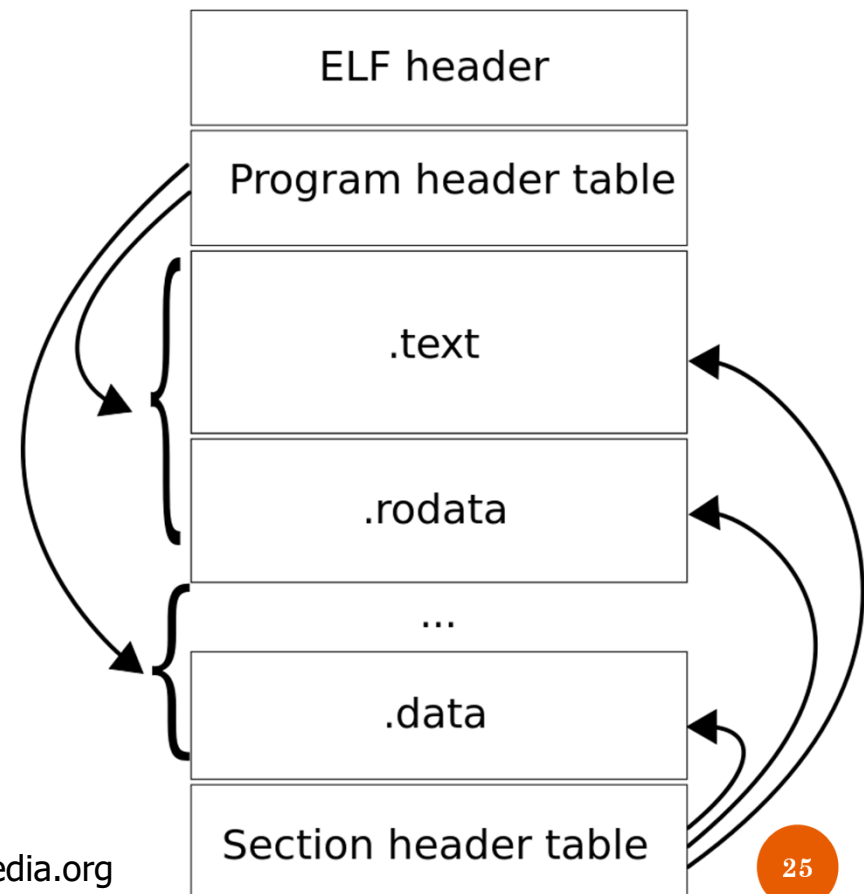
Oprogramowanie w języku C/C++ dla systemu Linux

- Dlaczego istnieje tak wiele bibliotek wsparcia dla języka C?
 - Zestaw usług jest zależny danej biblioteki
 - GLIBC - wspiera niemal wszystko zgodnie z POSIX
 - NewLibC - niektóre usługi są tylko zaślepkami (dla poprawnej kompilacji źródeł)
 - Specyfika systemów wbudowanych sprawia że pamięć składowania plików jest cenna
 - zastosowanie w węźle pamięci NOR-FLASH daje przestrzeń o wielkości maksymalnie 4MB
 - wielkość binariów*)
 - GLIBC: 2MB
 - uClibc: 500KB
 - Musl: ~400KB
 - Dietlibc: 120KB

*) Źródło: http://www.etalabs.net/compare_libcs.html

■ Budowa binarnego obrazu aplikacji

- Obecnie stosowany jest format ELF - wyparł między innymi A.OUT i COFF
 - Zapewnia relokowalność i niezależność od platformy czy CPU
 - Umożliwia przechowywanie także kodów bibliotek
- Budowa ELF
 - Nagłówek programu (x1)
 - Listy segmentów programu (x0,...,xN)
 - Listy nagłówków sekcji (x0,...,xN)
 - Danych zawierających segmenty i sekcje
- Przydatne narzędzia do analizy plików ELF
 - readelf
 - objdump



Oprogramowanie w języku C/C++ dla systemu Linux

■ Aplikacja po uruchomieniu otrzymuje swój identyfikator (PID)

- W systemach Unix/Linux aplikacja podczas działania jest procesem

- Stan każdego procesu opisuje wirtualny system plików

- jak jądro wywołało proces główny Init opisuje wpis w: `/proc/1/cmdline`

`/sbin/init`

- proces PID=404 (klient protokołu DHCP): `/proc/404/cmdline`

`dhclient -v -pf /run/dhclient.eth0.pid -lf /var/lib/dhcp/dhclient.eth0.leases eth0`

- opis limitów procesu PID=404: `/proc/404/limits`

Limit	Soft Limit	Hard Limi	Units
Max cpu time	unlimited	unlimited	seconds
Max file size	unlimited	unlimited	bytes
Max stack size	8388608	unlimited	bytes
Max core file size	0	unlimited	bytes
Max resident set	unlimited	unlimited	bytes
Max processes	15789	15789	processes
Max open files	1024	4096	files
Max locked memory	65536	65536	bytes
Max address space	unlimited	unlimited	bytes
Max pending signals	15789	15789	signals
Max msgqueue size	819200	819200	bytes
Max nice priority	0	0	
...			
Max realtime timeout	unlimited	unlimited	us

- Proszę pamiętać PROCFS czasami pomija spacje i wstawia znak `'\0'` a treść jego plików z poziomu powłoki najlepiej czytać za pomocą polecenia `'cat'`

Oprogramowanie w języku C/C++ dla systemu Linux

- Życie systemu inicjuje Init (PID=1) - rodzic wszystkich procesów
 - W systemach zgodnych z System-V - plik `/etc/inittab` ustala działanie procesu Init
 - System-V jest typowo używany przez systemy wbudowane, w przeciwieństwie do nowszego SystemD o nieco bardziej złożonej architekturze
 - Linie w `/etc/inittab` określają akcje które proces Init ma podjąć
 - Format: `id:runlevel:akcja:proces`
 - runlevel to poziom działania systemu (N,0,1,2,3,4,5,6) - określa ogólny stan jego działania
 - zwyczajowo określa się: N-boot, 0-system zatrzymany, 1-system przygotowany do pracy awaryjnej z jednym użytkownikiem, 6-restart systemu
 - dla przykładu aby umożliwić logowanie się do powłoki urządzenia bez względu na poziom działania systemu z wykorzystaniem portu szeregowego (`/dev/ttyATH0`) i akcją `askfirst` (po zakończeniu jest uruchamiany ponownie podany proces) służy wpis o treści:

```
ttyATH0::askfirst:/bin/ash --login
```
 - Nie jest zalecane dopisywanie własnych akcji do `inittab` - może zablokować system, a podmiana jego treści w systemie plików urządzenia wbudowanego może być bardzo utrudniona lub wręcz niemożliwa

Oprogramowanie w języku C/C++ dla systemu Linux

■ Uruchamianie własnych aplikacji po starcie w systemie Linux

- Zadanie złożone - aplikacje nie mają przypisanej im konsoli (stają się „DAEMONS”) a ich informacje o błędach muszą trafić do systemu logowania np.: /var/log/syslog

- Aplikacje można uruchamiać automatycznie przez utworzenie pliku: /etc/init.d/my_app.sh, z prawami do wykonywania

- `chmod +x /etc/init.d/my_app.sh`

- ustandaryzowana treść - początek pliku: Po czym można nakazać uruchomienie

```
### BEGIN INIT INFO
# Provides: moja_aplikacja
# Required-Start:    $remote_fs $syslog
# Required-Stop:    $remote_fs $syslog
# Default-Start:    2 3 4 5
# Default-Stop:     0 1 6
# Short-Description: moja_aplikacja
PATH=/sbin:/usr/sbin:/bin:/usr/bin
DESC="Moja aplikacja"
NAME= my_app.sh
DAEMON=/home/user/$NAME
DAEMON_ARGS=""
PIDFILE=/var/run/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME
...
```

aplikacji za pomocą:

```
sudo /etc/init.d/my_app.sh start
```

lub

```
su -c "/etc/init.d/my_app.sh start"
```

- Proszę pamiętać o zarejestrowaniu usługi (linki do /etc/rc<runlevel>.d/...)

- Niektóre dystrybucje w pliku `/etc/rc.local` pozwalają wpisać co ma być uruchomione po starcie

- Uwaga! obecnie zaczyna dominować ekwiwalent zwany SystemD

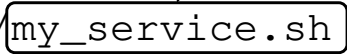
Oprogramowanie w języku C/C++ dla systemu Linux

■ Uruchamianie własnych aplikacji po starcie w systemie Linux

■ Podejście z SystemD

■ Baza to pliki opisu (/etc/systemd/services/my_service.service)

```
[Unit]
Description=my_service waiting for dhcp client finish its work
Requires=dhcpd.service
[Service]
User=root
Type=oneshot
ExecStart=/opt/my_service/my_service.sh
[Install]
WantedBy=multi-user.target
```



```
#!/bin/sh
#pamietaj o prawch dostępu "+x"

MY_ROUTER="192.168.1.1"
while [ 1 ]; do
    ping $MY_ROUTER 2>/dev/null ...
    if [ $? -eq 0 ]; then
        exit 0 #Router jest dostepny
    fi
    sleep 10 #czekamy 10sek.
done
```

■ Uruchomienie serwisu

```
sudo systemctl daemon-reload
sudo systemctl start my_service.service
```

■ Instalacja serwisu (usługa uruchomi się po ponownym starcie systemu)

```
sudo systemctl enable my_service.service
```

■ Jak sprawdzić stan usługi:

```
sudo systemctl status my_service.service
```

Oprogramowanie w języku C/C++ dla systemu Linux

■ Życie procesu

- Proces podczas działania jest związany ze standardowymi strumieniami I/O
 - STDIN - dane wejściowe, pobierane przez aplikacje z deskryptora pliku: 0
 - STDOUT - dane wyjściowe, „wrzucane” do pliku o deskrytorze: 1
 - STDERR - dane o błędach, „wrzucane” do pliku o deskrytorze: 2
- Procesy można łączyć w potoki
 - podejście umożliwia przekierowanie danych wyjściowych jednego procesu do wejścia innego
 - np.: zliczenie plików w całym systemie plików: `find -type f / | wc -l`
 - bardziej praktycznie - łącząc z działaniem funkcji fork (kreowanie procesów potomnych) umożliwia tworzenie nowych aplikacji
 - lub tworzenie jednolinijkowych poleceń - np.: odkrywanie za pomocą protokołu ARP jakie urządzenia są w określonej sieci IP

```
for i in `seq 1 254`; do arping -i eth0 -c 1 10.0.0.$i 2>/dev/null | grep ":" ; done
```

- Każdy proces po zakończeniu swojego działania może informować jak został zakończony
 - do tego właśnie służy `return` kończący funkcję `main` w aplikacji napisanej w C
 - z poziomu powłoki wynik ten można odczytać za pomocą zmiennej ``${?}``

Oprogramowanie w języku C/C++ dla systemu Linux

■ Skąd pobrać aplikacje dla swojej wersji systemu Linux

- Znakomita większość jest dostępna w wersjach źródłowych i na licencjach GPL/BSD/MIT
 - wystarczy je pobrać, skompilować i zainstalować
- Ale co konkretnie pobrać, czy istnieje repozytorium aplikacji 'ls'?
 - większość aplikacji tworzy większe grupy tematyczne - pakiet 'coreutils' zawiera wiele podstawowych aplikacji zarządzania plikami (w tym 'ls')
 - binaria tych aplikacji mogą nie mieścić się w małych zasobach pamięci trwałej węzła
 - Istnieją wersje pakietów dla systemów wbudowanych o małych zasobach
 - serwer ssh istnieje jako pakiet 'dropbear ssh' - nie wspiera jednak połączeń 'sftp' tylko 'scp'
 - odpowiednik aplikacji z 'coreutils' jest w specjalnym pakiecie 'busybox'

Oprogramowanie w języku C/C++ dla systemu Linux

■ Podejście stosowane przez Busybox jako panaceum na małe zasoby

- Podczas kompilacji wybiera się jakie usługi/polecenia mają być przez Busybox realizowane - co wybrano można zaobserwować:

```
>busybox
```

```
BusyBox v1.23.2 (2016-01-02 18:01:44 CET) multi-call binary.
```

```
Currently defined functions:
```

```
[, [[, arping, ash, awk, basename, brctl, bunzip2, bzip, cat, chgrp,
chmod, chown, chroot, clear, cmp, cp, crond, crontab, cut, date, dd,
find, free, fsync, grep, gunzip, gzip, halt, head, hexdump, hostid,
hwclock, id, ifconfig, kill, killall, less, ln, lock, logger, ls,
...
switch_root, sync, sysctl, tail, tar, tee, telnet, telnetd, test, time,
top, touch, tr, traceroute, true, udhcpc, umount, uname, uniq, uptime,
vconfig, vi, wc, wget, which, xargs, yes, zcat
```

- Uruchomienie polecenia 'ls' przechodzi przez link symboliczny, aby w końcu wywołał obsłużył sam Busybox np.:

```
/bin/ls -> busybox
```

```
/bin/ash -> busybox
```

```
/sbin/udhcpc -> ../bin/busybox
```

```
/sbin/ifconfig -> ../bin/busybox
```

- Zamiast wielu małych aplikacji system przechowuje jeden plik 'busybox' o wielkości ~300KB gdy na PC samo polecenie 'ls' to plik ~125KB
- Składnia (i działanie) poleceń busybox może być odmienna od ich ekwiwalentnych pełnych implementacji

Oprogramowanie w języku C/C++ dla systemu Linux

- Systemy wbudowane mają niewielkie zasoby pamięciowe - jak wiele ich potrzeba dla poprawnej pracy systemu Linux
 - Pamięć RAM: >2MB(dla obrazu jądra) + zależnie od wykorzystania węzła - dla całości zaleca się >8MB
 - Pamięć trwała: >2MB(dla przechowania obrazu jądra - choć można wytworzyć spakowany obraz o wielkości 400KB, to jest on użytkowo mało przydatny - brak większości mechanizmów) + zależnie od wykorzystania węzła (proste aplikacje to dziesiątki KB) - zaleca się dla całości >8MB
 - znane są udane konstrukcje routerów (zgodne z CPU: ADM5120) posiadające 2MB pamięci trwałej i 8MB pamięci RAM - ograniczeniem tych systemów jest utrudniona możliwości rozbudowy o nowe aplikacje
 - choć wykorzystując mechanizm pobierania (podczas startu) z sieci obrazów brakujących aplikacji można ominąć przeszkodę
- Łączność (niezbędna np.: dla węzłów IoT) w Linux to podstawa
 - Jest ukierunkowany na wsparcie dla IP (TCP/UDP/ARP/DHCP/...) zawiera także wiele aplikacji użytkowych dla sieci IP
 - Zawiera wbudowany firewall - konfigurowany vi 'iptables'

Moduły jądra Linux - tworzenie, uruchamianie, testowanie

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń (Device driver)

■ Urządzenie znakowe (Char Device)

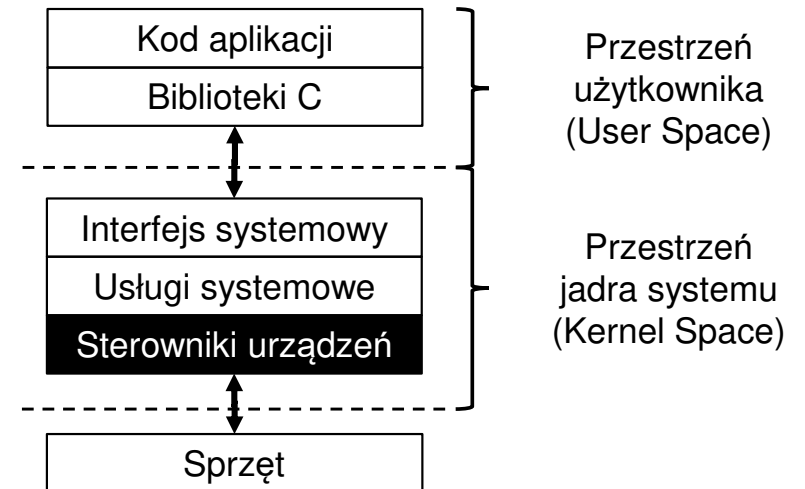
- Wspierają operacje odczytu i zapisu w stylu znakowym
- Przykłady
 - Klawiatura, mysz, ...

■ Urządzenia blokowe (Block Device)

- Wspierają operacje odczytu/zapisu bloków danych tworząc podsystem plików
- Przykłady
 - Dyski HDD, DVD, CD, ...

■ Urządzenie sieciowe (Network Device)

- Element podsystemu sieciowego, wspierają operacje wysyłania i odbierania pakietów danych
- Przykłady
 - Karty sieciowe (fizyczne), urządzenie pętli zwrotnej (loopback device)



■ Sterowniki urządzeń (Device driver), cd.

■ Budowa

■ Informacje pomocnicze

- Licencja: `MODULE_LICENSE("GPL v2");`
- Autor: `MODULE_AUTHOR("Author");`
- Zdawkowa informacja o module: `MODULE_DESCRIPTION("A sample driver");`
- Wersja oprogramowania: `MODULE_VERSION("2:1.0");`

■ Punkty wejścia

■ Init:

```
static int __init my_init_function(void) {  
    printk(KERN_INFO "Witam!");  
    ...  
}
```

```
module_init(my_init_function)
```

```
void __exit my_exit_funciton(void) {  
    ...  
}
```

```
module_exit(my_exit_funciton);
```

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń (Device driver), cd.

■ „printK”

- Odpowiednik funkcji printf() dla poziomu jądra systemu
- Czym jest pierwszy parametr – określa poziom logowania, np.:
 - KERN_EMERG – komunikaty o alarmach poprzedzających awarię
 - KERN_ALERT – komunikat wymagający natychmiastowej reakcji
 - KERN_INFO – poziom logowania zdarzeń systemowych
 - Inne: KERN_CRIT, KERN_ERR, KERN_WARNING, KERN_NOTICE
- Gdzie trafiają komunikaty
 - Domyślnie na domyślną konsolę systemową (monitor ekranowy, port szeregowy, ...)
 - Jądro utrzymuje specjalny bufor gdzie komunikaty są przechowywane – w czasie startu systemu może nie być jeszcze dołączonych (mount) dysków gdzie można byłoby zapisać te informacje
 - Istnieje narzędzie systemowe do „oglądania” tych logów: dmesg

■ Sterowniki urządzeń wymaga specjalnej kompilacji

■ Odwołuje się do jądra systemu na jakim ma moduł działać:

```
obj-m += my_module.o
```

```
KDIR = /lib/modules/$(shell uname -r)/build
```

```
all:
```

```
    make -C $(KDIR) M=$(shell pwd) modules
```

```
clean:
```

```
    make -C $(KDIR) M=$(shell pwd) clean
```

Wstawia wersję używanego jądra systemu

Wstawia miejsce w którym rozpoczęto proces kompilacji modułu

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń (Device driver), cd.

■ Ładowanie jądra

■ Statyczne

- Poprzez określenie konfiguracji jądra – moduł stanie się częścią obrazu binarnego nowo tworzonego jądra systemu

■ Dynamiczne

- Poprzez odpowiednie polecenia wydane po uruchomieniu systemu (lub jego trakcie)

```
sudo insmod /lib/modules/5.10.17/kernel/drivers/i2c/busses/i2c-gpio.ko
```

- Lub wygodniejsze – działa gdy system ma poprawnie zbudowaną listę zależności między modułami (polecenie `depmod -a`)

```
sudo modprobe my_module.ko
```

■ Proszę pamiętać że istnieje możliwość przekazywania parametrów do sterowników

■ Po co?

- Dla działania niektórych sterowników może istnieć konieczność przekazania informacji o ustawieniach sprzętowych – numer przerwania w starszych kartach sieciowych, adres bazowy
- Ze względu na wygodę zarządzania systemem - obecnie sterowniki unikają tego typu zabiegów, dzięki mechanizmom:
 - Plug-and-Play – sprzęt przedstawia się i podaje odpowiednie informacje
 - Device-Tree – podczas ładowania systemu operacyjnego bootloader przekazuje obraz konfiguracji opisujący konfigurację sprzętu (plik z rozszerzeniem `.dtb`)

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń (Device driver), cd.

■ Usuwanie sterownika

■ Po co?

- Aby dać szansę na właściwą operację na sprzęcie obsługiwanym przez określony sterownik
 - Np.: wyłączenie źródła przerw generowanych przez sprzęt, ...

■ Statyczne

- Podczas zamykania systemu, jądro automatycznie uruchamia właściwe funkcje (`my_exit_function`) sterowników urządzeń

■ Dynamiczne

- Uruchamiając narzędzie `rmmod`

```
sudo rmmod hello_world_module.ko
```

- Działanie takiego wywołania może być blokowane gdy moduł jest używany przez jakiś inny moduł

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń (Device driver), cd.

■ Informacje o modułach – lsmod, przykład:

Module	Size	Used by
hci_uart	40960	1
btbcm	16384	1 hci_uart
bluetooth	393216	29 hci_uart, bnep, btbcm, rfcomm
...		

■ Informacje o określonym module – przykład dla: modinfo i2c_dev

```
filename:      /lib/modules/5.10.17-v7l+/kernel/drivers/i2c/i2c-dev.ko
license:      GPL
description:   I2C /dev entries driver
author:       Simon G. Vogl <simon@tk.uni-linz.ac.at>
author:       Frodo Looijaard <frodol@dds.nl>
srcversion:   A8373F1DD184DE9263F531D
depends:
intree:      Y
name:        i2c_dev
vermagic:    5.10.17-v7l+ SMP mod_unload modversions ARMv7 p2v8
```


Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń – dla wirtualnych systemów plików

■ Po co?

- Utworzono dla zapewnienia komunikacji aplikacji użytkownika z jądrem systemowym, np.: dla zmiany jego ustawień, monitorowania działania
- Przykład – odczyt tablicy routingu: `cat /proc/net/route`

Iface	Destination	Gateway	Flags	RefCnt	Use	Metric	Mask	MTU	Window	IRTT
eth0	00000000	0100000A	0003	0	0	100	00000000	0	0	0
eth0	00000000	0100000A	0003	0	0	202	00000000	0	0	0
wlan0	00000000	0100000A	0003	0	0	303	00000000	0	0	0
eth0	0000000A	00000000	0001	0	0	100	00FFFFFF	0	0	0
eth0	0000000A	00000000	0001	0	0	202	00FFFFFF	0	0	0
wlan0	0000000A	00000000	0001	0	0	303	00FFFFFF	0	0	0

■ Przykład - opis sterownika i2C: `cat /sys/bus/i2c/devices/i2c-1/name`
bcm2835 (**i2c@7e804000**)

■ Przykład – pobranie adresu mac:

```
cat /sys/firmware/devicetree/base/scb/ethernet@7d580000/local-mac-address | hexdump -C
```

00000000	dc a6 32 41 42 43	..2ABC
----------	-------------------	--------

Co odpowiada adresowi: DC:A6:32:41:42:43

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń – dla wirtualnych systemów plików

■ ProcFS - historycznie pierwszy mechanizm komunikacji z jądrem systemu

■ Podstawowe wpisy

■ **/proc/devices** – zarejestrowane główne numery urządzeń znakowych i blokowych

- Konsola ma główny numer 5: `crw----- 1 root root 5, 1 maj 5 21:41 /dev/console`

■ **/proc/iomem** – adresy pamięci i urządzeń

`00000000-00000000 : System RAM`

`00000000-00000000 : Kernel code`

`00000000-00000000 : fd580000.ethernet ethernet@7d580000`

■ **/proc/interrupts** – liczba przerwanych zarejestrowanych urządzeń

	CPU0	CPU1	CPU2	CPU3				
27:	538427	644363	619734	610129	GICv2	30	Level	arch_timer
57:	23282842	0	0	0	GICv2	189	Level	eth0

■ **/proc/kallsyms** – tablica symboli jądra i sterowników urządzeń

`00000000 T stext`

`00000000 t sys_syscall`

■ **/proc/partitions** – aktualnie wspierane partycje urządzeń blokowych

major	minor	#blocks	name
179	0	15558144	mmcblk0

■ **/proc/filesystems** – aktualnie wspierane systemy plików

`nodev sysfs`

`nodev ramfs`

`ext4`

`nodev cifs`

`nodev smb3`

■ Sterowniki urządzeń – dla wirtualnych systemów plików

- ProcFS – kodu sterownika współdziałający z tym systemem plików, cd.

- Tworzenie wpisów:

- **struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent)**
 - name – nazwa wpisu-katalogu
 - parent – miejsce gdzie wpis-katalog się umiejscowi (gdy parent=NULL – w /proc)
- **struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct proc_dir_entry *parent, const struct file_operations *proc_fops)**
 - name – nazwa wpisu
 - mode – prawa własności tego wpisu
 - parent – miejsce gdzie wpis się umiejscowi (gdy parent=NULL – w /proc)
 - proc_fops – struktura opisu wspieranych przez sterownik operacji – najważniejszy element

```
static struct file_operations proc_fops = {  
    .open = my_open_proc,  
    .read = my_read_proc,  
    .write = my_write_proc,  
};
```

To jest mechanizm języka C do określania które pola struktury chcemy zainicjować – tu inicjujemy pole „open”

- Gdzie: open_proc, read_proc, write_proc – to funkcje wspierające operacje na podanym wpisie, odpowiednio otwarcia, odczytu, zapisu – główna część sterownika
- Lista wszystkich pól jest znacznie dłuższa(!)

■ Sterowniki urządzeń – dla wirtualnych systemów plików

- ProcFS – kodu sterownika współdziałający z tym systemem plików, cd.

```
struct file_operations {
    struct module *owner;
    loff_t(*llseek) (struct file *, loff_t, int);
    ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t(*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t(*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
    ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned
                                     long, unsigned long);
};
```

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń – dla wirtualnych systemów plików

■ ProcFS – kodu sterownika współdziałający z tym systemem plików, cd.

```
...//pliki nagłówkowe + prolog modułu
#define MY_BUFFER_LEN 128
char my_array[MY_BUFFER_LEN]="Witam!\n";
static struct proc_dir_entry *ent;
```

```
static struct file_operations proc_fops={
    .read = my_read_proc,
    .write = my_write_proc,
};
```

```
static ssize_t my_read_proc(struct file *filp, char __user *buf, size_t len, loff_t *off){
    printk(KERN_INFO "MY Read PROC\n");
    if(len<=strlen(my_array))
        return 0; //brak miejsca w buforze użytkownika
    if(copy_to_user(buf, my_array, strlen(my_array))) return -EFAULT;
    return strlen(my_array);
}
```

Funkcja czytania z urządzenia (czyli wirtualnego systemu plików)

```
static ssize_t my_write_proc(struct file *filp, const char *buf, size_t len, loff_t *off){
    printk(KERN_INFO "MY Write PROC\n");
    if(len>MY_BUFFER_LEN)
        return 0; //brak miejsca w buforze sterownika
    if(copy_from_user(my_array, buf, len)) return -EFAULT;
    return len;
}
```

Funkcji pisania do urządzenia

```
static int my_init(void){
    ent=proc_create("mydev",0660,NULL,&proc_fops);
    return 0;
}

static void my_cleanup(void){
    proc_remove(ent);
}
```

```
module_init(my_init);
module_exit(my_cleanup);
```

EFAULT – to błąd zwracany gdy argument zawiera adres wskazujący obszar nie będący własnością procesu użytkownika

■ Sterowniki urządzeń – dla wirtualnych systemów plików, cd.

■ SysFS

- Unowocześniony względem ProcFS mechanizm komunikacji z jądrem systemu, mocniej ukierunkowany na obsługę sprzętu
- Podstawą jest struktura: **kobject**, reprezentując obiekt jądra zapewniając połączenie z nim
 - mimo to tworząc moduł związany z SysFS jest ona mniej ważna

■ Jak utworzyć własny moduł SysFS

■ Budowa wpisów w SysFS na bazie kobject

```
#include <linux/sysfs.h>
```

```
...
```

```
static struct kobject *sykt; //tworzymy wskaźnik na kobject
```

```
...
```

```
        //tworzymy nowy katalog w SysFS, kernel_kobj - rodzic  
sykt=kobject_create_and_add("my_sysfs", kernel_kobj);
```

```
...
```

```
sysfs_create_file(sykt, &sykt_file_attr.attr); //tworzymy pliki w SysFS
```

```
...
```

```
kobject_put(sykt);
```

Wywołanie podczas uruchamiania modułu

Wywołanie gdy usuwamy moduł

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń – dla wirtualnych systemów plików, cd.

■ SysFS, cd.

■ A co to jest **sykt_file_attr.attr**?

- Właściwa część modułu – definiuje m.in. funkcje obsługi plików

```
#define MY_SYSFSDRV_BUFFER_LEN    PAGE_SIZE    //dla arch. zgodnych z i386: 4096B
char my_array[MY_SYSFSDRV_BUFFER_LEN]="Oto SYSFS!\n";

static ssize_t sykt_file_show(struct kobject *kobj, struct kobj_attribute *attr,
                                char *buffer){
    memcpy(buffer, my_array, strlen(my_array, MY_SYSFSDRV_BUFFER_LEN)+1);
    return strlen(my_array, MY_SYSFSDRV_BUFFER_LEN);
}

static ssize_t sykt_file_store(struct kobject *kobj,
                                struct kobj_attribute *attr, const char *buffer, size_t count){
    int n=count > MY_SYSFSDRV_BUFFER_LEN ? MY_SYSFSDRV_BUFFER_LEN : count;
    strncpy(my_array, buffer, n);
    return n;
}

static struct kobj_attribute sykt_file_attr = __ATTR_RW(sykt_file);
```

Moduły w jądra Linux - tworzenie, uruchamianie, testowanie

■ Sterowniki urządzeń – dla wirtualnych systemów plików, cd.

■ SysFS, cd.

■ Skąd bierze się nazwa pliku tworzonego przez moduł?

■ Wszystko dzięki makro definicji

```
#define __ATTR_RW(_name) __ATTR(_name, (S_IWUSR | S_IRUGO), \
                                   _name##_show, _name##_store)
```

■ Dla __ATTR_RW(sykt_file) rozwija się do

```
__ATTR(sykt_file, (S_IWUSR | S_IRUGO), sykt_file_show, sykt_file_store)
```

■ Natomiast __ATTR rozwija deklaracje do postaci

```
static struct device_attribute dev_attr_sykt_file = {
    .attr = {
        .name = "sykt_file",
        .mode = S_IWUSR | S_IRUGO,
    },
    .show = sykt_file_show,
    .store = sykt_file_store,
};
```

■ Proszę nie zapomnieć o funkcjach module_init(my_init_module) i module_exit(my_cleanup_module)

■ Sterowniki urządzeń – dla wirtualnych systemów plików, cd.

■ SysFS, cd.

- Jakie będą atrybuty dostępu do tworzonego w SysFS pliku? zamiast `__ATTR_RW()` dającego prawa 0644 (czyli owner: RW, group: R, other: R) możemy użyć m.in.:

`__ATTR_RO(name)`

- powstanie plik tylko do odczytu (0444)
- definiuje strukturę z dołączoną funkcji show

`__ATTR_WO(name)`

- powstanie plik do którego prawo zapisu będzie miał 'root' (0200)
- definiuje strukturę z dołączoną funkcji store

`__ATTR_NULL`

- nie definiuje żadnych połączeń z funkcjami store/show

- Proszę pamiętać że moduły jądra pracują z prawami użytkownika 'root'

■ Więcej informacji na temat SysFS

- <https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt>

■ Sterowniki urządzeń – właściwe odwołania do sprzętu

- Jądro systemu dostarcza specjalizowane funkcje dostępu do peryferii zmapowanych pod określonymi adresami w przestrzeni pamięciowej CPU, np.:

```
u32 readl(const volatile void __iomem *addr);  
  
void writel(u32 b, volatile void __iomem *addr);
```

- Ich użycie jest możliwe wyłącznie z poziomu jądra i dopiero po zmapowaniu pożądanej przestrzeni We/Wy za pomocą funkcji

```
static inline void __iomem *ioremap(unsigned long port, unsigned long size);
```

- „A successful call to `ioremap()` returns a kernel virtual address corresponding to start of the requested physical address range”

- Przykład użycia (zaczepnięty z wprowadzenia do projektu):

```
void __iomem *baseptr;  
  
baseptr=ioremap(SYKT_GPIO_BASE_ADDR, SYKT_GPIO_SIZE);  
  
writel(SYKT_EXIT | ((SYKT_EXIT_CODE)<<16), baseptr);
```

Dziękujemy za uwagę!