Rozdział 1

Uruchamianie systemu i niskopoziomowe mechanizmy obsługi procesów

Spis treści

Wstęp

Rozdział 1. omawia wykorzystywane przez jądro Linuksa mechanizmy niskopoziomowe architektury „i386”, czyli opartej na procesorach firmy Intel począwszy od modelu 386. Zawiera informacje o przerwaniach, przełączaniu kontekstu, działaniu sprzętowego mechanizmu obsługi błędów braku i ochrony strony, obsłudze DMA, zegarach sprzętowych i mechanizmie realizacji wywołań systemowych. Podrozdział 1.1. przedstawia dokładny opis przebiegu ładowania i uruchamiania systemu.
Autorami poszczególnych podrozdziałów są:

- **1.1.** i **1.7.** : Krzysztof Gwiazda, Marcin Malinowski, Robert Osowiecki, Paweł Pałucha i Wojciech Moczydłowski.
- **1.2.** : Paweł Walczewski, Marcin Dimitrow, Jacek Rembisz i Tomasz Juśkiewicz.
- **1.3.** i **1.4.** : Arkadiusz Kindziuk, Marcin Szlufik,Piotr Bugalski, Paweł Lenk i Artur Ratyński.
- **1.5.** i **1.6.** : Michał Bernat, Sławomir Leszczyński i Adam Borowski.

1.1 Uruchamianie systemu

1.1.1 Struktura MBR i tablic partycji dysku systemowego

Aby lepiej zrozumieć strukturę MBR i strukturę tablic partycji, zastanówmy się najpierw, jakie funkcje spełnia MBR i w jakim celu wprowadzono podział dysku twardego na partycje.
Współczesne komputery osobiste są urządzeniami o budowie modułowej, przy czym ta modułowość ujawnia się na każdym poziomie abstrakcji działania komputera. Jednym z podstawowych modułów, które można wyróżnić, jest oprogramowanie traktowane jako całość (system operacyjny, programy użytkowe). Z tego względu całe oprogramowanie komputera łatwo jest zastąpić innym, być może nowszym, być może lepszym, a może po prostu o własnościach przydatnych w pewnych zastosowaniach. Idea w pełni wymiennego oprogramowania w komputerze PC jest realizowana przez wydzielenie pewnych podstawowych operacji wykonywanych po uruchomieniu komputera, niezbędnych do późniejszego uruchomienia dowolnego systemu operacyjnego. Operacje te realizowane są przez **BIOS** (ang. *Basic Input Output System*) (dostarczany razem ze sprzętem) i możemy do nich zaliczyć wstępne testy sprzętu, udostępnienie przystępnego i uniwersalnego sposobu komunikacji z urządzeniami, z których może korzystać system operacyjny, oraz wczytanie pewnego kodu dostarczonego wraz z systemem operacyjnym, który następnie załaduje resztę systemu i zainicjuje jego poprawny start. Kod ten wczytywany jest z pierwszego sektora urządzenia startowego (**MBR – Master Boot Record**) pod adres `0x07C00` i następnie jest mu przekazywane sterowanie. W wyniku tego[1], za pomocą udostępnionych funkcji BIOS-u, załadowany zostaje system operacyjny.

Ten prosty mechanizm stosowany jest w przypadku startu systemu z dyskietki. W przypadku dysku twardego musimy uwzględnić dodatkowo podział dysku na logiczne fragmenty, nazywane

[1] Ze względu na niewielki rozmiar MBR często pierwszy program ładujący wczytuje następne, większe fragmenty kodu, które są w stanie wczytać i dokonać wstępnej inicjalizacji jądra systemu operacyjnego.

partycjami (ang. *partition*). Partycjonowanie dysku jest bardzo elastycznym rozwiązaniem, zapewniającym w dużym stopniu hermetyzację wydzielonych obszarów dysku twardego. Dodatkowym jego atutem jest to, że partycje (dyski logiczne) w wielu sytuacjach posiadają cechy osobnych urządzeń fizycznych. Możliwe dzięki temu jest utrzymywanie kilku systemów operacyjnych działających na jednym dysku twardym i operujących na rozłącznych jego obszarach. Możliwe jest też wsparcie dla systemów, które preferują korzystanie z kilku urządzeń dyskowych, z których każde realizuje inne funkcje. Wreszcie partycjonowanie dysku jest po prostu wygodne przy zarządzaniu zgromadzoną informacją.

Partycje są rozłącznymi[2], spójnymi obszarami dysku, dla których istnieje odpowiedni wpis w którejś z tablic partycji. Partycje muszą kończyć się wraz z końcem cylindra, zaś zaczynać od początku ścieżki dysku twardego. Dysk twardy może posiadać do czterech partycji **podstawowych** (ang. *primary*), to znaczy takich, na których bezpośrednio będą znajdować się dane. Zamiast którejś z partycji podstawowych może wystąpić partycja **rozszerzona** (ang. *extended*), na której znowu znajdzie się do czterech partycji, zwanych logicznymi. Partycje logiczne również mogą być partycjami rozszerzonymi, choć nie wszystkie systemy operacyjne (i programy do partycjonowania dysków) to obsługują. Niektóre ukrywają to przed użytkownikami[3].

Struktura logiczna dysku twardego jest jednoznacznie opisywana przez tablice partycji. W MBR znajduje się pierwsza tablica partycji, opisująca partycje podstawowe i partycje rozszerzone położone bezpośrednio na dysku twardym. W pierwszym sektorze dowolnej partycji rozszerzonej znajduje się tablica partycji opisująca partycje umiejscowione na tej partycji rozszerzonej[4]. Zauważmy, że w MBR zlokalizowane są jednocześnie program ładujący i tablica partycji. Program ładujący zajmuje pierwsze 446 bajtów sektora, tablica partycji kolejne 64 bajty, zaś ostatnie dwa bajty to znacznik, którego wartość powinna wynosić `0xAA55`[5]. Taką samą strukturę posiadają pierwsze sektory partycji rozszerzonych. W tablicy partycji znajdują się cztery wpisy, po 16 bajtów każdy, o następującej strukturze:

```
struct partition {
unsigned char boot_ind; /* 0x80 - aktywna */
unsigned char head; /* ścieżka początkowa */
unsigned char sector; /* sektor początkowy */
unsigned char cyl; /* cylinder początkowy */
unsigned char sys_ind; /* typ partycji */
unsigned char end_head; /* ścieżka końcowa */
unsigned char end_sector; /* sektor końcowy */
unsigned char end_cyl; /* cylinder końcowy */
unsigned int start_sect; /* sektor początkowy partycji */
unsigned int nr_sects; /* liczba sektorów wchodzących w skład partycji */
}
```

[2]Oczywiście partycja rozszerzona nie jest rozłączna z partycjami logicznymi znajdującymi się na niej. Jednak partycje rozszerzone pełnią tylko rolę administracyjną.

[3]W Linuksie bezpośrednio na partycji rozszerzonej może znajdować się najwyżej jedna partycja rozszerzona, gdyż pozostałe zostaną zignorowane podczas inicjalizacji urządzeń. Zatem dopuszczona jest struktura zagnieżdżona, ale nieobsługiwane są drzewiaste rozgałęzienia.

[4]Program Disk Druid, służący do partycjonowania dysku nie przestrzega tej reguły – w komputerze autora tablica partycji z partycji rozszerzonej opisuje partycję, która nie znajduje się w obrębie tej partycji rozszerzonej. Zawieranie się odpowiednich partycji nie przez Linuksa sprawdzane.

[5]Na dysku kolejność bajtów jest odwrócona.

Powyższą definicję struktury można znaleźć w pliku `include/linux/genhd.h`. Jest ona używana w trakcie inicjalizacji urządzeń w pliku `drivers/block/genhd.c`. Sektor początkowy partycji jest numerem sektora liczonym od sektora zawierającego bieżącą tablicę partycji. Typ `unsigned int`, jak można wywnioskować, jest tutaj typem 32 bitowym.

Ze względu na to, że w każdym pierwszym sektorze partycji rozszerzonej znajduje się wolna przestrzeń, można ją wykorzystać umieszczając kolejny program ładujący, wczytywany i uruchamiany przez ten z MBR. Również na partycjach podstawowych i logicznych zwykle pierwszy sektor jest rezerwowany na program, który będzie ładował do pamięci system umieszczony na tej partycji. Zadania programu z MBR ograniczają się zatem często do załadowania kolejnego programu ładującego. W ten sposób działa dosowy program ładujący. Partycja, z której załadowany ma być kolejny program, oznaczana jest za pomocą **flagi aktywności** (ang. *active flag*), która jest pierwszym polem struktury `partition`. W ten sposób można wybrać na stałe system, który będzie ładowany do pamięci podczas startu komputera. Jednak np. LILO[6], który można umieścić w MBR, nie korzysta z tej flagi i udostępnia możliwość wyboru systemu w trakcie startu komputera.

W Linuksie do poszczególnych partycji dysku twardego odwołujemy się poprzez urządzenia, przy czym jeżeli urządzeniem odpowiadającym całemu dyskowi jest np. `/dev/hda` to:

- `/dev/hda1`
- `/dev/hda2`
- `/dev/hda3`
- `/dev/hda4`
  oznaczają kolejne partycje podstawowe i partycje rozszerzone zlokalizowane bezpośrednio na dysku twardym
- `/dev/hda5` i wyższe, oznaczają partycje logiczne.

Partycje rozszerzone położone w obrębie partycji rozszerzonych nie są reprezentowane przez żadne urządzenia w systemie.
Do poprawnej interpretacji danych na różnych rodzajach partycji służy pole `sys_ind` ze struktury `partition`. Wartości tego pola obsługiwane przez linuksowy program fdisk są następujące:

0	Empty	16	Hidden FAT16	61	SpeedStor	a6	OpenBSD
1	FAT12	17	Hidden HPFS/NTF	63	GNU HURD or Sys	a7	NeXTSTEP
2	XENIX root	18	AST Windows swa	64	Novell Netware	b7	BSDI fs
3	XENIX usr	24	NEC DOS	65	Novell Netware	b8	BSDI swap
4	FAT16 <32M	3c	PartitionMagic	70	DiskSecure Mult	c1	DRDOS/sec (FAT-
5	Extended	40	Venix 80286	75	PC/IX	c4	DRDOS/sec (FAT-
6	FAT16	41	PPC PReP Boot	80	Old Minix	c6	DRDOS/sec (FAT-
7	HPFS/NTFS	42	SFS	81	Minix / old Lin	c7	Syrinx
8	AIX	4d	QNX4.x	82	Linux swap	db	CP/M / CTOS / .
9	AIX bootable	4e	QNX4.x 2nd part	83	Linux	e1	DOS access
a	OS/2 Boot Manag	4f	QNX4.x 3rd part	84	OS/2 hidden C:	e3	DOS R/O
b	Win95 FAT32	50	OnTrack DM	85	Linux extended	e4	SpeedStor

[6]LInux LOader, standardowy program ładujący dla Linuksa, zobacz rozdział 1.1.3.

```
 c  Win95 FAT32 (LB 51  OnTrack DM6 Aux 86  NTFS volume set eb  BeOS fs
 e  Win95 FAT16 (LB 52  CP/M            87  NTFS volume set f1  SpeedStor
 f  Win95 Ext'd (LB 53  OnTrack DM6 Aux 93  Amoeba          f4  SpeedStor
10  OPUS            54  OnTrackDM6      94  Amoeba BBT      f2  DOS secondary
11  Hidden FAT12    55  EZ-Drive        a0  IBM Thinkpad hi fe  LANstep
12  Compaq diagnost 56  Golden Bow      a5  BSD/386         ff  BBT
14  Hidden FAT16 <3 5c  Priam Edisk
```

Na uwagę zasługują partycje o wartościach pola typ równych 5h, 6h, 82h, 83h. Partycja typu Linux oznacza partycję z systemem plików ext2.

1.1.2 Obraz jądra

Jądro powstaje z kompilacji plików umieszczonych w katalogach ze zródłami. Ścieżki w dalszej części będziemy podawać od głównego katalogu źródeł, najczęściej `/usr/src/linux`. Katalog `arch` zawiera części kodu zależne od architektury, na której ma działać jądro. Nas interesuje jego podkatalog `arch/i386`, odpowiedni dla komputerów PC z procesorami Intela i pochodnymi. Inne katalogi zawierają:

- `drivers` - sterowniki urządzeń zewnętrznych
- `fs` - systemy plików
- `ibcs` - obsługa innych Uniksowych formatów wykonywalnych
- `include` - pliki nagłówkowe
- `init` - inicjalizacja struktur jądra
- `ipc` - mechanizmy komunikacji międzyprocesowej Systemu V
- `kernel` - podstawowe mechanizmy jądra
- `lib` - część bibliotek C dostępna dla jądra
- `mm` - zarządzanie pamięcią
- `modules` - skompilowane moduły jądra
- `net` - obsługa sieci

Specyfika ładowania jądra przez program ładujący powoduje, że teoretycznie nie powinno ono zajmować więcej niż 448 KB, gdyż tyle miejsca jest dla niego przeznaczone (od adresu `0x90000` będzie zaczynał się `setup`). Tymczasem, mimo kompresji, większość współczesnych jąder po kompilacji zajmuje więcej. W związku z tym możliwe jest skompilowanie jądra jako **dużego** (ang. *BIG*). Tak utworzone jądro różni się od zwykłego tylko nazwami plików pośrednich i ostatecznego obrazu (mają na początku dodaną literę 'b'). Informacja o tym, że jądro jest duże jest też przekazywana do kodu jądra (przez dodanie do opcji kompilatora `gcc` deklaracji `__BIG_KERNEL__`). W ten sposób kod jądra może rozpoznać, że ma się załadować „wyżej”, powyżej granicy 1 MB.

Kompilację jądra wywołuje się przez uruchomienie w głównym katalogu ze źródłami programu `make` z odpowiednią opcją, najczęściej *zImage* lub *bzImage* – odpowiadającą odpowiednio tworzeniu jądra zwykłego i dużego (BIG). Przed właściwą kompilacją należy jądro skonfigurować. Polecenie

```
# make config
```

pozwala skonfigurować jądro za pomocą odpowiadania na kolejne zapytania. Konfiguracja za pomocą menu tekstowego lub graficznego jest możliwa po wydaniu komend

```
# make menuconfig
```

lub

```
# make xconfig.
```

W wyniku konfiguracji powstaje plik `.config`, zawierający w postaci tekstowej wybrane przez użytkownika opcje. Jego zawartość, przetworzona przez `make`, będzie miała wpływ zarówno na to, jakie pliki wezmą udział w konfiguracji, jak i na opcje kompilatora i postać samego kodu (przez dyrektywy preprocesora). Następnie należy stworzyć zależności dla kompilatora gcc:

```
# make depend
```

i posprzątać po poprzedniej kompilacji:

```
# make clean.
```

Po kompilacji, czyli

```
# make zImage
```

lub

```
# make bzImage,
```

jeżeli jakaś część jądra została skompilowana jako moduł, należy jeszcze skompilować i zainstalować moduły:

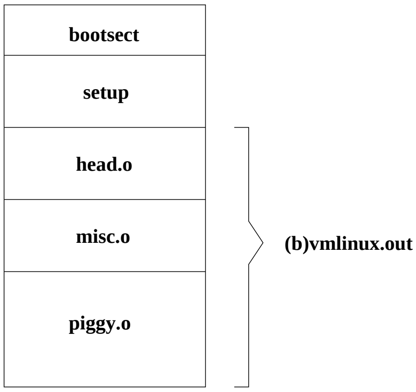
```
# make modules
# make modules install.
```

Instalowanie jądra zależy od używanego programu ładującego, np. w przypadku LILO polega na przegraniu go z katalogu `arch/i386/boot` do podkatalogu `/boot` w głównym katalogu, utworzeniu odpowiedniego wpisu w pliku `/etc/lilo.conf` i uruchomieniu programu `lilo`.

W wyniku kompilacji powstaje na dysku plik z jądrem, tzw. obraz jądra. Zawiera on po kolei następujące pliki: `bootsect`, `setup` i `vmlinux.out` (`bvmlinux.out` w przypadku dużego jądra), składający się z kolei z: `head.o`, `misc.o` i `piggy.o`.

`bootsect` powstaje z kompilacji pliku `arch/i386/boot/bootsect.S` i jest to obraz sektora ładującego system z dyskietki. Zadaniem umieszczonego tam kodu jest po prostu załadowanie do pamięci kolejnych ścieżek z dyskietki, zawierających dalszą część jądra. Taka struktura pliku jądra pozwala na bardzo proste tworzenie dyskietki ładującej, przez skopiowanie obrazu jądra do fizycznego urządzenia, np:

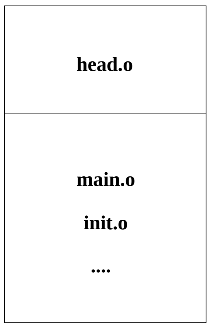
```
# dd if=/boot/vmlinuz of=/dev/fd0
```



Rysunek 1.1: Obraz jądra (`zImage` / `bzImage`)

(gdzie `vmlinuz` jest obrazem jądra). `bootsect` nie jest wykorzystywany przy ładowaniu systemu z dysku twardego, ale jest zawsze obecny w obrazie jądra.
`setup` powstaje z kompilacji dwóch plików: `arch/i386/boot/setup.S` oraz `arch/i386/boot-/video.S`. Zawiera inicjację sprzętu (w tym karty graficznej), a w szczególności przełącza procesor w tryb chroniony.
`head.o`, powstały z kompilacji `arch/i386/boot/compressed/head.S` ma za zadanie przygotować środowisko i wywołać pierwszą część kodu jądra napisaną w C – procedurę dekompresującą jądro, umieszczoną w pliku `/arch/i386/boot/compressed/misc.c`, kompilowanym do `misc.o`.

Skompresowane jądro umieszczone jest w pliku `piggy.o` (Rys. 1.2), który zawiera jeszcze część kodu inicjującego. Jest to między innymi `head.o` (powstały z `arch/i386/kernel/head.S`), odpowiedzialny za ostateczne przygotowanie środowiska dla kodu w C, oraz kod utworzony po kompilacji podkatalogu `/init`, zawierający funkcję wejściową jądra (`start_kernel()`) i tworzenie pierwszego procesu. Dalej zaś znajduje się już reszta kodu jądra. Składowe pliku `piggy.o`



Rysunek 1.2: Skompresowana część jądra (`piggy.o`)

są oddzielnie kompilowane, złączane za pomocą programu `objcopy`, kompresowane programem `gzip` i łączone linkerem z plikami `head.o` i `misc.o` – powstaje `vmlinux.out` (lub `bvmlinux.out` w przypadku dużego jądra). Ostatnim etapem tworzenia obrazu jądra jest uruchomienie programu build (powstałego z kompilacji arch/i386/boot/tools/build.c). Program ten dokonuje pewnej weryfikacji poprawności plików wejściowych, po czym ustawia je jeden za drugim i wysyła na standardowe wyjście. Jego wywołanie w arch/i386/boot/Makefile powoduje, że pliki bootsect, setup i vmlinux.out (bvmlinux.out) zapisywane są jako arch/i386/boot/zImage (bzImage).

1.1.3 Programy ładujące dla Linuksa

LILO

Linux Loader (LILO) jest standardowym **programem ładującym** (ang. *boot loader*) dla Linuksa. Umożliwia nie tylko wystartowanie Linuksa, ale również wybór innego systemu, który ma zostać załadowany, wybór jądra Linuksa, które ma być wczytane do pamięci, a także start systemu z dysku logicznego lub z drugiego dysku twardego.
Pakiet LILO składa się z dwóch głównych części:

- część, z której odczytywane są informacje podczas startu komputera
- część, uaktualniająca te informacje, uruchamiana w trakcie sesji Linuksa

Do pierwszej z nich należą pliki /boot/boot.b [7], /boot/map, do drugiej pliki /sbin/lilo i /etc-/lilo.conf. Pierwszy z nich zawiera program tworzący na dysku konfigurację zgodną z zawartością drugiego z plików. Pozostałe dwa pliki LILO, które można znaleźć w katalogu /boot: (chain.b i os2_d.b zawierają kody programów, które potrzebne są do uruchomienia obcych systemów operacyjnych
LILO umożliwia załadowanie jądra, niezależnie od typu systemu plików, na którym się ono znajduje. Możliwe jest to dzięki temu, że poszukując sektorów dyskowych, zawierających plik jądra, nie korzysta z systemu plików, ale wczytuje sektory o numerach wcześniej przekazanych przez system. Numery te przechowywane są w pliku /boot/map. W tym pliku znajdują się również informacje o tym jakie pliki jąder bądź inne systemy operacyjne mogą zostać załadowane do pamięci.

Ładowanie systemu za pomocą LILO rozpoczyna się od wczytania **sektora ładującego** (ang. *boot sector*) przygotowanego przez LILO. Sektor ładujący (umieszczany np. w MBR) jest kopią pierwszych 512 bajtów pliku /boot/boot.b (z bajtami przeznaczonymi na dane wypełnionymi przez program lilo z pliku /sbin/lilo). Znajduje się w nim tzw. **podstawowy program ładujący** (ang. *primary boot loader*). Jego struktura, którą można znaleźć w pliku lilo/common.h ze źródeł LILO, wygląda następująco:

```
typedef struct {
    char jump[6]; /* skok do kodu */
    char signature[4]; /* "LILO" */
    unsigned short stage,version; /* stage: 0x01 -podstawowy, 0x02 -drugorzędny,
                                    0x10 -chain */
    unsigned short timeout; /* czeka 54 msec zanim mierzony będzie limit czasu;
                              0xffff: nigdy */
    unsigned short delay; /* opóźnienie: czekaj tyle 54 msec jednostek. */
    unsigned char port; /* port COM. 0 = nie ma, 1 = COM1, etc. */
```

[7] Podane położenia są domyślnymi położeniami dla plików LILO.

```
    unsigned char ser_param; /* parametry RS-232, 0 jeśli nie używane */
    unsigned long timestamp; /* stempel czasowy */
    SECTOR_ADDR descr[3]; /* 2 sektory tablicy deskryptorów i sektor domyślnej
                             linii poleceń */
    unsigned char prompt; /* 0 = tylko na żądanie, =! 0 = zawsze */
    unsigned short msg_len; /* 0 jeśli nie ma */
    SECTOR_ADDR msg; /* sektor z komunikatem powitalnym */
    SECTOR_ADDR keytab; /* sektor tablicy translacji klawiatury */
    SECTOR_ADDR secondary[MAX_SECONDARY+1]; /* 10 pozycji; sektory drugorzędnego
                                              programu ładującego */
} BOOT_PARAMS_1; /* początek sektora ładującego LILO */
```

Pozostałe bajty sektora zajmowane są przez kod wykonywalny i ewentualnie tablicę partycji. `SECTOR_ADDR` jest 5-cio bajtową strukturą zawierającą adres fizyczny sektora.

Po załadowaniu podstawowy program ładujący przesuwa się sam pod adres `0x9a000`, ustawia stos (który rośnie w dół od `0x9b000` do `0x9a200`), wczytuje **drugorzędny program ładujący** (ang. *secondary boot loader*) pod adres `0x9b000` i przekazuje do niego sterowanie. Rozmiar drugorzędnego programu ładującego nie może przekroczyć 8-miu sektorów (chociaż w sektorze ładującym LILO jest przewidziane miejsce na adresy 10-ciu sektorów). Drugorzędny program ładujący zajmuje pozostałą część pliku `/boot/boot.b`. Po załadowaniu wczytuje dwa sektory tablicy deskryptorów (pod `0x9d200`) oraz sektor zawierający domyślną linię poleceń dla jądra (`0x9d600`). Tablica deskryptorów ma następującą strukturę:

```
typedef union {
    struct {
unsigned short checksum;
IMAGE_DESCR descr[MAX_IMAGES]; /* deskryptory obrazów ładowalnych */
    } d;
    unsigned char sector[SECTOR_SIZE*2]; /* 1024 bajty */
} DESCR_SECTORS;
```

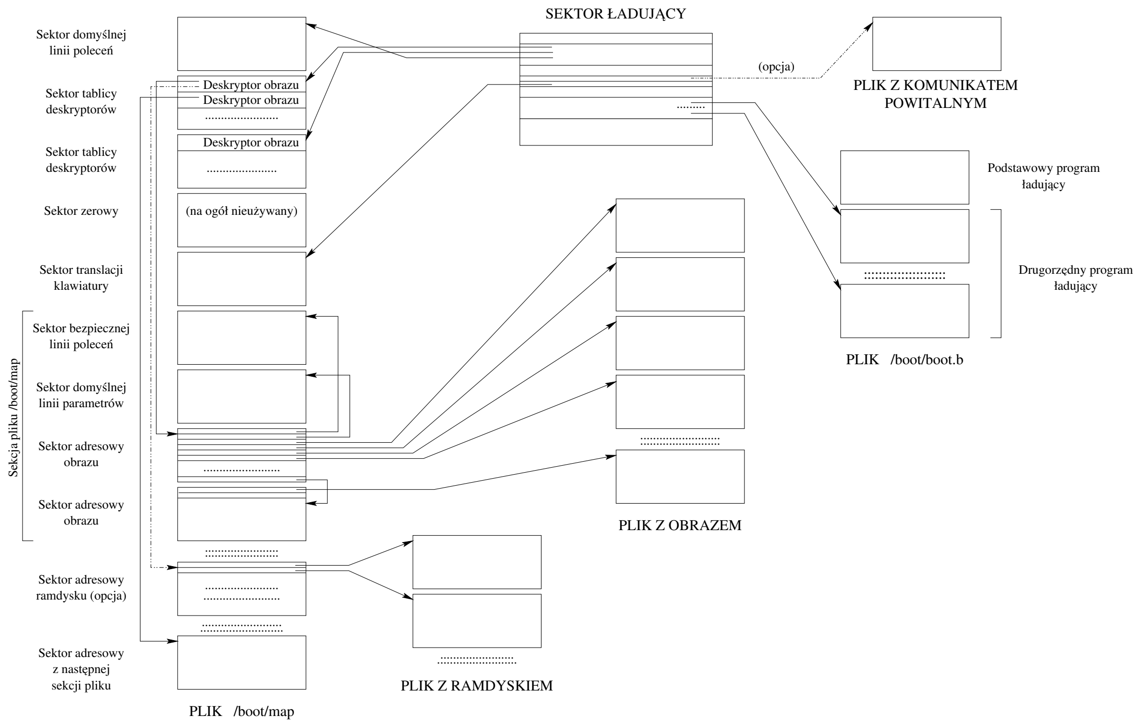
Jest ona przechowywana w drugim i trzecim sektorze pliku `/boot/map`, natomiast w pierwszym sektorze tego pliku znajduje się **domyślna linia poleceń** (ang. *default command line*). **Obrazem ładowalnym** (ang. *boot image*) będziemy nazywać plik zawierający jądro Linuksa, lub specjalnie przygotowane trzy sektory, z których dwa (612 bajtów) zawierają kopię **pośredniego programu ładującego** (ang. *chain loader*) wraz z wyspecyfikowaną w `/etc/lilo.conf` tablicą partycji, zaś trzeci jest kopią pierwszego sektora (ładującego) partycji wybranej w `/etc/lilo.conf`. Te trzy sektory nie są umieszczane w osobnym pliku (jak jądro), lecz wpisywane są do pliku `/boot/map`. Każdy obraz powinien posiadać odpowiedni wpis w `/etc/lilo.conf` zaczynający się od `image` dla jąder i od `other` dla pośrednich programów ładujących. Pośredni program ładujący służy do uruchamiania sektorów początkowych innych partycji (a więc uruchamiania innych systemów operacyjnych). Struktura pojedynczego deskryptora została opisana poniżej:

```
typedef struct {
    char name[MAX_IMAGE_NAME+1]; /* 16 bajtów */
    char password[MAX_PW+1]; /* 16 bajtów */
```

```
    unsigned short rd_size[2]; /* rozmiar RAM dysku w bajtach, 0 jeśli nie ma */
    SECTOR_ADDR initrd,start; /* opisane niżej */
    unsigned short start_page; /* pierwsza strona pamięci (4kB), na którą ma być
                                  załadowane jądro, jeśli ładowane jest wysoko ;
  0 jeśli ładowane nisko */
    unsigned short flags,vga_mode;
} IMAGE_DESCR;
```

Cała struktura zajmuje 52 bajty, zatem w tablicy deskryptorów może być 19 wpisów, jednak dokumentacja LILO informuje o maksymalnej liczbie obrazów równej 16.
Pola initrd, oraz start są adresami pierwszego sektora adresów dyskowych, odpowiednio RAM-dysku i obrazu. Każdy sektor z tablicą adresów może zmieścić 100 adresów sektorów dyskowych. Ostatni adres jest adresem następnego sektora z tablicą adresów, a zatem sektory adresowe tworzą listę jednokierunkową na dysku. Dodatkowo, w przypadku pierwszego sektora z tablicą adresów dla obrazu, pierwsze dwa adresy dyskowe to dowiązania nie na sektory obrazu lecz na sektor **bezpiecznej linii poleceń** (ang. *fallback command line*) oraz sektor **domyślnej linii parametrów** (ang. *default parameter line*). Jeśli odpowiednia opcja jest aktywna, bezpieczna linia poleceń przy starcie wybranego obrazu jest kopiowana do sektora zawierającego domyślną linię poleceń (nie mylić z domyślną linią parametrów). Wszystkie sektory adresowe dla konkretnego obrazu wraz z sektorami bezpiecznej linii poleceń i domyślnej linii parametrów tworzą tzw. **sekcję** (spójny fragment) w pliku `/boot/map`. Sekcja może się składać również z sektorów adresowych konkretnego RAM-dysku. Pełna struktura wzajemnych wskazań umożliwiających załadowanie obrazu przez LILO przedstawiona jest na rysunku:

Po wczytaniu domyślnej linii poleceń, drugorzędny program ładujący przetwarza linię poleceń przekazaną przez użytkownika. Teraz zostaje odszukany odpowiedni obraz ładowalny i jest wczytywany sektor z domyślną linią parametrów pod adres `0x9d600`. Pod adresem `0x9d800` konstruowana jest bieżąca linia parametrów z domyślnej linii i z parametrów przekazanych w czasie startu przez użytkownika. Parametr lock w skonstruowanej linii parametrów spowoduje zapisanie podanej przez użytkownika linii jako nowej domyślnej linii poleceń. Jeśli deskryptor wybranego obrazu zawierał informacje o RAM-dysku, to RAM-dysk jest ładowany na koniec pamięci leżącej poniżej 16MB. Jest to koniec pamięci osiągalnej dla funkcji BIOS-u, wykorzystywanych do transferu RAM-dysku do pamięci. Używana pamięć znajduje się na końcu aby łatwo było ją w przyszłości zwolnić. Jeśli ładowanym obrazem jest jądro Linuksa, to pod adres `0x90000` wczytany zostaje sektor ładujący dla stacji dyskietek wkompilowany w jądro. Nie będzie uruchamiany kod w nim zawarty, lecz będą wykorzystywane dane w nim zawarte. Pod adres `0x90200` zostaje załadowana część jądra pochodząca z pliku **`setup.o`** odpowiedzialna za prawdziwy start systemu. Pozostała część jądra jest ładowana pod `0x10000`, chyba że pole `start_page` struktury `IMAGE_DESCR` miało wartość różną od zera. Wówczas jądro ładowane jest „wysoko" (ang. *high*) na stronę o podanym numerze (domyślnie jest to adres `0x100000`). Opcja ta używana jest w przypadku dużych jąder, których rozmiar jest większy niż różnica adresów, pod którymi umieszczany jest **`setup.o`** i pozostała część jądra. Po załadowaniu jądra sterowanie zostaje przekazane do części inicjującej (**`setup.o`**).
Jeśli jest ładowany inny system operacyjny, pośredni program ładujący zostaje wczytany pod `0x90200`, zaś sektor ładujący partycji, na której rezyduje uruchamiany system, umieszczony zostaje pod `0x90400`. Podobnie jak poprzednio sterowanie zostaje przekazane pod adres `0x90200` i wykonuje się kod pośredniego programu ładującego. Przesuwa on tablicę partycji zawartą w swoim kodzie pod adres `0x00600`, zaś sektor ładujący partycji pod `0x07c00` czyli w miejsce, w które BIOS



Rysunek 1.3: Zależności między sektorami odczytywanymi w czasie uruchamiania.

ładuje MBR. Następnie sterowanie zostaje przekazane pod `0x07c00` i następuje normalny start innego systemu.

W trakcie swego działania LILO wypisuje, po kolei litery `L`, `I`, `L`, `O`, po zakończeniu kolejnych faz wykonywania, co jest prostym sposobem informowania o przebiegu ładowania.

`L` po pierwszym przesunięciu,

`I` przed wczytaniem drugorzędnego programu ładującego (secondary boot loader), jeśli wystąpi błąd to wypisywany jest dwucyfrowy (hex) kod ostatniego błędu,

`L` po wczytaniu i uruchomieniu drugorzędnego programu ładującego,

`O` po utworzeniu tablicy deskryptorów (jeśli tablica ma złą sume kontrolną to „-”) i wczytaniu sektora z linią poleceń dla jądra

Aby LILO mogło dokonać załadowania systemu, musi ono wcześniej zostać odpowiednio skonfigurowane. Program lilo dokonuje tego wypytując Linuksa o położenie sektorów jąder na podstawie czego aktualizuje plik `/boot/map`. Jeśli przeniesiemy jądro do innych sektorów i nie uruchomimy lilo, plik `/boot/map` będzie zawierał nieaktualną informację. Podobnie jeśli przesuniemy plik `/boot/map` lub `/boot/boot.b` nieaktualna informacja znajdzie się w sektorze ładującym LILO, umieszczonym np. w MBR. W związku z powyższymi rozważaniami dochodzimy do wniosku, że

pewnym źródłem problemów będzie uruchomienie pod kontrolą DOS-u, na partycji współdzielonej z Linuksem, programu do defragmentacji partycji. Inne problemy może sprawić starsza wersja BIOS-u, która nie obsługuje dysków większych niż 8,4 GB (tzn. ma dostęp tylko do obszaru poniżej). Jeżeli jakiś z istotnych dla LILO sektorów zostanie wstawiony przez system plików powyżej tej granicy (Linux dostarcza własną niskopoziomową obsługę dysków więc może to zrobić) to system nie zostanie uruchomiony poprawnie (LILO korzysta z funkcji BIOS-u). Rozwiązaniem obydwu problemów jest wydzielenie małej partycji w obszarze dostępnym dla BIOS-u, nieużywanej przez inne systemy operacyjne i umieszczenie tam plików używanych przez LILO w czasie startu.
LILO posiada wiele opcji konfiguracyjnych, z których część podaje się w pliku `/etc/lilo.conf`, część przy uruchomieniu lilo, a część bezpośrednio w linii poleceń podczas startu. Bardzo szczegółowe informacje o ich zastosowaniu można znaleźć w podręczniku użytkownika ([Almesberger 98]).
Istotnym elementem konfiguracji jest wybranie miejsca gdzie ma znaleźć się podstawowy program ładujący, zawarty w sektorze ładującym LILO. Do wyboru mamy jedno z następujących miejsc:

- sektor ładujący na dyskietce;
- MBR na pierwszym dysku twardym;
- sektor ładujący na partycji linuksowej na pierwszym dysku twardym;
- pierwszy sektor partycji rozszerzonej (większość programów typu fdisk nie pozwala ustawić flagi aktywności dla partycji rozszerzonej, dlatego trzeba albo zrobić to ręcznie albo posłużyć się dołączonym do LILO programem activate);
- inne oryginalne lokacje;

Ze względu na sposób działania LILO może być umieszczony praktycznie w każdym wolnym miejscu na dysku (nieużywanym w czasie pracy przez żaden system). Musimy tylko zatroszczyć się o umieszczanie tam podstawowego programu ładującego złączonego z odpowiednimi danymi przez lilo, oraz o dostarczenie odpowiedniego programu, umieszczanego w MBR, który go stamtąd wczyta i uruchomi.

Loadlin

Opis programu odnosi się do wersji 1.5.

Loadlin jest programem ładującym Linuksa napisanym pod DOS. Korzysta on z obrazu jądra umieszczonego w jednym pliku na partycji dosowej. Po skompilowaniu jądra w Linuksie obraz ten mieści się w katalogu `arch/i386/boot`. Po przegraniu go na partycję dosową można już uruchomić Linuksa z DOS-a – bez modyfikowania boot-sektora twardego dysku, jak ma to miejsce w przypadku LILO.

Parametry

Jako iż loadlin jest zwykłym programem dosowym, można podawać mu parametry – z pliku bądź z linii poleceń. Loadlin rozpoznaje następujące ustawienia:

1. Nazwa pliku z jądrem – obowiązkowa.

2. Wielkość RAM-dysku, którego zawartość wczytywana jest przez jądro z pliku, znajdującego się najczęściej na dyskietce. Ramdysk jest później montowany jako główne urządzenie (ang. *root device*).
3. Tryb graficzny VGA pracy Linuksa. (80x25, 80x50 etc.)
4. Tryb zamontowania głownego dysku – `read-only` lub `read-write`.

Poza tym można przekazać parametry jądru – takie jak ilość dostępnej pamięci, parametry dla kart sieciowych, adresy IO do zarezerwowania, parametry dla kart dźwiękowych, ustawienie kontrolerów SCSI itp. Szczegółowy opis składni parametrów można znaleźć w dokumentacji do loadlina dostarczanej wraz z nim.

Działanie programu

Głównym zadaniem loadlina jest pozbycie się DOS-a, aby móc bez problemów uruchomić kod jądra. Opiszemy tu działania, które są wykonywane. Loadlin w trakcie pracy wykonuje wiele testów sprawdzających ustawienia sprzętowe. Wszystkie testy muszą zakończyć się pozytywnym wynikiem, aby można było załadować Linuksa.

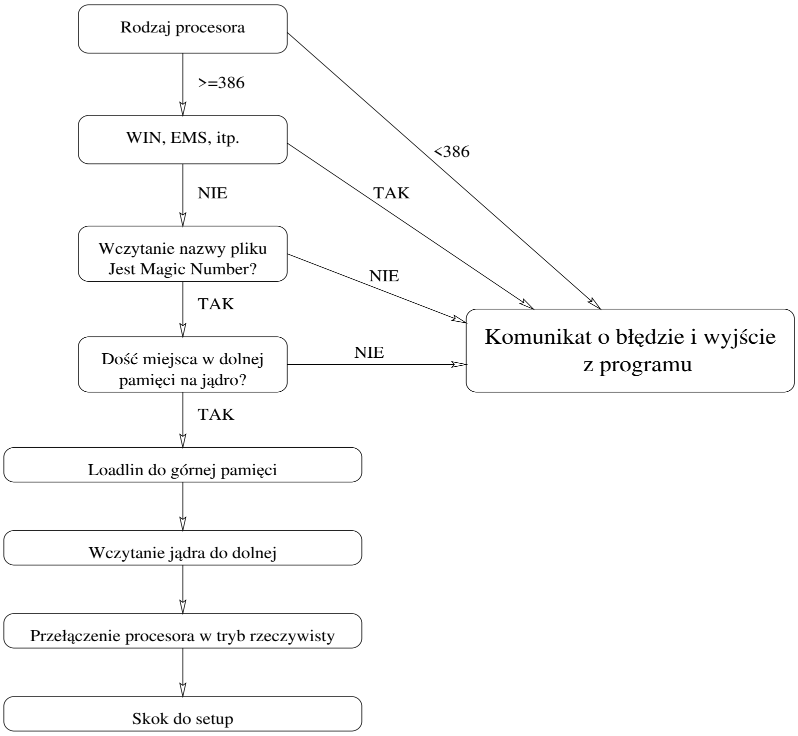
Najpierw sprawdzany jest rodzaj procesora. Linux potrzebuje 386 lub lepszego, bo dopiero one dysponują potrzebnymi mechanizmami stronicowania i segmentacji. Aby móc wykonać kod jądra, procesor musi być w trybie P0 – tzw. *supervisor mode*. Nie mogą być zainstalowane menadżery trybu chronionego, Windows itp. Program w sposób dość ciekawy sprawdza obecność systemu Windows – bada czy w środowisku ustawiona jest zmienna `WINDIR`. Sterownik EMS może być w pamięci, o ile jest zainstalowany jednocześnie serwer VCPI, który umożliwia przejście procesora w tryb P0 pomimo obecności sterownika EMS.

Następnie z pliku lub z parametrów podanych w linii poleceń wczytywana jest nazwa pliku, w którym znajduje się obraz jądra. Loadlin sprawdza jego poprawność – w bajcie `0x01FE` musi znajdować się tzw. Magic Number - `0xAA55`. Potem do pamięci wczytywany jest `setup` z jądra – dokładny opis `setup` znajduje się w rozdziale 1.1.5. Testowana jest następnie wielkość jądra – musi się ono zmieścić w dolnej pamięci. W odpowiednie miejsce w `setup` wstawiane są parametry opisane powyżej, podobnie jak robi to LILO. Następnie cały kod loadlina przesuwany jest do górnej pamięci, aby nie został zamazany po następującym chwilę później wczytaniu jądra do dolnej pamięci. Jeśli w parametrach została podana wielkość RAM-dysku, to loadlin prosi teraz o włożenie dyskietki z obrazem RAM-dysku do stacji. Następuje przełączenie procesora w tryb rzeczywisty, przywracane są domyślne przerwania BIOS-u i następuje skok do etykiety `START` w `setup`, czyli przejęcie kontroli przez jądro.

1.1.4 Proces ładowania jądra

Ładowanie jądra z dyskietki

Ładowanie jądra z dyskietki jest sprawą dość prostą, gdyż obraz jądra jest zapisany w kolejnych sektorach dyskietki, z której uruchomiono `bootsect`. 512-bajtowy kod sektora ładującego dyskietki jest ładowany przez BIOS pod adres `0x7c00`, przesuwa się pod `0x90000`, pod `0x90200` ładuje `setup`, a pod `0x10000 (0x100000)` system - `vmlinux.out` (`bvmlinux.out`). Aby przyspieszyć ładowanie, próbuje się odczytywać naraz całe ścieżki. Ilość sektorów na ścieżce jest ustalana „doświadczalnie", przez próbę czytania na raz 36, 18, 15 i ostatecznie 9 sektorów (co odpowiada stacjom 2.88 MB,



Rysunek 1.4: Schemat działania programu `loadlin`

1.44 MB, 1.2 MB i 720 KB). Czytanie jest wykonywane przez wywoływanie przerwania `0x13` BIOS-u. Po załadowaniu `setup` i ustaleniu rozmiaru ścieżki wypisywany jest komunikat „`Loading`" (znów BIOS, przerwanie `0x10`). Podobnie jak w innych miejscach kodu ładującego system, część błędów (np. błędy odczytu z dyskietki) może spowodować kręcenie się programu w nieskończonej pętli. Ostatnim wykonywanym w ramach pliku `bootsect` działaniem jest skok do etykiety `START` w pliku `setup`.

Ładowanie jądra z dysku twardego

W przypadku ładowania jądra z dysku twardego, pojawia się problem, skąd niewielki program łądujący ma wiedzieć, gdzie szukać obrazu jądra. Jeśli używamy programu `Loadlin`, jest to o tyle prostsze, że mamy działający system DOS i możemy po prostu korzystając z jego funkcji wczytać do pamięci plik, nie martwiąc się o jego fizyczne położenie na dysku.

Inaczej jest, gdy Linux ma być załadowany przez LILO, po starcie komputera. Nie ma jeszcze przecież dostępnego systemu plików, który mógłby zrobić całą czarną robotę, odszukując plik z obrazem jądra. Nie musi on przecież zajmować ciągłego obszaru, a więc trzeba w jakiś sposób dowiedzieć się o numerach zajmowanych przez niego sektorów. Sektor ładujący (`boot.b`) zawiera informacje o położeniu drugiego programu ładującego, ten zaś wie, gdzie szukać pliku z mapą (plik `/boot/map` w głównym katalogu). W pliku mapy znajdują się informacje o wszystkich zarejestrowanych przez LILO obrazach jądra, w tym o numerach zajmowanych przez nie sektorów. To właśnie między innymi ze względu na konieczność uaktualnienia tych informacji po kompilacji nowego jądra należy uruchomić `lilo`[8].

Ostatecznie, podobnie jak po załadowaniu systemu z dyskietki, następuje skok do etykiety `START` w pliku `setup`.

1.1.5 Przebieg budowy struktur jądra i tworzenia pierwszego procesu

`setup` jest odpowiedzialny za uzyskanie od BIOS-u podstawowych informacji o sprzęcie, które zapisywane są pod adresy `0x90000 - 0x901FF` – tam, gdzie poprzednio znajdował się `bootsect`. Później informacje te zostaną odczytane przez kod już w trybie chronionym, zanim obszar ten zostanie przydzielony na inne cele. `setup` zawiera na początku nagłówek, w którym zapisane są między innymi informacje o wymaganej wersji loadera i informacja, czy mamy do czynienia z dużym jądrem[9]. Korzystając z BIOS-u dowiadujemy się, jaki jest rozmiar pamięci.

Następnie inicjalizowana jest karta graficzna, sprawdzany jest jej rodzaj i uzyskane informacje zapisywane są do późniejszego wykorzystania.

Sprawdzana jest ilość i parametry dysków twardych podłączonych do pierwszego kontrolera, obecność architektury MCA (Micro Channel Architecture - szyna systemowa stosowana w pierwszych PS/2), myszy PS/2, BIOS-u APM (Advanced Power Managment). Teraz czeka nas przełączenie procesora w tryb chroniony. W tym celu najpierw ładowane są lokalna i globalna tablica deskryptorów. Lokalna na razie nie będzie używana, więc jej wskaźnik jest inicjowany na 0, globalna będzie zawierać deskryptory dwóch 4 GB-owych segmentów zaczynających się od adresu fizycznego `0x00` – segment kodu i danych jądra. Przed przełączeniem się w tryb chroniony należy jeszcze wyłączyć przerwania, w tym przerwanie niemaskowalne (przez programowanie kontrolera przerwań). Teraz włączamy linię adresową A20, która umożliwia dostęp do pamięci powyżej 1 MB.

Pozostało jeszcze kilka szczegółów – reset koprocesora oraz przeprogramowanie wektorów przerwań, gdyż BIOS ustawia je tak, że przerwania programowe zachodzą na sprzętowe – przesuwamy je więc za sprzętowe, od adresu `0x20`. Samo przełączenie w tryb chroniony polega na ustawieniu najmniej znaczącego bitu w słowie stanu procesora.

Ostatnią wykonywaną instrukcją, już w trybie chronionym, jest skok pod adres `0x1000` lub (`0x100000` w wypadku dużego jądra), gdzie `setup` wcześniej przesunął skompresowane jądro. Trafiamy więc do punktu `startup_32` w pliku `head.o`. Plik ten w przypadku dużego jądra rezyduje w miejscu, gdzie później będzie katalog stron, zostanie więc nadpisany. Ustawiane jest kilka parametrów potrzebnych do wykonania kodu rozkompresowującego jądro, napisanego w C – np. segmenty CS, DS, BSS. Jeśli mamy do czynienia z dużym jądrem, po dekompresji trzeba je będzie przesunąć w odpowiednie miejsce, tzn. do `0x100000` (małe będzie tam umieszczane od razu). Kod dekompresujący korzysta z kilku funkcji w C, np. tych wykonujących rzeczywisty algorytm dekompresji PKZIP metody 8, zamieszczonych w `lib/inflate.c`. W tym celu w `arch/i386-/boot/compressed/misc.c` zaimplementowane są np. własne funkcje `malloc()` i `free()`. `free()` nie robi nic, natomiast `malloc()` po prostu zwiększa globalny wskaźnik o podany rozmiar i zwraca go. Własna funkcja `puts()` wykorzystywana jest między innymi do wypisywania znanego wszystkim komunikatu „`Uncompressing Linux...`”, a po skończonej dekompresji „`Ok, booting the kernel.`”

[8] Zobacz też rozdział o programach ładujących 1.1.3.

[9] Informacja ta została uzyskana podczas kompilacji z pliku `Makefile` i wstawiona do kodu przy pomocy dyrektywy preprocesora

Następuje teraz krótki powrót do pliku `head.o` i po ewentualnym przesunięciu dużego jądra skaczemy pod `0x100000`, do etykiety `startup_32` we właściwym pliku `head.o`. Poprzednio załadowane pliki `bootsect`, `setup`, `head.o` i `misc.o` już nie będą potrzebne, część z nich została już zresztą zamazana.

Pod adresem `0x101000` ustawiany jest katalog stron, zawierający wskaźniki na dwie tablice – 0-wą dla użytkownika i 768-ą dla jądra. Pozostałe wpisy są zerowane. Oba katalogi wskazują na tę samą tablicę stron pod `0x102000`. Tablica te realizuje mapowanie identycznościowe, tzn. kolejne strony odpowiadają kolejnym ramkom, począwszy od zerowej. W ten sposób mamy dostępne 1024 strony, czyli 4 MB pamięci. Po tej operacji włączany jest bit mapowania w słowie stanu procesora.

Kolejną sprawą jest ustawienie wszystkich wektorów przerwań na pustą domyślną funkcję `ignore_int()`, której jedynym zadaniem jest wypisywanie za pomocą funkcji `printk()` komunikatu „`Unknown interrupt`".

Następnie do strony zerowej (`empty_zero_page`) kopiowane są parametry przekazane do jądra przez program ładujący i przez użytkownika podczas ładowania systemu.

Typ procesora jest sprawdzany przez testowanie odpowiednich flag, sprawdzana jest też obecność koprocesora. Informacje te są zapisywane do rekordu `boot_cpu_data` do późniejszego wykorzystania przez jądro.

Przesuwanie jądra spowodowało, że informacja o adresie lokalnej (*LDT* – ang. *Local Descriptor Table*) i globalnej (*GDT* – ang. *Global Descriptor Table*) tablicy deskryptorów jest nieaktualna, więc należy ustawić ją jeszcze raz. Ustawiane są rejestry segmentowe. Deskryptor LDT w GDT jest zerowany.

Teraz dopiero zaczyna się właściwa inicjalizacja jądra, przez skok do funkcji `start_kernel()` z pliku `init/main.c`. Przeprowadza ona inicjalizację wszystkich struktur jądra.

Funkcja `start_kernel()`

Uwaga o pamięci:
Procedury inicjalizacyjne, przyjmują parametry `memory_start, memory_end`. Rezerwują one na własne potrzeby pamięć z początku, zwracając zmodyfikowany wskaźnik `memory_start`.

printk(linux_banner);

Wypisanie linuksowego tekstu powitalnego.

setup_arch(&command_line, &memory_start, &memory_end);

- pobieranie informacji o sprzęcie (procesor, płyta główna, BIOS) – przeważnie odczytywane są informacje ustalone wcześniej na etapie bootowania (`setup`)
- ustawiane są pola struktury `init_task.mm`.
- odczytywanie rozmiaru pamięci
  Wykorzystywane są do tego informacje BIOS-u, lecz można je przeciążyć za pomocą linii poleceń. Rozmiar pamięci można podać w bajtach, kilobajtach lub megabajtach.
- Ustalanie rozmiaru `initrd` (czyli RAM-dysku, na którym montuje się korzeń systemu plików). Sprawdza się, czy obszar przeznaczony na RAM-dysk nie wystaje poza pamięć.
- Rezerwowanie portów dla standardowych urządzeń (dma, timer, fpu). Wywoływana jest funkcja `request_region`.

- Ustawianie funkcji odpowiedzialnej za przełączanie konsol wirt. (zmienna `conswitchp`).

paging_init(memory_start,memory_end);
Przygotowuje grunt pod stronicowanie. Katalog stron i tablice stron są ustawiane tak, że odwzorowują kolejne ramki pamięci fizycznej. Uwzględnia się to, że pierwsze 4 MB zostały już zamapowane przez `head.o`. Strona odpowiadające adresowi wirtualnemu 0 jest odmapowywana, aby wyłapywać odwołania do NULL. Jeżeli wykryje się obecność procesora Pentium, wykorzystywane są strony o rozmiarze 4 MB (lecz tylko na potrzeby jądra – zwykłe procesy nadal działają na stronach o zwykłym rozmiarze).

trap_init();
Ustawienie bramek dla przerwań, w tym bramki dla wywołań systemowych.

init_IRQ();
Inicjalizacja przerwań. Przypisywane są domyślne handlery przerwaniom sprzętowym. Następnie ustawiane są bramki dla przerwań zewnętrznych (za wyjątkiem obsługującego wywołania systemowe).

sched_init();
Inicjalizuje tablicę procesów oraz tablicę haszującą dla PID-ów procesów. Instaluje też procedury obsługi przerwania zegarowego (ich dolne połowy) – funkcja `init_bh`.

time_init();
Pobiera z CMOS-u czas rzeczywisty i inicjalizuje nim strukturę `xtime`.

parse_options(command_line);
Przetwarza opcje podane w linii poleceń jądra.

console_init(memory_start, memory_end);
Inicjalizuje osługę konsoli. Ustawia domyślną dyscyplinę linii (`tty_register_ldisc`) oraz strukturę opisującą typ terminala. Następnie inicjalizuje konkretne terminale: wirtualne lub szeregowe.

init_modules();
Inicjalizuje obsługę modułów. Pola struktury `kernel_module` mają przeważnie stałe wartości, inicjalizowane jest jedynie pole `nsyms` (ilość symboli w tablicy jądra).

kmem_cache_init(memory_start, memory_end);
Inicjalizuje pamięć podręczną dla jądra.

sti();
Włączenie przerwań.

calibrate_delay();
Wyznacza BOGO-Mipsy. Ten test polega na zmierzeniu szybkości procesora w kręceniu się w pustej pętli. Wyznaczona wartość przydaje się niektórym sterownikom urządzeń.

Sprawdzenie czy nie został zamazany obraz RAM-dysku.
Mogło tak się stać, gdyż procedury wywoływane powyżej mogły zabrać ten obszar pamięci na swoje potrzeby.

mem_init(memory_start,memory_end);
Inicjalizuje tablicę `mem_map` (wywołując `clear_bit`). Zaznacza strony zajęte przez jądro (`PG_reserved`), strony możliwe do wykorzystania przez mechanizm DMA (`PG_DMA`). Następnie wypisuje informację statystyczną (ilość dostępnej pamięci, w tym ilość pamięci zajętej przez jądro, dane, inicjalizację). Potem sprawdza się, czy procesor poprawnie interpretuje zabezpieczanie stron przed zapisem (nie robią tego modele 386 i niektóre dziwne 486). Błąd polega na ignorowaniu tego zabezpieczenia kiedy procesor pracuje w trybie jądra – wymaga to stosowania w trybie jądra dodatkowych testów zamiast czystego algorytmu *copy_on_write*.

kmem_cache_sizes_init();
Dalszy ciąg inicjalizacji pamięci podręcznej jądra.

proc_root_init();
Rejestrowanie funkcji obsługi dla systemu plików proc;

Rezerwacja pamięci podręcznej (przy pomocy `kmem_cache_create`) oraz inicjalizacja ewentualnych tablic haszujących dla:

- systemu zarządzania procesami.
  `uidcache_init();`
  `filescache_init();`
- systemu plików
  `dcache_init();`
- systemu zarządzania pamięcią
  `vma_init();`
- buforów systemowych
  `buffer_init(memory_end-memory_start);`
- kolejki sygnałów
  `signals_init();`
- i znów systemu plików
  `inode_init();`
  `file_table_init();`

- Wstawianie w tablice struktur IPC wartości `IPC_UNUSED`
  `ipc_init();`
- Sprawdzenie błędów procesora
  `check_bugs();`
  Sprawdzany jest tu między innymi głośny swego czasu błąd `F00F` występujący w starszych procesorach Pentium. Linux został uodporniony na ten błąd niemal natychmiast po jego wykryciu.
- Linux szczyci się zgodnością ze standardem POSIX. Nie są tu wykonywane żadne testy – wypisuje się tylko tekst.
  `printk("POSIX conformance testing by UNIFIX\n");`

- Uruchamianie wątku jądra, który uruchamia pierwszy proces – funkcję `init()`.
  `kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);`

- Powiadomienie funkcji szeregującej, że już czas zacząć pracę.
  `current->need_resched = 1;`

- Marnowanie czasu procesora.
  `cpu_idle(NULL);`
  Jeśli sprzęt to umożliwia, zamiast kręcić się w pustej pętli procesor wykonuje instukcję `halt`, co powoduje zmniejszenie poboru mocy. Jest to szczególnie ważne dla komputerów przenośnych, a komputery stacjonarne czyni odporniejszymi na przetaktowanie[10] (ang. *overclocking*).

Funkcja `init()`

Jest to pierwsza funkcja, która staje się procesem z prawdziwego zdarzenia. Najważniejszą sprawą jest wywołanie funkcji `do_basic_setup()`, która wykonuje następujące czynności:

- zawiadamia, że wątek `init()` będzie adoptował [11] osierocone procesy:
  `child_reaper = current;`

- rozpoczyna obsługę magistrali systemowych (ISA, PCI itp...) oraz dołączonych do nich urządzeń.

- inicjalizuje obsługę sieci `sock_init();`

- uruchamia demony:

```
/* Launch bdflush from here, instead of the old syscall way. */
kernel_thread(bdflush, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kupdate, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
/* Start the background pageout daemon. */
kswapd_setup();
kernel_thread(kpiod, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
kernel_thread(kswapd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
```

- wykonuje inne niezbędne czynności:

```
/* Inicjalizacja urządzeń.. */
device_setup();

/* .. formatów plików wykonywalnych .. */
binfmt_setup();
```

[10] Linuks jest bardzo wrażliwy na błędy sprzętowe i w związku z tym raczej nie powinno się używać go z „podkręconym” procesorem.

[11] Autorzy ujmują to nieco inaczej: *„Tell the world that we're going to be the grim reaper of innocent orphaned children.”* – co dowodzi, że uwielbiają się oni znęcać nie tylko nad nieszczęsnymi studentami usiłującymi cokolwiek zrozumieć na podstawie źródeł :-).

```
/* .. systemów plików .. */
filesystem_setup();

/* montowanie korzenia systemu plików.. */
mount_root();
```

Po powrocie do funkcji init():

- otwiera się konsolę i przekierowuje się na nią `stdout` i `stderr`:

```
if (open("/dev/console", O_RDWR, 0) < 0)
      printk("Warning: unable to open an initial console.\n");

(void) dup(0);
(void) dup(0);
```

- wybiera się program, który zajmie się dalszym działaniem systemu. Jako pierwszy jest brany pod uwagę program wskazany w linii poleceń.

```
if (execute_command)
        execve(execute_command,argv_init,envp_init);
execve("/sbin/init",argv_init,envp_init);
execve("/etc/init",argv_init,envp_init);
execve("/bin/init",argv_init,envp_init);
execve("/bin/sh",argv_init,envp_init);
panic("No init found.  Try passing init= option to kernel.");\\
```

Na tym proces uruchamiania systemu się kończy, sterowanie zostało przekazane np. do programu `init`, który rozpoczyna czytanie plików konfiguracyjnych.

1.2 Obsługa przerwań i wywołań systemowych w Linuksie

1.2.1 Wprowadzenie

Po co są przerwania? z przerwań korzysta się w celu umożliwienia sprzętowi komunikowania się z systemem operacyjnym.

W trybie z przerwaniami urządzenie informuje procesor za pośrednictwem kanału przerwań (IRQ) o zakończeniu działania. Wstrzymuje on wtedy wykonywanie bieżącej czynności i wykonuje procedurę obsługi danego przerwania (ISR). Następnie komunikacja z urządzeniem odbywa się wewnątrz procedury obsługi przerwania (ISR). Procedury obsługi przerwań są uaktywniane wtedy, gdy sprzęt (urządzenie) sygnalizuje sytuację wyjatkową np. wpisanie nowych znaków z klawiatury lub sygnał pochodzący z zegara, generowany co 10 ms.

W systemie Linux występują dwa typy przerwań:

- **przerwania sprzętowe** (ang. *hardware interrupts*), zgłaszane są przez sterownik przerwań na żądanie urządzenia (np. zegara lub klawiatury).

- **wyjątki** (ang. *exceptions*), generowane są przez sam procesor; dzielą się na:
  - **niepowodzenia** (ang. *faults*), zgłaszane przed pełnym wykonaniem instrukcji i umożliwiające jej powtórzenie. Dobrym przykładem jest tu wyjątek PF (ang. *Page Fault*), powstający przy odwołaniu do strony nieobecnej w pamięci.
  - **potrzaski** (ang. *traps*), sygnalizowane po wykonaniu instrukcji, gdy spełnione są wyzwalające je warunki. Przykładem jest instrukcja `int`, która zawsze podnosi wyjątek
  - **załamania** (ang. *aborts*), powstające w wyniku poważnych i nienaprawialnych błędow; np. wyjątek DF (ang. *Double Fault*) podnoszony jest wtedy, gdy przed zakończeniem obsługi niepowodzenia zgłoszone zostaje następne

1.2.2 Obsługa wyjątków w Linuksie

Wyjątek, w odróżnieniu od przerwań sprzętowych, jest generowany przez procesor. Zwykle przyczyną wyjątku jest błąd w wykonaniu instrukcji. Niektóre wyjątki są podnoszone jako reakcja na pewne specjalne instrukcje, np. bound. Podczas podnoszenia niektórych wyjątków procesor umieszcza na stosie dodatkowo kod błędu. Ponieważ wyjątki mają przypisane wektory, tak jak przerwania, można wywołać procedurę obsługi wyjątku instrukcja `int n`, np `int 0x80` wywołuje funkcję `system_call`. w tym przypadku żaden kod błędu nie jest umieszczany na stosie.

Podprogram obsługi danego wyjątku składa się z dwóch częsci: zdefiniowany w pliku `entry.S` fragment kodu wywołuje w trybie jądra funkcje `do_nazwa_wyjatku()` (zdefinowaną zwykle w pliku `traps.c`) i poprzez algorytm `ret_from_sys_call` wraca do przerwanego procesu. Większość podprogramów obsługi generowana jest przez makro `DO_ERROR()`, które woła funkcję `force_sig()` z odpowiednim numerem sygnału. Jeśli wyjątek spowodowało jądro, to procedura `die_if_kernel` zatrzyma system. Wyjątki i odpowiadające im podprogramy opisano w tabeli.

Tablica 1.1: Podprogramy obsługi wyjątków

Wyjątek	Sygnał	Nazwa podprogramu obsługi
0	SIGFPE	do_divide_error
3	SIGTRAP	do_int3
4	SIGSEGV	do_overflow
5	SIGSEGV	do_bounds
7	SIGSEGV	do_device_not_available
8	SIGSEGV	do_double_fault
9	SIGFPE	do_coprocessor_segment_overrun
10	SIGSEGV	do_invalide_TSS
11	SIGBUS	do_segment_not_present
12	SIGBUS	do_stack_segment
17	SIGSEGV	do_alignment_check

- wyjątek 1 (praca krokowa) obsługuje funkcja `do_debug()`, która woła `force_sig(SIGTRAP, current)`

- wyjątek 2, będący w rzeczywistości przerwaniem niemaskowalnym (NMI), obsługuje funkcja `do_nmi()`, wyświetlająca komunikat o potencjalnych problemach z pamięcia RAM
- wyjątek 14 obsługuje funkcja `do_page_fault()`
- wyjątek 16 obsługuje funkcja `do_coprocessor_error()`, ktora po zebraniu argumentów woła (pośrednio) `force_sig(SIGFPE)` w kontekście procesu, który ostatni używał koprocesora.
- wyjątki 15 i od 18 do 47 są zarezerwowane. Wykonywana po ich zgłoszeniu funkcja `do_reserved()` wyświetla stosowny komunikat.

1.2.3 Mechanizm wołania funkcji systemowych

Można powiedzieć, że istnieje jeszcze jeden rodzaj przerwań, mianowicie wołania systemowe. Za ich pośrednictwem jest udostępnianych wiele funkcji systemu operacyjnego. Wołanie systemowe opiera się na przejściu z trybu użytkownika do trybu systemowego. w systemie Linux jest to możliwe tylko dzięki przerwaniom. Z tego tez powodu przerwanie 0x80 jest zarezerwowane dla wołań systemowych.

Normalnie, do wykonania określonego zadania, użytkownik wywołuje funkcję biblioteczną (np. `fork()`). Ta funkcja zapisuje swoje argumenty i numer w odpowiednich rejestrach i generuje przerwanie `0x80`. Gdy procedura obsługi przerwania zwróci sterowanie, zwracana jest wartość odczytywana z odpowiednich rejestrów i kończone jest wykonywanie funkcji bibliotecznej. W zasadzie cała praca wołania systemowego jest wykonywana w procedurze obsługi przerwania. Oto jej pseudokod w C. W rzeczywistości jest napisana w asemblerze.

```
system_call(int sys_call_num, sys_call_args)
{
```

Najpierw zapisywane są wszystkie rejestry procesu

```
SAVE_ALL;           /* makro w entry.s */
```

Jeśli `sys_call_num` jest wartością poprawną, to wołana jest procedura obsługi przypisana do tego numeru wołania systemowego. Jest ona w polu `sys_call_table[]` w pliku `arch/i386-/kernel/entry.s` Jeśli ustawiona jest flaga `PF_TRACESYS` procesu, jest on monitorowany przez swój proces macierzysty, tym zajmuje się funkcja `sys_call_trace()` z pliku `arch/i386/kernel-/ptrace.c`

```
if (sys_call_num >= NR_syscalls)
    errno = -ENOSYS;
else {
    if (current -> flags & PF_TRACESYS) {
syscall_trace();
errno = (* sys_call_table[sys_call_num])(sys_call_args);
syscall_trace();
    }
    else
errno = (* sys_call_table[sys_call_num])(sys_call_args);
```

Rzeczywiste zadanie wołania systemowego jest w tym miejscu skończone. Jednak zanim będzie można kontynuować wykonywanie procesu, trzeba jeszcze zająć się kilkoma „administracyjnymi sprawami". Wykonać zgłoszone dolne połowy przerwań

```
ret_from_sys_call:
{
    if (bh_mask & bh_active)
handle_bottom_half();
}
```

W razie potrzeby wywoływany jest program szeregujący. Powoduje to uaktywnienie innego procesu. Funkcja `schedule()` zwróci sterowanie tylko wtedy, gdy proces zostanie ponownie uaktywniony przez program szeregujący.

```
if (need_resched)
{
    schedule();
    goto ret_from_sys_call;
}
```

Jeśli do bieżącego procesu zostały wysłane sygnały i nie zablokował on ich odbierania, są teraz przetwarzane.

```
if (current -> signal & ~current -> blocked)
{
    do_signal();
}
```

Teraz wszystkie rejestry są odtwarzane, a procedura obsługi przerwania jest kończona instrukcją `iret`.

```
exit_now:
    RESTORE_ALL;
} /*   system_call()   */
```

1.2.4 Obsługa przerwań sprzętowych w Linuksie

Rezerwacja przerwania i rejestracja procedury obsługi przerwania

Dla wszystkich podprogramów obsługi dostępny jest wspólny interfejs rezerwacji i rejestracji procedury obsługi. Procedura obsługi którą rejestruje dany podprogram musi mieć następujący typ:

```
void (*handler)(int, void *, struct pt_regs *);
```

Parametry oznaczają kolejno: numer przerwania, typ urządzenia (co ma znaczenie przy współdzieleniu przerwań) i wskaźnik do struktury opisującej zawartość rejestrów procesora.

Przerwania mogą być dzielone między kilka podprogramów obsługi urządzeń. Po zgłoszeniu przerwania są one wywoływane po kolei z drugim parametrem będącym wskaźnikiem do struktury opisującej urządzenie, a przekazanej podczas rejestrowania procedury obsługi. Dzięki temu procedura jest w stanie rozpoznać czy przerwanie pochodzi od jej urządzenia. Podprogramy umożliwiające współdzielenie przerwań rejestrują swoje procedury obsługi z flagą `SA_SHIRQ`. Wszystkie muszą zgodzić się na współdzielenie przerwania.

Funkcją rejestrującą jest:

```
int request_irq(unsigned int irq,
    void (*handler)(int, void *, struct pt_regs *),
    unsigned long irqflags,
    const char * devname,
    void *dev_id)
```

Znaczenia parametrów:

- `irq` – numer przerwania
- `handler` – procedura obsługi
- `irqflags` – flagi `SA_*`
- `devname, dev_id` – nazwa i opis urządzenia

Funkcja zwraca:

- `0` w przypadku powodzenia
- `-EINVAL` w przypadku błędnych parametrów
- `-ENOMEM` gdy nie można zaalokować struktury `irqaction`
- `-EBUSY` gdy jest już zarejestrowana procedura obsługi i niemożliwe jest współdzielenie

Usunięcie procedury obsługi realizowane jest za pomocą funkcji:

```
void free_irq(unsigned int irq, void *dev_id)
```

- `irq` - numer przerwania
- `dev_id` - dotyczy rodzaju obsługiwanego urządzenia

Do przechowywania informacji o procedurach obsługi służy następująca struktura

```
struct irqaction {
    void (*handler)(int, void *, struct pt_regs *);
    unsigned long flags;
    unsigned long mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
};
```

Pola zawierają wartości przekazane do funkcji `request_irq()`. W polu `next` znajduje się wskaźnik do struktury opisującej kolejną zarejestrowaną dla danego przerwania procedurę obsługi.

Niskopoziomowa obsługa przerwań

Powyżej zostały przedstawione procedury służące do rejestracji procedur obsługi przerwań. w tym punkcie zostanie omówiony kod i struktury jądra odpowiadające m.in. za wywoływanie tych funkcji.

Linux posiada niezależną od sprzętowego kontrolera architekturę obsługi przerwań zewnętrznych. Dla każdego typu kontrolera jądro przechowuje następującą strukturę:

```
struct hw_interrupt_type {
    const char * typename;
    void (*startup)(unsigned int irq);
    void (*shutdown)(unsigned int irq);
    void (*handle)(unsigned int irq, struct pt_regs * regs);
    void (*enable)(unsigned int irq);
    void (*disable)(unsigned int irq);
};
```

znajdują się w niej wskaźniki do funkcji wykonujących czynności zależne od typu kontrolera, a związane z obsługą (`handle`), włączeniem (`enable`) i wyłączeniem (`disable`) konkretnego przerwania.

W jądrze 2.2.12 zaimplementowana jest obsługa sterownika 8259A zwanego **PIC** (ang. *Programmable Interrupt Controler*) występującego w prawie wszystkich maszynach o architekturze wywodzącej się z PC-XT. W wyżej opisywanej strukturze dotyczącej tego typu kontrolera w polach `startup` i `shutdown` znajdują się te same wskaźniki co w polach `enable` i `disable`.

Dla każdego przerwania jądro przechowuje w tablicy `irq_desc[NR_IRQS]` następującą strukturę

```
typedef struct {
    unsigned int status; /* IRQ status - IRQ_INPROGRESS, IRQ_DISABLED */
    struct hw_interrupt_type *handler; /* handle/enable/disable functions */
    struct irqaction *action; /* IRQ action list */
    unsigned int depth; /* Disable depth for nested irq disables */
    unsigned int unused[4];
} irq_desc_t;
```

W polu `status` przechowywana jest suma logiczna następujących flag:

```
/*
 * IRQ line status.
 */
#define IRQ_INPROGRESS 1 /* IRQ handler active - do not enter! */
#define IRQ_DISABLED 2 /* IRQ disabled - do not enter! */
#define IRQ_PENDING 4 /* IRQ pending - replay on enable */
#define IRQ_REPLAY 8 /* IRQ has been replayed but not acked yet */
#define IRQ_AUTODETECT 16 /* IRQ is being autodetected */
#define IRQ_WAITING 32 /* IRQ not yet seen - for autodetection */
```

Ustawiona flaga `IRQ_INPROGRESS` oznacza, że przerwanie jest obsługiwane. Ustawiona flaga `IRQ_DISABLED` oznacza, że przerwanie jest wyłączone, zablokowane. Pozostałe flagi są wykorzystywane przy czynnościach związanych z wykrywaniem przerwań.

W polu `handler` znajduje się wskaźnik do struktury z funkcjami obsługi zależnymi od typu sterownika. W polu `action` lista zarejestrowanych procedur obsługi.

W IDT (ang. *Interrupt Description Table*) znajdują sie funkcje , które realizują następujący algorytm:

- odkładają na stos numer przerwania
- zapamiętują rejestry procesora
- wołają funkcję `do_IRQ()`
- odtwarzają rejestry, a w razie potrzeby wołają funkcje `schedule()`

Wyjątkiem jest przerwanie zegarowe, które jest obsługiwane w sposób specjalny.

Funkcja `do_IRQ()`:

- pobiera ze stosu numer przerwania `irq`
- zwiększa licznik obsłużonych przerwań
- woła `irq_desc[irq].handler->handle(irq, &regs);` (patrz opis `irq_desc_t`)
- w razie potrzeby woła `do_bottom_half();`

W polu `handle` struktury `hw_interrupt_type` dotyczącej sterownika 8259A znajduje się wskaźnik do funkcji `void do_8259A_IRQ(unsigned int irq, struct pt_regs * regs)`, która wykonuje następujące czynności (rzeczywista obsługa przerwania):

- sprawdza czy przerwanie ma ustawioną flagę `IRQ_DISABLED | IRQ_INPROGRESS` – jeśli tak, kończy działanie
- dla każdej zarejestrowanej procedury obsługi
  - w zależności od ustawionej (lub nie) flagi `SA_INTERRUPT` ustawia (lub nie) flagę IF procesora
  - wywołuje ją
- jeśli któraś z procedur miała ustawioną flagę `SA_SAMPLE_RANDOM` wywołuje funkcję

  `add_interrupt_randomness()`, która zwiększa losowość wbudowanego w Linuksa generatora liczb pseudolosowych
- zeruje flagę IF procesora
- jeśli przerwanie nie ma ustawionej flagi `IRQ_DISABLED` zostaje odblokowane

Dolne i górne połowy

Przy obsłudze przerwań nie zawsze koniecznym jest, by wszystkie funkcje były wykonywane bezpośrednio po wystąpieniu przerwania. Często zdarza się, że naprawdę istotne jest wykonanie się jednych funkcji, natomiast reszta może być wykonana później (np. kopiowanie danych do bufora w pamięci). Wykonanie ich natychmiast nie jest niezbędne, a zajęłoby względnie dużo czasu i niepotrzebnie blokowałoby przerwanie.

W celu wykorzystania takiej sytuacji stworzono koncepcję „górnych i dolnych połów".

Przez **górne połowy** rozumiemy funkcje, które powinny się wykonać bezpośrednio, natychmiast po wystąpieniu przerwania, **dolne połowy** (ang. *bottom halves*) to zaś te podprogramy, które zostaną wykonane w późniejszym czasie z odmaskowanymi przerwaniami.

Dolne połowy umieszczone są w ogólnosystemowej tablicy wskaźników do funkcji dolnych połów `void (* bh_base[32])(void)`. w trakcie obsługi przerwania górna połowa zaznacza przy wykorzystaniu funkcji `mark_bh(int nr)` do wykonania pewne dolne połowy, po czym może zająć się wykonywaniem swoich zadań. Przy najbliższej okazji, czy to po `ret_from_syscall`, czy też funkcji `schedule()`, wszystkie zaznaczone części zostaną wykonane. Po wykonaniu funkcji jest ona zaznaczana jako nieaktywna. Części te mają własność atomiczności, co oznacza, że tak długo jak dolna część jest aktywna, nie można wykonać żadnej z pozostałych. Powoduje to, że nie jest konieczne użycie `cli()` do zabezpieczenia ich przed przerwaniami. Każda dolna część musi być zainstalowana. Służy do tego funkcja `init_bh(int nr ,void (*routine)(void))`, która umieszcza dolną połowę w tablicy `bh_base`. Robi się to zwykle już w chwili zajmowania przerwania w funkcji inicjującej urządzenie.

Kolejki zadań

Ponieważ istniał problem z ograniczoną ilością dolnych połów, poszukiwano rozwiązania mogącego temu zaradzić. Tak powstała koncepcja **kolejek zadań** (ang. *task queues*).

Kolejki zadań umożliwiają umieszczenie w kolejce określonej liczby funkcji w celu późniejszego wykonania.

Istnieje metoda polegająca na umieszczeniu funkcji obsługi kolejki w kolejce `TQ_IMMEDIATE`, która ma implementować i zastępować stary system obsługi dolnych połów. Na kolejki te można wstawiać dowolne funkcje przy wykorzystaniu struktury `tq_struct`. Struktura ta ma postać:

```
struct tq_struct {
    struct tq_struct *next;
    int sync;
    void (*routine)(void *);
    void *data;
};
```

Gdzie:

- `next` - wskaźnik do następnego elementu kolejki
- `sync` - flaga synchronizacji, zabezpieczenie przed wstawieniem tej samej struktury kilka razy
- `routine` - funkcja, która ma być wywoływana
- `data` - argument przekazywany do funkcji `routine`.

Przetwarzaniem kolejki zajmuje się funkcja: `void run_task_queue(task_queue *list)`. Przetwarza ona wszystkie struktury w kolejce, wywołując zapisane w nich funkcje. Do uruchomienia zdefiniowanej kolejki zadań, funkcja obsługująca kolejkę zadań wykonywana jest przy wykorzystaniu kolejki `TQ_IMMEDIATE`. Dzieje się to w miejscu, gdzie wywoływana jest obsługa dolnych połów.

1.3 Niskopoziomowa obsługa błędów braku i ochrony strony

1.3.1 Wsparcie sprzętowe

Obsługa błędów braku i ochrony strony wymaga wsparcia sprzętowego.
Wsparcie takie zapewniają procesory rodziny 386. W trybie chronionym przy włączonym stronicowaniu adresy liniowe generowane przez procesor przechodzą sprzętową translację do adresów fizycznych. Odbywa się ona pod kontrolą zewnętrznych struktur danych zwanych katalogiem i tablicami stron. Struktury te, oprócz swojej podstawowej funkcji polegającej na określaniu sposobu odwzorowania adresów liniowych w fizyczne dają możliwość zdefiniowania stanu każdej ze stron. Podczas translacji informacje o stanie strony są odczytywane przez procesor i na ich podstawie może zostać wygenerowany wyjątek zwany niepowodzeniem stronicowania. Procesor zachowuje się wtedy podobnie jak przy wywołaniu przerwania, to znaczy skacze do procedury obsługi, wczesniej jednak zapamietuje na stosie adres instrukcji, która wygenerowała ten wyjątek oraz odkłada na stos kod błędu. Kod błędu umożliwia stwierdzenie przyczyny niepowodzenia stronicowania, natomiast adres powrót do tej samej instrukcji tak że proces, który był wykonywany przy generacji wyjątku może być wykonywany nadal. Wyjątek niepowodzenia stronicowania na procesorach rodziny 386 posiada numer 14.

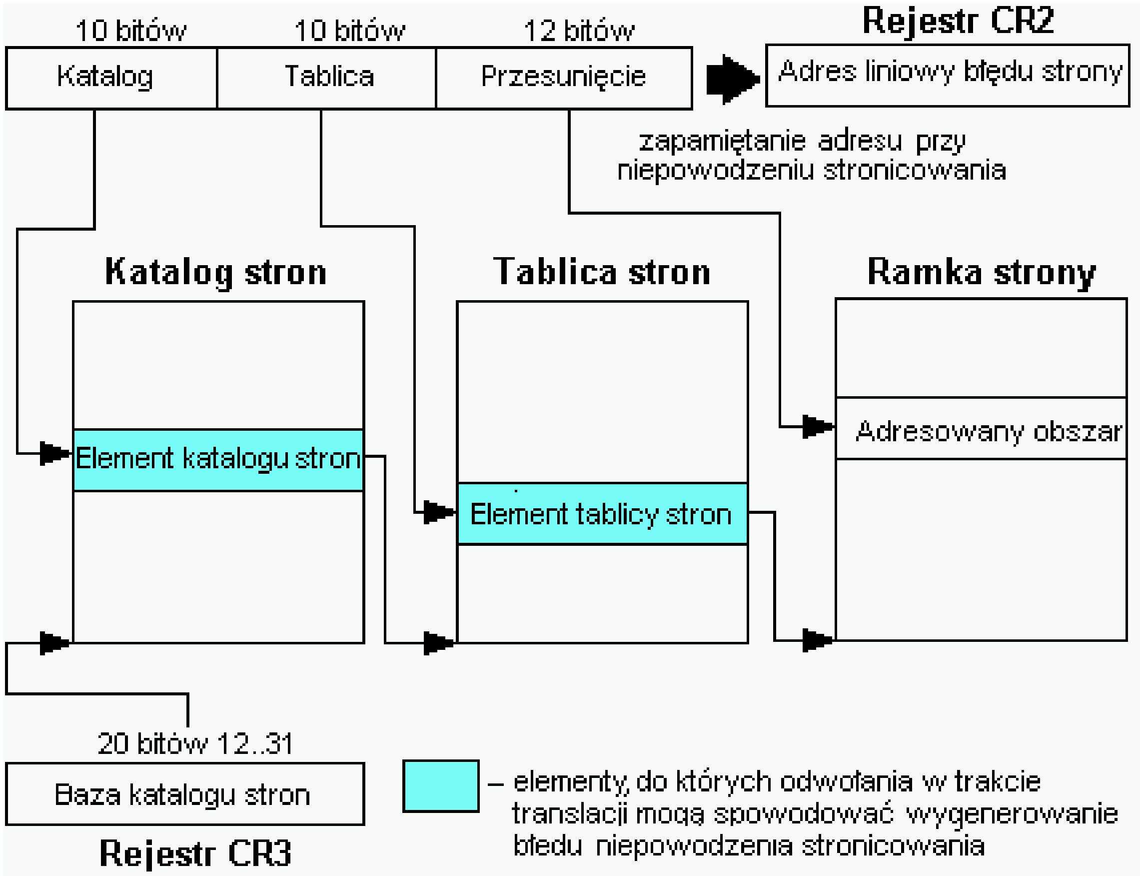
Schemat translacji adresów liniowych w fizyczne

Ponieważ elementy katalogu i tablic stron do opisu położenia strony w pamięci wykorzystują 20 bitów pozostaje 12 bitów na inne funkcje. Z tych 12-u tylko stan 3 bitów może spodować wygenerowanie wyjątku niepowodzenia stronicowania.

Znaczenie bitów w elementach katalogu i tablicy stron

Opis bitów (do rysunku Znaczenie bitów w elementach katalogu i tablicy stron):

- bit P równy 1 oznacza że strona znajduje się w pamieci
- bit P równy 0 oznacza brak strony w pamieci, generownay jest wyjątek przy próbie dostępu do strony (zarówno zapis jak i odczyt ze strony)
- bit W równy 1 na stronę można pisać jak i czytać z niej
- bit W równy 0 brak prawa do zapisu na stronie, ze strony można czytać ale próba zapisu spowoduje wygenerowanie wyjątku
- bit U równy 1 strona dostępna jest na wszystkich poziomach uprzywilejowania procesora



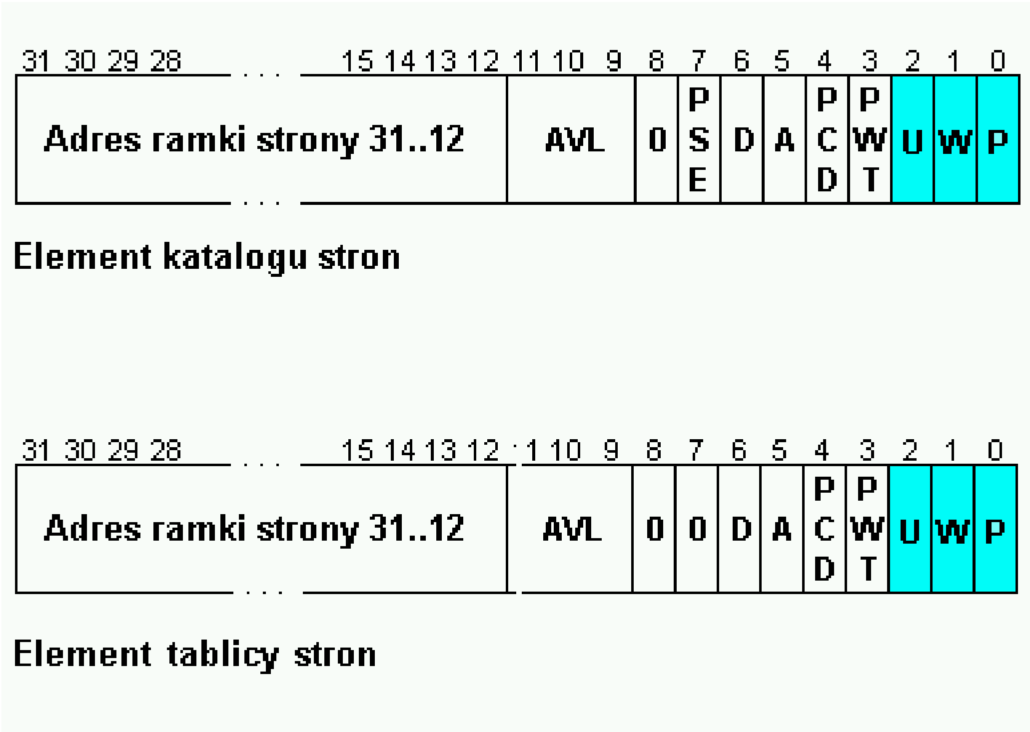
Rysunek 1.5: Schemat translacji adresów liniowych w fizyczne

- bit U równy 0 wymagany poziom uprzywilejowania procesora CPL < 3, wyjątek generowany jest jeśli translacja będzie się odbywać na poziomie uprzywilejowania procesora CPL $= 3$; CPL (ang. *Current Privilege Level*) oznacza aktualny poziom uprzywilejowania procesora. Określony przez bity 1..0 w rejestrze CS.

Bity W i U są ignorowane przez procesor 386 na poziomach uprzywilejowania CPL < 3, procesory 486 i Pentium uwzgledniają bit W na wszystkich poziomach uprzywilejowania.

Znaczenie pozostałych bitów:

- bity PCD, PWT umożliwiają sterowanie pamięcią cache
- bit A ustawiany przy dostępie do strony
- bit D ustawiany przy zapisie na stronę



Rysunek 1.6: Znaczenie bitów w elementach katalogu i tablicy stron

- bity AVL mogą być wykorzystane do przechowywania własnych informacji, może z nich korzystać na przykład system operacyjny
- bit PSE równy 1 umożliwia korzystanie z 4MB stron na procesorach Pentium

Adres liniowy, przy translacji którego powstał błąd jest zapisywany do rejestru CR2. Informacja ta może zostać użyta przez procedurę obsługi wyjątku. Na stosie odkładany jest kod błędu znaczenie mają tylko trzy najmłodsze bity :

- bit 0 równy 0 oznacza odwołanie do strony oznaczonej jako nieobecna
- bit 0 równy 1 oznacza naruszenie ochrony (odwołanie do strony z bitem U równym 0 na poziomie uprzywilejowania procesora CPL równym 3)
- bit 1 równy 0 oznacza próbę odczytu
- bit 1 równy 1 oznacza próbę zapisu
- bit 2 równy 0 oznacza że poziom uprzywilejowania, przy którym wygenerowany został wyjątek był wyższy niż 3 (CPL $<$ 3)
- bit 2 równy 1 oznacza że poziom uprzywilejowania, przy którym wygenerowany został wyjątek był równy 3 (CPL $=$ 3)

Bity te również mogą zostać użyte przez procedurę obsługi wyjątku.

1.3.2 Realizacja w Linuksie

Błędy braku i ochrony strony są to sytuacje wyjątkowe obsługiwane przez procesor w przypadku gdy proces (np. program użytkownika) próbuje się odwołać do obszaru pamięci, do którego z rozmaitych powodów nie ma praw dostępu. U źródeł powstania takich mechanizmów leżała chęć zabezpieczenia systemów operacyjnych przed wpływem źle napisanych programów. Linux także poprzez wywoływanie tego błędu umożliwia korzystanie z zaawansowanych metod zarządzania pamięcią wirtualną (np. stronicowanie na żądanie (ang. *paging on-demand*). Tak więc przetłumaczenie page_fault na błąd strony jest trochę niefortunne. Może się bowiem zdarzyć sytuacja, kiedy wystąpi błąd, a my tylko mamy daną stronę w pamięci wirtualnej.

Ogólny algorytm obsługi błędu i braku strony

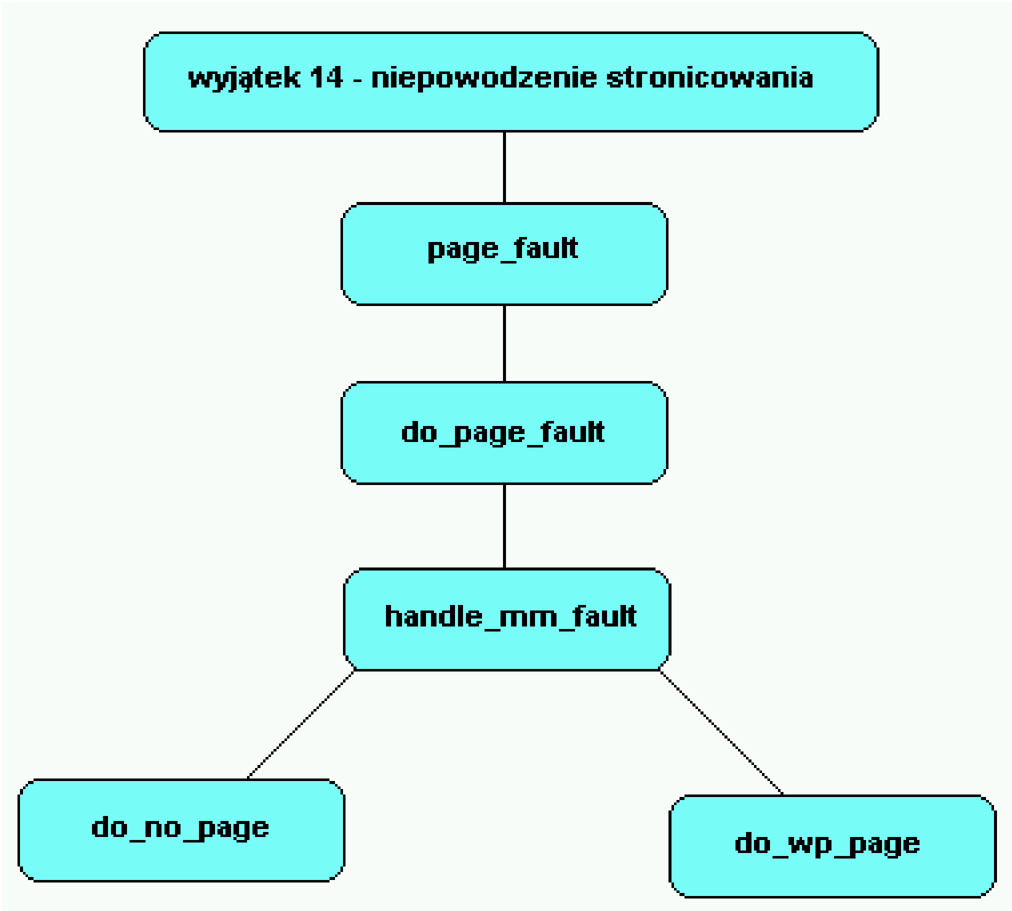
W przypadku wystąpienia błędu strony procesor wywołuje wyjątek 14. Oznacza to niepowodzenie stronicowania.
Linux już podczas startu podkłada pod ten wyjątek procedurę `page_fault` Jej kod znajduje się w `/usr/src/linux/arch/i386/kernel/entry.S`. Jest to bardzo prosta precedura napisana w asemblerze. Jej zadaniem jest wywołanie `do_page_fault` z odpowiednimi parametrami. Aby tak się stało, system odkłada na stos adres `do_page_fault` a później wywołuje podprocedurę `error_code`.Ta z kolei zwraca wskaźnik na wartości rejestrów i skacze pod adres ze stosu.
W ten sposób dochodzimy do wywołania procedury `asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)` Jest ona napisana już w C. Znajduje się ona w `/arch/i386/mm/fault.c`. Jej zadaniem jest sprawdzenie czy rzeczywiście wystąpił błąd w systemie.

- Jeśli nastąpił błąd braku strony spowodowany tym, że Linux zabrał prawa do pamięci procesowi lub przeniósł gdzieś daną stronę, to uznajemy, że wszystko jest w porządku i przekazujemy sterowanie dalej. Wywoływana jest procedura :
  `int handle_mm_fault(struct task_struct *tsk, struct vm_area_struct *vma, unsigned long address, int write_access)`. Jest to już procedura wysokopoziomowa i w tym miejscu nie bedziemy się nią już zajmować.

- Jeśli nastąpił inny błąd, to system nie potrafi go zwykle poprawnie obsłużyć. W tym przypadku, aby zachować stabilność systemu, kończymy proces, który zgłosił wyjątek. W przypadku sytuacji awaryjnych możemy nawet zatrzymać cały system. Dokładny algorytm przedstawiamy poniżej.

Procedura do_page_fault

`asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)`
Parametr `regs` wskazuje na kopię zawartości rejestrów procesora w momencie wystąpienia błędu, potrzebną do wznowienia działania procesu po obsłużeniu sytuacji wyjątkowej, a `error_code` niesie informacje o przyczynie jego wystąpienia.



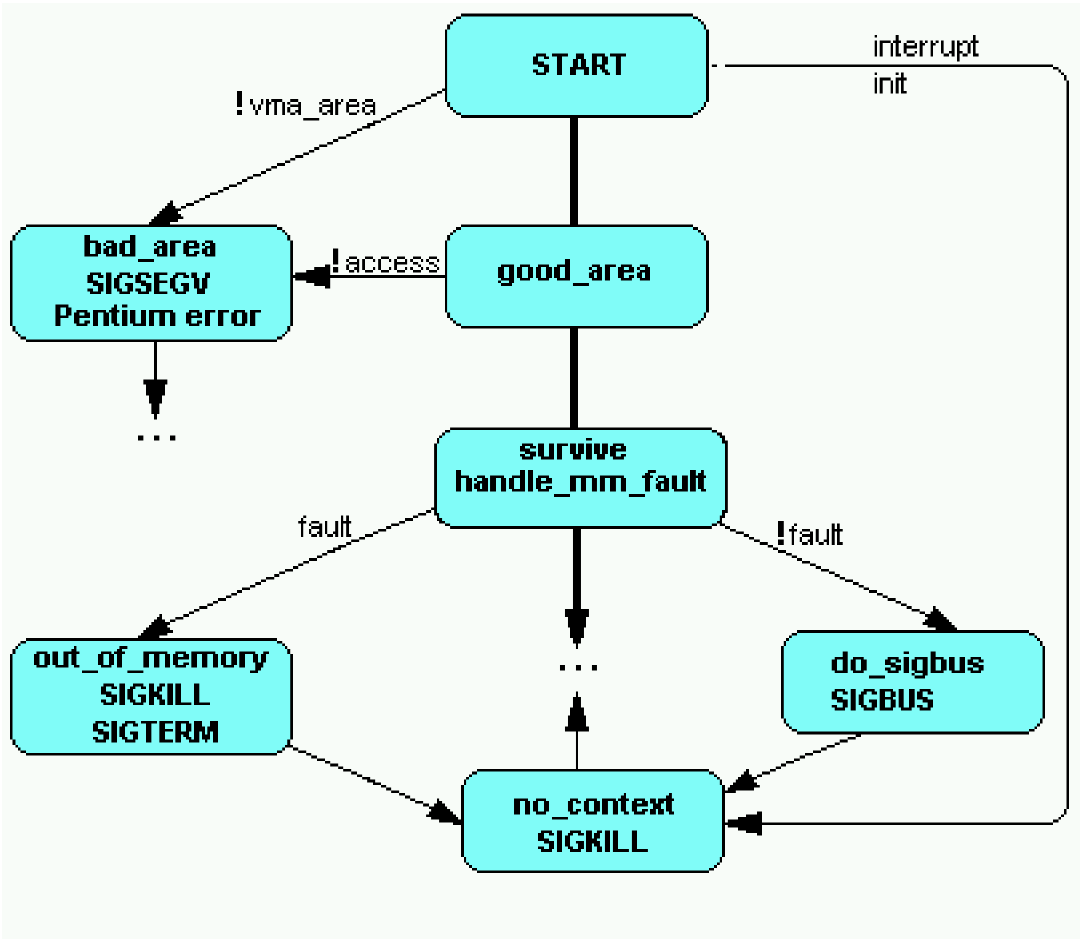
Rysunek 1.7: Algorytm obsługi błędu i braku strony

Aby obsłużyć błąd strony procedura `do_page_fault` wyznacza adres liniowy miejsca jego wystąpienia, a następnie na jego podstawie obszar pamięci, którego błąd dotyczy.

W przypadku, gdy poprawne zidentyfikowanie obszaru gdzie wystąpił wyjątek okaże się niemożliwe, aktualnie wykonywany proces zostaje przerwany, gdyż prawdopodobnie doszło do zniszczenia jego struktur danych i istniałoby niebezpieczeństwo załamania systemu. Odbywa się to poprzez wysłanie sygnału `SIGSEGV` do danego procesu.
Jeśli błąd wystąpił w jądrze, to niestety zatrzymujemy system. Nie jesteśmy bowiem w stanie zapewnić bezpieczeństwa systemu.

W innym przypadkach wywoływana jest procedura wysokopoziomowa `handle_mm_fault`. W przypadku, kiedy zwróci błąd, to niestety nie możemy przydzielić pamięci dla procesu. Jedynym wyjściem jest jego zakończenie poprzez wysłanie sygnału `SIGKILL`. W przypadku, kiedy system ma zaufanie do procesu (ustawiona odpowiednia flaga), to wysyłany jest sygnał SIGTERM.

Algorytm w tej postaci nie jest zbyt skomplikowany. Jednak musimy liczyć się jeszcze z architekturą i386. Czasami niektóre procesory Pentium generują błąd po napotkaniu instukcji `F0 0F C7 C8`. Tutaj obsługujemy taką sytuację, tj. wykonujemy tą instrukcję poprawnie. Szkic algorytmu przedstawiamy na rysunku:



Rysunek 1.8: Szkic algorytmu procedury do_page_fault

1.4 Przełączanie kontekstu

1.4.1 Wsparcie sprzętowe

Segment TSS

Najważniejszą strukturą danych podczas przełączania kontekstu jest segment **TSS** (ang. *Task State Segment*). Przechowuje on kontekst obecnie działającego procesu. Specjalny rejestr procesora **TR** (ang. *Task Register*) przechowuje selektor do aktywnego segmentu TSS. Na jego podstawie procesor dokonuje uaktualnienia danych w TSS. W TSS przechowywane są następujące dane obecnie działającego procesu:

- adresy stosów dla poziomów pracy procesora 0-2 (rejestry ESP0, SS0, ESP1, SS1, ESP2, SS2)
- rejestry ogólnego przeznaczenia (EAX, EBX, ECX, EDX)
- rejestry indeksowe (ESI, EDI)
- rejestry segmentowe (DS, ES, FS, GS)
- aktualny stos (SS, ESP, EBP)
- adres kolejnej instrukcji (CS, EIP)
- rejestr kontrolny katalogu stron (CR3)
- flagi (EFLAGS)
- mapę bitową dostępnych portów I/O
- selektor lokalnej tablicy deskryptorów
- opcjonalne dane systemu operacyjnego

Instrukcja `LTR`

Do załadowania nowego selektora do rejestru TR służy instrukcja `LTR`. Jej parametrem może być rejestr albo komórka pamięci.

Instrukcja `STR`

Dla symetrii oprócz instrukcji `LTR` istnieje instrukcja `STR`. Umożliwia ona odczytanie selektora z rejestru TR.

Deskryptory bram do TSS

Oprócz deskryptorów segmentów TSS, w tablicy deskryptorów można tworzyć deskryptory bramy do zadań (ang. *Task Gate Descriptor*). Każdy taki deskryptor zawiera selektor jednego deskryptora TSS.

Przełączanie zadań

W programie można wykonać daleki skok (zwykłe `far jmp`) pod selektor TSS albo selektor deskryptora bramy, co spowoduje automatyczne zapamiętanie bieżącego kontekstu, odtworzenie kontekstu z TSS selektora, do którego skaczemy, oraz uruchomienie zadania, które jest powiązane z tym TSS. Jest to odrobinę mylące, gdyż instrukcja `jmp` nie oznacza skoku pod nowy adres, ale wywołanie sprzętowej procedury przełączania zadań. Daje to gotowy mechanizm przełączania kontekstu. Czas takiego przełączania jest około 300 razy dłuższy od czasu zwykłego skoku.

Najprostszy sposób na przełączanie procesów

Na początek w GDT należy utworzyć segment TSS dla pierwszego procesu i załadować jego selektor instrukcją LTR. Dodawanie następnych zadań nie wymaga już wywoływania LTR. Wystarczy utworzyć nowy deskryptor TSS. Aby przełączyć zadanie należy wywołać `jmp far <selektor>`, gdzie `selektor` jest selektorem deskryptora TSS zadania, do którego zamierzamy się przełączyć.

1.4.2 Realizacja w Linuksie

W skład kontekstu procesu wchodzą: zawartość rejestrów procesora i przestrzeni adresowej procesu oraz struktury jądra związane z danym procesem. Jądro zachowuje kontekst gdy system otrzymuje przerwanie lub proces wywołuje funkcję systemową. Przełączanie kontekstu polega na zmianie aktualnie wykonywanego procesu. Przyczyną zmiany kontekstu może być zaśnięcie lub zakończenie się procesu, bądź wykorzystanie przydzielonego procesowi kwantu czasu.
W Linuksie wywołanie systemowe jest równoważne wygenerowniu odpowiedniego przerwania, natomiast przełączenie kontekstu może nastąpić tylko tuż przed powrotem z procedury obsługi przerwania. Jednak w trakcie wykonywania procedury obsługi przerwania może wystąpić przerwanie o wyższym priorytecie i kontekst procesu, który wykonuje się w trybie jądra, musi być zachowany. Dlatego każdy proces posiada własną przestrzeń na stosie jądra, w której odkładane są poszczególne warstwy kontekstu.
Warstwa kontekstu składa się z:

- Licznika programu określającego adres następnej instrukcji, ktorą wykona procesor (zawartość rejestru EIP).
- Rejestru flag, ktory określa stan procesora po wykonaniu ostatniej instrukcji (rejestr EFLAGS).
- Wskaźnika stosu, który zawiera adres wierzchołka stosu procesu w trybie jądra lub użytkownika - w zależności od trybu wykonania (rejestr ESP).
- Rejestrów ogólnego przeznaczenia oraz rejestrów segmentowych.

Ponadto w skład kontekstu procesu wchodzą struktury jądra związane z tym procesem:

- Odpowiednia pozycja w tablicy procesów.
- Pozycje w globalnej tablicy segmentów, lokalna tablica segmentów i tablice stron, które definiują przestrzeń adresową procesu.
- Stos procesu w segmencie jądra, który jest wykorzystywany gdy proces wykonuje się w trybie jądra.

Procesory i386 posiadają wsparcie dla przełączania kontekstu. Z każdym zadaniem związany jest specjalny segment (task state segment), który zawiera informacje o stanie danego procesu (rejestry, mapa portów). Wznowienie wykonywania procesu odbywa się przez skok do jego segmentu TSS. Jednak Linux nie wykorzystuje w pełni wsparcia sprzętowego. Wprawdzie korzysta z segmentów stanu zadania, lecz jądro samo zapamiętuje rejestry procesora (odkładając je na stos danego procesu w segmencie jądra) i wykonuje skok do kodu nowego procesu. Umożliwia to lepszą kontrolę nad segmentami procesu i zapewnia większe bezpieczeństwo systemu.

Struktury jądra związane z przełączaniem kontekstu

1. Pola struktury `task_struct` [`include/linux/sched.h`].
   - `struct thread_struct tss` Dokładnie odpowiada segmentowi TSS danego procesu, w którym zapamiętywane są wszystkie informacje potrzebne do odtworzenia stanu procesu.
   - `struct mm_struct mm` Zawiera informacje o pamięci przydzielonej procesowi oraz o jego segmentach w tablicy LDT. Procesy na ogół nie wykorzystują lokalnej tablicy deskryptorów segmentów.
2. Struktura `thread_struct` [`include/asm-i386/processor.h`].
   - rejestry SS i ESP poziomów: 0, 1, 2 Adresy stosu procesu w odpowiednich trybach uprzywilejowania. Gdy proces wykonuje sie w trybie jądra, to korzysta ze stosu poziomu 0.
   - rejestry segmentowe i ogólnego przeznaczenia
   - adres mapy portów we/wy
   - zawartość rejestru CR3 (fizyczny adres katalogu stron procesu) oraz LDT (adres lokalnej tablicy deskryptorów)
   - rejestry `debugregs` (wykorzystywane przy debugowaniu procesu)
3. Globalna tablica deskryptorów segmentów [`include/asm-i386/desc.h`]. `struct desc_struct {unsigned long a, b} gdt_table[]` (tablica 64-bitowych deskryptorów segmentów).

Etapy przełączania kontekstu

1. Wygenerowane zostało przerwanie lub proces wywołał funkcję systemową.
   - Następuje skok przez odpowiednią bramkę w IDT (tablica deskryptorów przerwań) do kodu jądra (DPL = 0), który obsługuje przerwania i wywołania systemowe.
   - W tablicy GDT znajdują się deskryptory następujących segmentów [`include/asm-i386-/desc.h`]:
   - Procesor ładuje SS poziomu 0 i ESP poziomu 0, które są zapamiętane w segmencie TSS danego procesu (jego selektor zawiera rejestr TR) i odkłada na nowy stos stare wartości SS i ESP (czyli adres wierzchołka stosu procesu). Następnie zapamiętywane są rejestry: EFLAGS, CS, EIP [`arch/i386/kernel/entry.S`].
   - Funkcja obsługująca przerwanie lub wywołanie systemowe odkłada na stos pozostałe rejestry (makro `SAVE_ALL`). Każdy proces posiada własny stos w segmencie danych jądra. Jest on wykorzystywany do zapamiętywania zawartości rejestrów i przekazywania wartości zwracanych przez funkcje systemowe. [`arch/i386/kernel/irq.h`] - `BUILD_COMMON_IRQ` (funkcja obsługi przerwań) [`arch/i386/kernel/entry.S`]
     - `ENTRY(system_call)` (obsługa wywołań systemowych i przerwań)
   - Jeśli obsługiwane jest przerwanie zegarowe, to system sprawdza, czy proces wyczerpał swój kwant czasu. Jeśli tak, to ustawiane jest pole `need_resched` w strukturze `task_struct` danego procesu.

Tablica 1.2: Zawartość tablicy GDT

...	...
2	segment kodu jądra
3	segment danych jądra
4	segment kodu procesu użytkownika
5	segment danych procesu użytkownika
...	...
12	TSS procesu 0
13	LDT procesu 0
14	TSS procesu 1
15	LDT procesu 1
...	...

Tablica 1.3: Zawartość stosu

warstwa kontekstu 0
warstwa kontekstu 1
warstwa kontekstu 2
...

- Po zakończeniu obsługi przerwania (wywołania systemowego) wykonywana jest procedura `ret_from_sys_call`, która przed oddaniem sterownia do procesu sprawdza wartość pola `need_resched`. Jeśli pole jest ustawione, to wywołuje funkcję `schedule()`.

2. Jeśli została wywołana funkcja `schedule()` i proces nie jest uprawniony do dalszego wykonywania się, to należy przełączyć kontekst.

   - Wywołanie makra `switch_to(prev, next, last)` [`include/asm-i386/system.h`].
     - Zapamiętanie ESI, EDI, EBP.
     - Zapisanie ESP w TSS procesu `prev` (wskaźnik na wierzchołek stosu procesu w segmencie jądra).
     - Odtworzenie ESP na podstawie TSS procesu `next` (stos nowego procesu).
     - Zapisanie adresu powrotu w TSS procesu `prev`.
     - Odłożenie tego samego adresu na stosie (aktualny stos to stos jądra dla procesu `next`; adres powrotu wskazuje na koniec funkcji).
   - Wywołanie funkcji `__switch_to(prev, next)` [`arch/i386/kernel/process.h`].
     - Zapisanie stanu FPU.
     - Wyzerowanie bitu B w tablicy GDT w deskryptorze segmentu TSS procesu `next` (oznacza, że zadanie nie jest aktualnie wykonywane).
     - Załadowanie rejestru TR dla nowego procesu.
     - Zapamiętanie rejestrów FS i GS w TSS starego procesu (rejestry te nie są używane przez jądro i wskazują na segmenty procesu).

  - Ładowana jest tablica LDT dla nowego procesu.
  - Ładowanie rejestru CR3 z TSS procesu `next` (CR3 zawiera fizyczny adres katalogu tablic stron procesu).
  - Ustawiane są nowe wartości rejestrów FS i GS.
  - Jeśli proces wykonuje się w trybie *debug*, to ładowane sa rejestry `debugregs`.
- Powrót do makra `switch_to`.
  - Ponieważ na stosie był adres instrukcji po `__switch_to` , to powrócimy w miejsce po wywołaniu tego makra. W tym momencie jądro działa już praktycznie w kontekście nowego procesu.
  - Ściągamy EBP, EDI i ESI, które nowy proces odłożył na swój stos w jądrze, gdy przestawał być aktywnym procesem.

3. Powrót z funkcji systemowej lub funkcji obsługi przerwania.
   - Odtworzenie rejestrów (makro `RESTORE_ALL`).
   - Powrót do nowego procesu. Procesor ładuje SS i ESP (stos procesu w jego segmencie danych), EFLAGS i skacze pod CS:EIP ze stosu jądra.

   [`arch/i386/kernel/entry.S`]

4. Uwagi.
   - Zwykle procesy nie używają swoich tablic LDT. Segmenty aktualnie wykonywanego procesu okreslają deskryptory nr 4 i 5 w GDT. Jednak deskryptory te nie zmieniają się nawet przy przełączaniu kontekstu. Ładowany jest jedynie rejestr CR3, ktory zmienia odwzorowanie adresów liniowych w ramach tego samego segmentu w adresy fizyczne.
   - Przełączanie kontekstu nie jest wykonywane automatycznie przez procesor, lecz krok po kroku przez system operacyjny. Dzięki temu można mieć kontrolę nad tym, czy segmenty procesu, do którego skaczemy, znajdują się w pamięci.

1.5 Mechanizm DMA

1.5.1 Wprowadzenie

DMA (ang. *Direct Memory Access*) służy do kopiowania danych z pamięci operacyjnej do urządzenia lub w stronę przeciwną bez angażowania czasu procesora. Obsługa DMA w jądrze Linuxa jest bardzo silnie zależna od wykorzystywanej architektury (zwłaszcza magistrali systemowej).

1.5.2 Przydzielanie bufora DMA

Bufor DMA musi znajdować się w ciągłym obszarze pamięci. Wynika to z tego, że zarówno magistrala ISA jak i PCI operują na fizycznych (a właściwie magistralowych) adresach pamięci. Do przeliczania adresów wirtualnych na adresy jakich używa magistrala i z powrotem służą funkcje (`asm/io.h`)

- `unsigned long virt_to_bus(volatile void * address)`

- `void * bus_to_virt(unsigned long address)`

Istnieją też magistrale w których stosowane są inne rozwiązania (np. magistrala Sbus pracuje na adresach wirtualnych). Aby przydzielić bufor DMA musimy się posłużyć funkcją `kmalloc` lub `get_free_pages`. Dla magistrali ISA trzeba przy alokacji bufora określić priorytet `GFP_KERNEL` aby zagwarantować ograniczenia przez nią nakładane.

1.5.3 Obsługa DMA dla magistrali ISA

Oryginalny kontroler DMA dla magistrali ISA mógł obsługiwać tylko cztery kanały. Aktualnie używany sprzęt zawiera układ równoważny dwóm takim kontrolerom. Posiada więc on 8 kanałów. Kanał 4 lub inaczej kanal 0 drugiego kontrolera nie jest jednak dostępny dla urządzeń ISA służy bowiem do połączenia z pierwszym kontrolerem. Kanały 0–3 są kanałami 8 bitowymi natomiast 5–7 są 16 bitowe. Sterownik DMA posiada kilkanaście rejestrów sterujących jego działaniem. Dla nas istotne jest, że każdemu kanałowi DMA przypisane są trzy rejestry: adresowy, liczący oraz strony. Rejestr adresowy jest 16 bitowy, natomiast stron 8 bitowy. Ponieważ pierwszy sterownik dokonuje transmisji 8 bitowej, a drugi 16 bitowej podczas jednej operacji mogą one przesłać odpowiednio 64 i 128 kilobajtów danych. Rozmiar rejestru stron nakłada na magistralę ISA ograniczenie transmisji do dolnych 16 megabajtów pamięci. Ograniczenie to jest sprawdzane podczas przydzielania pamięci na bufor DMA jeśli określimy priorytet `GFP_KERNEL`.

Struktury danych dla kanałów DMA

Źródła znajdują się w pliku `kernel/dma.c`

```
struct dma_chan {
        int  lock;
        const char *device_id;
};

struct dma_chan dma_chan_busy[MAX_DMA_CHANNELS]
```

Pole `lock` struktury `dma_chan` określa czy kanał jest wolny czy przydzielony jakiemuś urządzeniu (w tym drugim przypadku nazwa tego urządzenia jest wskazywana przez `device_id`). Nazwa urządzenia jest wykorzystywana wyłącznie przez system plików *proc*. Tablica `dma_chan_busy` zawiera informacje o wszystkich kanałach DMA (`MAX_DMA_CHANNELS` jest dla magistrali ISA zawsze rowne 8).

Przydzielanie i zwalnianie kanałów DMA

Zanim skorzystamy z kanału DMA jądro musi nam go przydzielić. Ponieważ o końcu transmisji DMA informuje nas zazwyczaj przerwanie je także musimy mieć przydzielone. Ustaliła się konwencja żeby najpierw żądać przerwania, a dopiero potem kanału DMA. Następujące funkcje służą do przydzielania i zwalniania kanałów:

`int get_dma_list(char *buf)` - wypisuje stan kanałów DMA i urządzenia z nich korzystające `int request_dma(unsigned int dmanr,const char * device_id)` - żądanie przydzielenia kanału DMA

`void free_dma(unsigned int dmanr)` - zwolnienie kanału DMA

Funkcje sterujące kontrolerem DMA

Deklaracje znajdują się w `asm/dma.h`

`void disable_dma(unsigned int dmanr)` - wyłącza kanał DMA. Funkcję tę powinno się stosować przed konfigurowaniem sterownika.

`void enable_dma(unsigned int dmanr)` - włącza kanał DMA

`void clear_dma_ff(unsigned int dmanr)` - kasuje przerzutnik bistabilny. Ponieważ sterownik DMA posiada 8 bitowe porty danych to dostęp do jego wewnętrznych rejestrów musi być rozbity na części. Przerzutnik bistabilny decyduje czy wartość jest interpretowana jako bajt mniej czy bardziej znaczący. Jeśli chcemy wywołać funkcje ustawiające rejestry sterownika, a nie jesteśmy pewni czy następna wartość będzie interpretowana jako mniej znaczący bajt powinniśmy wywołać wcześniej tę funkcję.

`set_dma_mode(unsigned int dmanr, char mode)` - ustawia tryb pracy kanału DMA. Trybami obsługiwanymi przez system Linux są:

- `DMA_MODE_READ` - pojedyncza transmisja z urządzenia do pamięci bez automatycznej inicjalizacji, adresy są zwiększane
- `DMA_MODE_WRITE` - pojedyncza transmisja z pamięci do urządzenia bez automatycznej inicjalizacji, adresy są zwiększane
- `DMA_MODE_CASCADE` - połączenie kaskadowe z innym sterownikiem
- `DMA_AUTOINIT` - automatyczna inicjalizacja

`set_dma_page(unsigned int dmanr, char pagenr)` - ustawia stronę dla której będzie wykonany transfer. Funkcja ta jest użyteczna jeśli chcemy przekroczyć ograniczenie 64 (lub 128) kilobajtów pamięci.

`set_dma_addr(unsigned int dmanr, unsigned int a)` - ustawia adres bufora DMA, dla drugiego sterownika trzeba podawać liczbę nieparzystą

`set_dma_count(unsigned int dmanr, unsigned int count)` - liczba bajtów (lub słów w przypadku kanałów 5–7) do przesłania, dla drugiego sterownika trzeba podawać liczbę nieparzystą

`get_dma_residue(unsigned int dmanr)` - zwraca 0 jeśli transmisja się zakończyła, w przeciwnym przypadku wartość jest nieokreślona

1.5.4 Obsługa DMA dla magistrali PCI

Implementacja DMA dla magistrali PCI jest dużo prostsza niż dla magistrali ISA. Urządzenie które chce czytać lub pisać do pamięci żąda kontroli magistrali, a gdy ją otrzyma samo kontroluje sygnały elektryczne nią sterujące. Aby przesłać dane do lub z pamięci wystarczy więc przydzielić bufor DMA oraz zapisać informacje sterujące transmisją do rejestrów urządzenia. W przypadku magistrali PCI nie obowiązuje ograniczenie 16 dolnych magabajtów. Bufor DMA może się znajdować w dowolnym miejscu pamięci głównej.

1.6 Zegary w Linuksie

1.6.1 Liczenie czasu

W Linuksie czas jest liczony od uruchomienia systemu. Jednostka czasu jest zdefiniowana za pomocą makrodefinicji `HZ (linux/param.h)`. Dla systemów opartych na procesorze Alpha jest to 256 lub 1024, dla pozostałych platform 100 czyli 10 milisekund. Oznacza to, że co 10 milisekund generowane jest przez układ zegara czasu rzeczywistego tyknięcie, które następnie liczone jest przez przerwanie zegarowe. Aktualna liczba wygenerowanych tyknięć znajduje się w globalnej zmiennej `jiffies (linux/sched.h)`. Jeśli zainteresowani jesteśmy czasem rzeczywistym to znajduje się on w zmiennej `xtime`, jednak dokładność zapewniana przez tę zmienną wynosi także tylko 10 milisekund. Aby odczytać aktualny czas z większą dokładnością musimy skorzystać z zegara czasu rzeczywistego (robi to np. funkcja `void do_gettimeofday(struct timeval *tv)` `(linux/time.h)`).

1.6.2 Przerwania zegarowe i ich obsługa

Ponieważ przerwania zegarowe wykonywane są dość często to czas ich wykonania powinien być jak najkrótszy. Z tego powodu procedura ich obsługi podzielona jest na połowę górną i dolną. Oto jak wygląda górna połowa `(linux/sched.h)`:

```
void do_timer(struct pt_regs * regs)
{
        (*(unsigned long *)&jiffies)++;
        lost_ticks++;
        mark_bh(TIMER_BH);
        if (!user_mode(regs))
                lost_ticks_system++;
        if (tq_timer)
                mark_bh(TQUEUE_BH);
}
```

Jak widzimy zwiększa ona wartości zmiennych `jiffies` oraz `lost_ticks` i `lost_ticks_system`. Zmienne te liczą liczbę przerwań jakie upłynęły od czasu ostatniego wywołania dolnej części procedury obsługi przerwania. Procedura `do_timer` zaznacza też jako aktywną swoją dolna połowę oraz kolejkę zadań `TQUEUE_BH`. Zadania wymagające większej ilości czasu wykonywane są przez połowę dolną obsługi przerwania zegarowego:

```
static void timer_bh(void)
{
        update_times();
        run_old_timers();
        run_timer_list();
}
```

Funkcja `update_timers` odpowiada za aktualizację wartości zegarów, zaś `run_old_timers` oraz `run_timer_list` zajmują się uruchamianiem funkcji znajdujących się na listach starych i nowych liczników czasu (patrz następny podrozdział).

W systemie Linux można ograniczyć zużycie procesora przez proces. Robi się to za pomocą funkcji systemowej `setrlimit`. Przekroczenie granicy sprawdzane jest w funkcji `update_times`, a proces jest informowany przez sygnał `SIGXCPU`, albo przerywany przez `SIGKILL`.

```
psecs = (current -> stime + current -> utime) / HZ;
if (psecs > current -> rlim[ RLIMIT_CPU ].rlim_cur)
/* Wysyłaj SIGXCPU co sekundę */
   if (psecs * HZ == current -> stime +  current -> utime)
      send_sig( SIGXCPU , current ,1);
/* A SIGKILL, gdy zostanie przekroczone maksimum */
   if (psecs > current -> rlim [ RLIMIT_CPU ].rlim_max)
      send_sig( SIGKILL ,current ,1);
```

1.6.3 Planowanie zadań

W Linuksie istnieje możliwość odłożenia wykonania jakiegoś zadania na później. Najważniejsze mechanizmy wykorzystywane w tym celu to kolejki zadań i liczniki czasu. O ile kolejki zadań umożliwiają nam uruchamianie zadań w bliżej nieokreslonej przyszłości to liczniki czasu pozwalają na podanie dokładnego czasu wykonania.

Kolejki zadań

Kolejki zadań są mocno związane z mechanizmem dolnych połówek procedur obsługi przerwań. Struktury i funkcje związane z kolejkami zadeklarowane są w pliku `linux/tqueue`.

```
struct tq_struct {
        struct tq_struct *next;
        unsigned long sync;
        void (*routine)(void *);
        void *data;
};
```

`void queue_task(struct tq_struct *bh_pointer, task_queue *bh_list)` - umieszczenie zadania w kolejce

`void run_task_queue(task_queue *list)` - wykonanie kolejki zadań

Predefiniowane kolejki zadań

`tq_timer` - kolejka uruchamiana przy obsłudze przerwania zegarowego (zadania w niej są wykonywane w trakcie przerwania)

`tq_immediate` - kolejka zadań pilnych. Zadania w niej wykonywane są w momencie powrotu z wywołania systemowego lub przy uruchamianiu programu szeregującego. Kolejka jest uruchamiana w trakcie przerwania.

`tq_scheduler` - kolejka zadań wykonywanych podczas działania programu szeregującego (zadania nie są wykonywane w trakcie przerwania)

`tq_disk` - wykorzystywana do wewnętrznego zarządzania pamięcią

Stare liczniki czasu

Stare liczniki czasu zostały w jądrze linuxa w celu zachowania kompatybilności. Odradza się ich stosowanie w nowych programach. Starych liczników czasu jest 32 z czego część ma już zdefiniowane znaczenie:

```
#define BLANK_TIMER     0       /* Screen-saver konsoli */
#define BEEP_TIMER      1       /* Głośnik konsoli */
#define RS_TIMER        2       /* Port RS-232 */
#define SWAP_TIMER      3       /* Wymiana stron */
#define BACKGR_TIMER    4       /* Żądanie wejścia/wyjścia */
#define HD_TIMER        5       /* Stary kontroler IDE */
#define FLOPPY_TIMER    6       /* Stacja dysków */
#define QIC02_TAPE_TIMER 7      /* Taśma QIC 02 */
#define MCD_TIMER       8       /* CDROM Mitsumi */
#define GSCD_TIMER      9       /* CDROM Goldstar */
#define COMTROL_TIMER   10      /* Comtrol serial */
#define DIGI_TIMER      11      /* Digi serial */
#define GDTH_TIMER      12      /* Kontroler Gdth scsi */

#define COPRO_TIMER     31      /* 387 błąd sprzętowy (w czasie bootowania) */
```

Wszystkie struktury i funkcje związane z licznikami czasu zadeklarowane są w pliku `linux/timer.h`. Liczniki składają się z funkcji oraz czasu w jakim ma ona zostać wykonana. Funkcja jest wykonywana tylko raz, jeśli chcesz ją uruchomić po raz drugi musisz ją ponownie zarejestrować w liczniku czasu.

```
struct timer_struct {
        unsigned long expires;
        void (*fn)(void);
};

extern struct timer_struct timer_table[32];
```

Nowe liczniki czasu

Nowe liczniki czasu nie nakładają ograniczenia na ilość zarejestrowanych funkcji są bowiem zaimplementowane jako lista dwukierunkowa. Dają też możliwość przekazywania argumentów do uruchamianych funkcji (pole `data`). Zaleca się korzystanie z funkcji jądra w celu modyfikowania wartości pól struktury `timer_list` po wstawieniu jej do listy.

```
struct timer_list {
        struct timer_list *next;
        struct timer_list *prev;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

`void init_timer(struct timer_list * timer)` - inicjowanie struktury liczników czasu

`void add_timer(struct timer_list * timer)` - wstawienie licznika do listy liczników aktywnych. W wersjach jądra do 1.2 pole `timer->expires` traktowano jako wartość bezwzględną, obecnie zaś jako przesunięcie w czasie względem aktualnej wartości zmiennej `jiffies`

`int del_timer(struct timer_list * timer)` - usunięcie licznika z listy liczników aktywnych (jeśli czas wygaśnięcia upłynął wcześniej to licznik zostanie usunięty automatycznie)

`int timer_pending(struct timer_list * timer)` - sprawdzenie czy na liście znajduje się jakiś licznik

`void mod_timer(struct timer_list * timer, unsigned long expires)` - modyfikuje czas wygaśnięcia licznika

1.7 Jak zaimplementować własną funkcję systemową

Nowe wywołanie systemowe można dołączyć do jądra na dwa sposoby. Pierwsza, tradycyjna metoda polega na wstawieniu go do jądra na stałe. Istnieje także inna możliwość. Jest nią stworzenie własnego modułu. Zajmiemy się tutaj tylko tą pierwszą ewentualnością. Przypomnijmy, jaki jest mechanizm wywołania funkcji systemowej.

Mechanizm wywoływania funkcji systemowych

Podczas wywołania funkcji systemowej przechodzimy z trybu użytkownika do trybu jądra. W Linuksie dla procesorów x86 jest to realizowane za pomocą przerwań (przerwanie **0x80**).

Zazwyczaj w celu wywołania funkcji systemowej wywołujemy funkcję biblioteczną. Jej działanie polega na wypełnieniu odpowiednich rejestrów i wywołaniu przerwania 0x80. Do tego celu zdefiniowane są specjalne makra, które zwalniają nas z obowiązku bezpośredniego pisania w asemblerze.

Kod funkcji systemowej

Pisząc kod funkcji systemowej możemy odwoływać się do struktur, zmiennych i funkcji jądra, dołączając jedynie odpowiednie pliki nagłówkowe. Gotową funkcję, zgodnie z konwencją, umieszczamy w katalogu `/usr/src/linux/kernel`[12].

Nazwa funkcji powinna mieć postać `sys_nazwa_funkcji`.

Przykład

Przykładowo możemy chcieć stworzyć własną funkcję `getpid`. W pliku `kernel/my_getpid.c` umieszczamy jej treść:

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

int sys_my_getpid(void)
{
 return current->pid;
}
```

To jednak jeszcze nie wszystko. Każda funkcja systemowa musi być zarejestrowana.

Zarejestrowanie nowego wywołania systemowego

W pliku `linux/unistd.h` (lub `asm/unistd.h`) zdefiniowane są numery wszystkich funkcji systemowych. Aby dodać nową funkcję, należy dopisać do jednego z tych plików linijkę:

```
#define __NR_nazwa_funkcji    numer
```

Na przykład, dla naszej funkcji `my_getpid` mogłoby to wyglądać tak:

```
#define __NR_my_getpid        191
```

(gdzie 191 jest najbliższym wolnym numerem)

Plik `arch/386/kernel/entry.S` zawiera zainicjowaną tablicę wywołań systemowych (`_sys_call_table`). Tam też dodajemy naszą funkcję:

```
.long SYMBOL_NAME(sys_my_getpid)
```

Należy również zmodyfikować liczbę funkcji:

```
.rept NR_syscalls-191
```

Kompilacja

W pliku `Makefile` w katalogu `kernel` do listy `O_OBJS` dodajemy `my_getpid.o`. Tak zmodyfikowane jądro możemy skompilować (zobacz rozdział 1.1.2.), a następnie instalujemy je w tradycyjny sposób (należy pamiętać o wywołaniu `/sbin/lilo` przed zrestartowaniem systemu).

[12] Dalej ścieżki dla plików `.c` podawane są względem katalogu `/usr/src/linux`, a dla plików `.h` względem `/usr/include`.

Co dalej

Naszej nowej funkcji systemowej możemy używać w taki sposób:

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

_syscall0(int,my_getpid)

void main(void)
{
    printf("%d\n",my_getpid());
}
```

Makro `_syscall0(int,nazwa_funkcji)` generuje wywołanie przerwania `0x80` z odpowiednimi parametrami w rejestrach dla funkcji bezparametrowej. Istnieją też makra `_syscall1`, `_syscall2`,. . . , `_syscall5` dla funkcji systemowych mających więcej parametrów.

Makro `_syscall` dzięki wpisowi do `linux/unistd.h` zna numer naszej funkcji i umieszcza go w rejestrze wywołując przerwanie `0x80`. Procedura obsługi przerwania szuka w tablicy `sys_call_table` adresu funkcji pod pozycją przekazaną w rejestrze. Znajduje ten adres i wykonuje tam skok. Funkcje systemowe działają w trybie jądra i dzięki temu mają dostęp do wszystkich jego struktur i zmiennych. Jednocześnie nie jest możliwe korzystanie z funkcji biblioteki C – na szczęście większość ich odpowiedników jest wkompilowana w jądro.

1.8 Zadania

1. Załóżmy, że zależy nam na tym, aby obraz jądra na dysku był jak najmniejszy, przy zadanej funkcjonalności. Jakie kroki należałoby podjąć, aby do kompresji jądra można było użyć innego, skuteczniejszego algorytmu kompresji? Czy można by kompresować większą część jądra? [13]

2. W hipotetycznym systemie podczas kompilacji jądra okazało się, że odpowiednie pliki mają w przybliżeniu następujące rozmiary:

   - `arch/i386/boot/bootsect` - 0.5 KB,
   - `arch/i386/boot/setup` - 3.4 KB,
   - `arch/i386/boot/compressed/head.o` - 0.8 KB,
   - `arch/i386/boot/compressed/misc.o` - 14.4 KB,
   - `arch/i386/boot/kernel/head.o` - 4.9 KB,
   - pozostałe pliki razem - 695.1 KB.

   Przyjmując, że podczas kompresji rozmiar kompresowanych plików zmniejszył się do 60 procent początkowej wartości, oblicz jaki będzie rozmiar utworzonego jądra. Ile pamięci musi mieć komputer, żeby załadować to jądro? (Co nie znaczy, że potem przy tej ilości pamięci system będzie działać.) [14]

[13] Autor: Paweł Pałucha
[14] Autor: Paweł Pałucha

3. Obecnie w adresie liniowym 10 bitów używanych jest do zaadresowania elementu katalogu tablicy stron, 10 do zaadresowania elementu tablicy stron a 12 do zaadresowania offsetu w stronie. Jaki byłby rozmiar pamięci fizycznej, do której mógłby odwołać się proces, gdyby liczby te wynosiły odpowiednio 9, 9 i 14? [15]

4. Przyjmijmy, że w tablicy partycji dysku `/dev/hda` znajdują się 3 wpisy o partycjach podstawowych i jeden o partycji rozszerzonej. Ile urządzeń (`/dev/hda1`, `/dev/hda2` itp.) będzie dostępnych dla systemu? [16]

5. Dodać możliwość podawania w lini poleceń jądra rozmiaru pamięci w gigabajtach (2^{30} bajtów). Wskazówka: przyjrzyj się się plikowi `setup.c` w katalogu `/arch/i386/kernel/`. [17]

6. Zrobić dyskietkę startową z jądrem dla swojego komputera. [18]

7. Policz ile czasu może pracować Linux do momenu przepełnienia się licznika `jiffies`? [19]

8. Policz wartość zmiennej `jiffies` jeśli system został włączony dokładnie 13 lutego 1978 o godzinie 11:07pm z dokładnością do jednej sekundy? [20]

9. Jaka różnica wartości zmiennej `jiffies` odpowiada tygodniowi czasu rzeczywistego? [21]

10. O ile zwiększy się plik `/boot/map` po dodaniu w pliku `/etc/lilo.conf` opisu nowego obrazu ładowalnego dla systemu MS-DOS? [22]

11. Mamy urządzenie, które nie potrafi generować przerwań, o prędkości transmisji 900 KB/s. Jak często powinna być wywoływana funkcja sprawdzająca zakończenie transmisji DMA, jeśli wielkości bufora DMA wynoszą odpowiednio 50, 90 i 170 KB? [23]

12. Na liście liczników czasu znajdują się zadania, których momenty uruchomienia wynoszą odpowiednio 2,5,6,9 (zakładając, że wartość zmiennej `jiffies` wynosi 0). Przyjmijmy, że wielkość opóźnienia przy uruchomieniu dolnych części procedur wynosi maksymalnie 4 takty zegara. Jakie będzie łączne opóźnienie czasów uruchomienia zadań w najgorszym przypadku? [24]

1.9 Rozwiązania zadań

1. Nie można kompresować większej części obrazu jądra, chyba żeby procedura dekompresująca była napisana w asemblerze i działała w trybie rzeczywistym. Do działania funkcji dekompresującej w C potrzebne jest załadowanie systemu (`bootsect`), przełączenie w tryb chroniony (`setup`) i przygotowanie środowiska do wykonywania kodu w C (`head.o`) – a więc te pliki nie mogą być skompresowane. Do zaimplementowania innego algorytmu kompresującego należałoby:

[15] Autor: Wojciech Moczydłowski
[16] Autor: Wojciech Moczydłowski
[17] Autor: Robert Osowiecki
[18] Autor: Robert Osowiecki
[19] Autor: Michał Bernat
[20] Autor: Michał Bernat
[21] Autor: Michał Bernat
[22] Autor: Krzysztof Gwiazda
[23] Autor: Sławomir Leszczyński
[24] Autor: Sławomir Leszczyński

- zmienić `Makefile`, żeby kompresję wykonać innym programem,
- podmienić plik `lib/inflate.c` na zawierający inne funkcje dekompresujące; aby używać w nim jakiś dodatkowych funcji w C, należałyby je zaimplementować np. w pliku `arch-/i386/boot/compressed/misc.c`.

2. Kompresji podlega plik `arch/i386/boot/kernel/head.o` i „pozostałe". Stąd też rozmiar powstałego jądra będzie wynosił mniej więcej: 0.6 * (4.9 KB + 695.1 KB) + 0.5 KB + 3.4 KB + 0.8 KB +14.4 KB = 439.1 KB. Jest to mniej niż 448 KB, czyli można jądro skompilować jako „małe". W związku z tym przed dekompresją jest używana tylko pamięć poniżej 1 MB. Przy dekompresji jądro o rozmiarze 700 KB (`head.o` + „pozostałe") umieszczane jest od 1 MB. Stąd też wielkość używanej pamięci to około 1.7 MB, czyli komputer z 2 MB powinien wystarczyć. Oczywiście do działania systemu taka wielkość pamięci raczej nie wystarczy...

3. Oczywiście dokładnie taki sam - 4 gigabajty, bo rozmiar adresu pozostaje niezmieniony - 32 bity.

4. Nie wiadomo - na partycji rozszerzonej mogą znajdować się różne ilości partycji logicznych.

5. W procedurze setup_arch zjajudje się kod pobierający rozmiar pamięci z parametru `mem=xxx?`, gdzie xxx – liczba naturalna, **?** – puste dla rozmiaru w bajtach, **k** lub **K** dla rozmiaru w kilobajtach i analogicznie dla megabajtów. Dodajemy na wzór tych instrukcji warunek dla znaku **G**:

```
memory_end = simple_strtoul(from+4, &from, 0);
if ( *from == 'K' || *from == 'k' ) {
    memory_end = memory_end << 10;
    from++;
} else if ( *from == 'M' || *from == 'm' ) {
    memory_end = memory_end << 20;
    from++;
/* dopisany kod */
} else if ( *from == 'G' ) {
    printk("%luGB of memory! WOW!\n",memory_end);
    memory_end = memory_end << 30;
    from++;
}
```

Jeśli zechcesz przetestować swoją poprawkę, to po skompilowaniu jądra i umieszczeniu dla niego wpisu w `/etc/lilo.conf` dopisz do paramerów `append = "mem=1G"` . [25]

6. Po kompilacji jądra upewniamy się, czy make ustawił właściwe urządzenie jako główny system plików (`rdev obraz_jądra`) i zmieniamy go w razie potrzeby (`rdev obraz_jądra urządzenie_blokowe`). Następnie kopiujemy obraz jądra na sformatowaną dyskietkę poleceniem:
`dd if=obraz_jądra of=/dev/fd?  bs=512`

[25] Jeśli uda ci się wtedy uruchomić system, to pewnie wszyscy dokoła ci zadroszczą (ale tylko teraz: za parę lat każdy będzie tyle miał).

Najczęściej spotykane jest `/dev/fd0`. Jeśli chcemy zmieniać parametry już nagranego jądra, podajemy jako obraz jądra urządzenie (`np. /dev/fd0`). Utworzoną dyskietkę zachować na pamiątkę: może się kiedyś przyda...

7. Brak

8. Brak

9. Brak

10. Ponieważ wpis dotyczy obrazu dla obcego systemu operacyjnego, więc obraz będzie się składał z trzech sektorów: dwóch dla kopii pliku `/boot/chain.b` i tablicy partycji oraz jednego, do którego skopiowany zostanie odpowiedni sektor ładujący. Te trzy sektory zostaną umieszczone w pliku `/boot/map`. Dodatkowo plik ten zwiększy się o sektor adresowy dla tego obrazu oraz sektor bezpiecznej linii poleceń i sektor domyślnej linii parametrów. Razem dostajemy dodatkowe sześć sektorów.

11. Odpowiedź: 18, 10 i 6 razy na sekundę.

12. Odpowiedź: 10 taktów. Obsługa dolnych części procedur odbywać się powinna w taktach 4, 8 i 13.

Literatura

[Rubini 99] Rubini A.: *Linux sterowniki urządzeń.* Warszawa, RM 1999.

[Beck 99] Beck M. i inni: *Linux Kernel. Jądro systemu.* Warszawa, Mikon 1999.

[Almesberger 98] Almesberger W.: *LILO, Generic boot loader for Linux, User's guide.* December 4, 1998.