Artur Koniński Mateusz Kocielski
Paweł Wieczorek

System operacyjny Impala

Licencjacki projekt programistyczny
Uniwersytet Wrocławski

Wrocław, 16 września 2014

Spis treści

1 Opis projektu. **3**
1.1 Wstęp. 3
1.1.1 Cele i motywacje projektu. 3
1.1.2 Co udało się osiągnąć. 3
1.2 Organizacja. 4
1.2.1 Użyte narzędzia. 4
1.2.2 Opis drzewa z kodem źródłowym. 4

2 Dokumentacja użytkownika. **6**
2.1 Uruchomienie systemu. 6
2.1.1 Parametry jądra 6
2.2 Powłoka systemowa. 7
2.3 Przestrzeń użytkownika. 7
2.3.1 Drzewo katalogów 7
2.3.2 Polecenia standardowe. 8
2.3.3 Polecenia specjalne. 8
2.3.4 Programy demonstracyjne. 9

3 Dokumentacja techniczna. **11**
3.1 Zarządzanie pamięcią. 11
3.1.1 Mapa pamięci. 12
3.1.2 Odwzorowania stron. 12
3.1.3 Wirtualne segmenty i przestrzenie adresowe. 14
3.1.4 Sterta jądra. 15
3.2 Sterowniki urządzeń. 15
3.2.1 Szyna ISA. 17
3.2.2 Sterownik stacji dyskietek. 17
3.2.3 Sztuczne urządzenia. 18
3.3 Przerwania. 18
3.4 Strumieniowe wejście-wyjście. 20
3.5 Buforowane wejście-wyjście. 20
3.5.1 Tablica haszująca. 21
3.5.2 Cykl życia nagłówka buforu. 22
3.6 Interfejs terminali. 22
3.6.1 Reżim linii. 23
3.6.2 Konsola. 25

3.6.3 Emulacja terminala. 26
3.6.4 Termcap. 26
3.7 Wirtualny system plików (VFS). 26
3.7.1 Pliki od strony użytkownika. 26
3.7.2 Pliki od strony jądra. 27
3.7.3 System plików. 28
3.8 Wielozadaniowość. 30
3.8.1 Zmienianie kontekstu procesora. 31
3.8.2 Rozwidlanie procesów. 32
3.8.3 Synchronizacja. 32
3.8.4 Biblioteka wątków użytkownika. 33
3.8.5 Szeregowanie zadań. 34
3.9 Pomniejsze usługi jądra. 34
3.9.1 Wywołania systemowe. 34
3.9.2 Sygnały. 35
3.9.3 Kolejki wiadomości. 35
3.10 Szczegóły techniczne. 36
3.10.1 Obraz jądra. 36
3.10.2 Obrazy programów. 36
3.10.3 Rozruch systemu. 36
3.10.4 System budowania. 37

A Licencja. **39**

Rozdział 1

Opis projektu.

1.1 Wstęp.

System operacyjny Impala został zrealizowany w ramach licencjackiego projektu programistycznego na Uniwersytecie Wrocławskim. W skład systemu wchodzi jądro, biblioteka systemowa (języka C), biblioteka wątków oraz oprogramowanie. System jest wzorowany na systemach wywodzących się z systemu UNIX i przeznaczony jest dla komputerów klasy PC.

1.1.1 Cele i motywacje projektu.

Głównym celem i motywacją do powstania projektu była chęć stworzenia systemu operacyjnego, który stanowiłby pomoc w nauce ich budowy i zasad działania oraz mechanizmów niskopoziomowych komputera. Projekt miał być wzorowany na systemach wywodzących się z systemu UNIX i implementować możliwie najwięcej zgodnie ze standardem POSIX. Jako wyznacznik dojrzałości projektu oraz funkcjonalności kodu przyjęte zostało przeniesienie powłoki systemowej `ash` oraz edytora tekstu vi. Projektowany system pomimo, że tworzony wyłącznie na komputerach klasy PC, wyposażonych w procesor Pentium Pro, miał być możliwie najbardziej przenośny. Dodatkową motywacją była chęć zgłębienia kodu i mechanizmów działania popularnych następców systemu UNIX.

Powłoka `ash` została zaprogramowana w 1992 roku przez Kennetha Almquista dla systemu BSD UNIX[1] jako zastępca oryginalnej powłoki Bourna. Nie posiada wielu wygodnych funkcji, jak historia poleceń i dopełnianie komend, dostępnych w popularnych powłokach jak `bash`, `tcsh`, `zsh`, lecz posiada obsługę zarządzania zadaniami, stąd można ją uznać za wyznacznik podobieństwa do systemu UNIX.

1.1.2 Co udało się osiągnąć.

Zrealizowany projekt z pewnością stwarza możliwość zgłębienia tajników budowy i zasad działania systemu operacyjnego, wszystkie ważne mechanizmy zostały zaprojektowane, zaimplementowane i omówione w tej dokumentacji. Założenie o przenośności systemu zostało zrealizowane poprzez oddzielenie kodu działającego na poziomie sprzętu i operującego bezpośrednio na nim od kodu działającego niezależnie od platformy. Podejście to umożliwia

[1] Powłoka do dziś jest stosowana w systemie FreeBSD jako główna powłoka systemowa. Niektóre dystrybucje systemu Linux używają tej powłoki jako wyznacznik zgodności skryptów ze standardem, w dystrybuowanych paczkach jest dostępna pod nazwą `dash`.

studiowanie mechanizmów niskopoziomowych niezależnie od reszty systemu. Udało się zrealizować założenie podobieństwa do systemu UNIX, czego konsekwencją jest przeniesienie z powodzeniem powłoki systemowej `ash`, nie został jednak zrealizowany cel przeniesienia edytora tekstowego, który wymaga dalszego dostosowania. Udało się również przenieść program `vttest`, który posłużył przy sprawdzaniu i udoskanalaniu naszej implementacji terminali w emulacji VT100.

Ponieważ powłoka systemowa została zaprojektowana dla systemu BSD to podczas rozwijania naszego systemu by był w stanie uruchomić ten program mogliśmy odbiec od semantyki procedur systemowej ze standardu POSIX, na rzecz systemu docelowego powłoki. Stwierdzenie zgodności ze standardem wymaga przeprowadzenia testów.

1.2 Organizacja.

1.2.1 Użyte narzędzia.

Realizacja projektu wymagała użycia wielu narzędzi dostarczonych z zewnątrz. Poniżej znajduje się zwięzły opis tych, które odegrały kluczową rolę podczas pracy.

Zarządzanie pracą grupową i podział zadań na poszczególnych programistów było wspomagane przez system EdgeWall Trac[2], kolejne etapy projektu zostały podzielone na mniejsze zadania i wprowadzone jako bilety, dzięki wykorzystaniu tego mechanizmu możliwa była na bieżąco kontrola efektów pracy i planowanie kolejnych etapów rozwoju. Wykorzystane zostały również mechanizmy zarządzania treścią do budowania zalążków dokumentacji oraz wymiany najistotniejszych informacji o projekcie. Interfejs dodatkowo stanowił graficzną nakładkę na system kontroli wersji pozwalając wygodnie przeglądać kolejne rewizje kodu, udostępniony mechanizm wiki umożliwił również robienie krótkich notek na stronie dotyczących projektu.

System kontroli wersji SVN został wykorzystany do synchronizacji kodu, umożliwił on niezależną i wygodną pracę nad projektem każdemu z uczestników. Wykorzystane zostały mechanizmy gałęzi do rozwijania większych mechanizmów, które następnie były scalane do głównego drzewa. System pozwolił na bieżąco śledzić zmiany w kodzie wprowadzane przy kolejnych rewizjach.

Emulatory komputerów PC, Qemu oraz BOCHS, pozwoliły na wygodne testowanie pisanego kodu, dzięki tym programom nie istniała potrzeba ciągłych restartów komputera w celu sprawdzenia czy wprowadzane do systemu zmiany przyniosły oczekiwany skutek. Zostało jednak uznane za sprawę ważną, aby system działał także na prawdziwych komputerach klasy PC, po dodaniu istotnych części systemu był więc testowany i dostosowywany tak, aby działał na prawdziwym sprzęcie. Emulator Qemu dodatkowo posłużył do wpięcia debuggera gdb, co umożliwiało prostsze wyłapywanie błędów oraz śledzenie przebiegu programu. Do obydwu emulatorów wraz z projektem dostarczone są proste łaty wypełniające bajty pamięci komputera wzorcem `0xe9`, które pozwalają wykryć więcej błędów z użyciem emulatorów, domyślnie zerujących pamięć RAM.

1.2.2 Opis drzewa z kodem źródłowym.

Drzewo z kodem źródłowym ma ustaloną strukturę, w poniższym opisie katalog `.` oznacza korzeń drzewa.

[2] Instalacja tego programu na potrzeby naszego projektu jest dostępna pod adresem internetowym `http://bitbucket.org/wieczyk/impala/`

- `./doc/doxygen/` - katalog z dokumentacją kodu automatycznie generowaną przez program DoxyGen (po wykonaniu polecenia `make doc` w katalogu wyższym).
- `./doc/handbook/` - źródła tego dokumentu.
- `./sys/` - drzewo z kodem jądra systemu.
- `./sys/arch/` - drzewo z wydzieloną obsługą platform
- `./sys/arch/x86/` - kod obsługi platformy PC z procesorem PentiumPro.
- `./sys/dev/` - biblioteka `libdev.a` zawierająca sterowniki urządzeń.
- `./sys/fs/` - biblioteka `libfs.a` zawierająca obsługę systemów plików.
- `./sys/kern/` - kod właściwy jądra.
- `./sys/kern/sc/` - obsługa wywołań systemowych.
- `./sys/sys/` nagłówki języka C.
- `./usr/` - drzewo z kodem programów i bibliotek.
- `./usr/bin/` - źródła standardowych programów.
- `./usr/sbin/` - źródła systemowych programów.
- `./usr/demos/` - źródła programów demonstracyjnych.
- `./usr/lib/` - poddrzewo zawierające kody bibliotek.
- `./usr/lib/libc/` - biblioteka systemowa (języka C).
- `./usr/lib/libpthread/` - biblioteka wątków użytkownika.
- `./usr/etc/` - katalog ze skryptami startowymi systemu.
- `./mk/` - skrypty dla programów Makefile tworzące nasz system budowania.
- `./image/` - skrypty budujące obraz dyskietki.

Rozdział 2

Dokumentacja użytkownika.

2.1 Uruchomienie systemu.

Dyskietka z systemem ma zainstalowany program ładujący GRUB, który może zostać uruchomiony przez BIOS. Jeżeli w komputerze już istnieje program ładujący obsługujący format ELF (3.10.1) oraz kompresję programu `gzip` to można sprawdzić czy jest w stanie ręcznie uruchomić nasz system, bez konfiguracji BIOSu, aby uruchamiał stację dyskietek. Dla przykładu, jeżeli już używamy programu GRUB na dysku twardym, to można dopisać następujące linijki do jego konfiguracji:

```
# Wpis 1
title Uruchom system ze stacji dyskietek A:
    root (fd0)
    chainloader +1

# Wpis 2
title System operacyjny Impala-LPP
    root (fd0)
    kernel /boot/impala.gz
```

Pierwszy wpis uruchamia program ładujący z dyskietki, tak jak by to zrobił BIOS, a drugi wpis bezpośrednio uruchamia jądro naszego systemu.

Po załadowaniu jądra następuje proces inicjalizacji systemu, w skład którego wchodzi rozpakowanie danych systemu z dyskietki oraz uruchomienie skryptów startowych. Po wykonaniu tej fazy zostaną udostępnione trzy wirtualne terminale, na których będzie uruchomiona powłoka.

2.1.1 Parametry jądra

Użyty w procesie startu progam ładujący GRUB daje możliwość przekazania parametrów do ładowanego jądra. Parametry przekazuje się poprzez klauzulę kernel programu GRUB jako ciąg parametrów oddzielonych znakiem spacji.

- `debug` - wyświetlanie komunikatów diagnostycznych.
- `init` - ścieżka do programu init.

- `iobufs` - ilość buforów w pamięci podręcznej BIO.
- `stacksize` - domyślny rozmiar stosu dla wątków użytkownika.
- `kstacksize` - domyślny rozmiar stosu alternatywnego i wątków jądra.
- `sched_quantum` - kwant czasu przyznawany programom przez planiste systemowego.

2.2 Powłoka systemowa.

Ta dokumentacja nie zawiera dokładnego opisu wbudowanych poleceń powłoki `ash` oraz języka jakim się posługuje, te opisy znajdują się w [6] oraz [5]. Poniżej znajduje się jedynie opis podstawowych komend niezbędnych do swobodnego testowania systemu.

- `cd` - zmiana aktualnego katalogu.
- `echo` - wyświetlenie dostarczonego napisu.
- `exit` - zakończenie pracy.
- `pwd` - wyświetlenie aktualnego katalogu.
- `read` - wczytanie zmiennej z linii poleceń do środowiska, np `read l` wczyta dane do zmiennej `$l`.
- `alias` - stworzenie aliasu dla polecenia, np `alias ls="ls -laiF"`.
- `export` - definiuje zmienną środowiskową, np `export TERM=vt100-8025`.
- `jobs` - drukuje listę zadań uruchomionych w tle.
- `fg` - przywraca zadanie działające w tle do pierwszego planu.

Zmienne środowiskowe są podstawione za napisy `$nazwa_zmiennej`.
Standardowe wejście i wyjście poleceń można przekierowywać za pomocą znaków `<` i `>`.

2.3 Przestrzeń użytkownika.

Z punktu widzenia użytkownika opisane poniżej programy można traktować jako komendy wydawane powłoce systemowej, należy mieć jednak świadomość, że są one zewnętrznymi programami i nie stanowią jej integralnej części. W projekcie udało się zawrzeć minimalny podzbiór podstawowych komend systemów UNIXowych, poniżej znajduje się krótki opis programów, które dostępne są dla użytkownika systemu.

2.3.1 Drzewo katalogów

Drzewo katalogów systemu jest zgodne ze standardem przyjętym w systemach UNIXowych.

- `/bin` - programy standardowe.
- `/etc` - skrypty oraz pliki konfiguracyjne.

- `/tmp` - pliki tymczasowe.
- `/sbin` - programy specjalne.
- `/demos` - programy demonstrujące.
- `/var` - katalog systemowy.

2.3.2 Polecenia standardowe.

W systemie zaimplementowano część podstawowych komend systemów UNIXowych, na uwagę zasługują programy sh oraz vttest, które zostały przeniesione do projektu bez istotnych modyfikacji w ich kodzie.

- `cat` - wypisywanie zawartości pliku na standardowe wyjście.
- `ls` - listowanie zawartości katalogów.
- `minigzip` - kompresja i dekompresja plików.
- `mkdir` - tworzenie katalogów.
- `ps` - listowanie procesów.
- `sleep` - śpi daną ilość sekund.
- `sh` - powłoka systemowa.
- `tar` - taśmowy program archiwizujący.
- `truncate` - przycięcie (stworzenie) pliku do odpowiedniej długości, np `truncate -s 50 plik`.
- `uname` - wyświetlenie informacji o systemie, np `uname -a` wydrukuje nazwę jądra, nazwę wydania, nazwę platformy oraz nazwę komputera.
- `vttest` - testowanie zgodności z terminalem VT100. Warto go użyć jako program demonstrujący emulację terminalu.

2.3.3 Polecenia specjalne.

Polecenia specjalne służą do wykonywania prac administracyjnych, systemowych oraz ewentualnego uzyskiwania informacji na temat systemu.

- `init` - rozruch systemu.
- `ttyvrun` - uruchomienie programu z zadanym terminalem kontrolującym.
- `login` - rozpoczęcie sesji użytkownika.

2.3.4 Programy demonstracyjne.

Ponieważ nasz projekt jest głównie przeznaczony dla programistów to stworzyliśmy kilka dodatkowych programów demonstracyjnych pokazujących możliwości jądra. Wszystkie te programy można skompilować na dowolnym systemie UNIXowym w celu porównania zachowania systemów.

- `pfault` - program demonstrujący obsługę pułapki procesora wygenerowanej przez program odwółujący się do nieprawidłowego adresu `0xdeadbabe`.
- `pipedemo` - program testujący implementację potoków FIFO, przekopiowany ze strony podręcznika standardu. Implementacja potoków jest również testowana przez program `tar`, który uruchamia pocmoniczy program dekompresujący `minigzip` i komunikuje się z nim za pomocą potoku. Potoki można dodatkowo testować powłoką systemową wydając polecenia z użyciem znaku `|`, np `echo "To jest napis" | cat`.
- `pthdemo1` - pierwszy program demonstrujący bibliotekę wątków POSIX. Tworzy dodatkowy wątek w programie, który przez pięć sekund drukuje napis na ekranie napis `I am alive!`, a następnie zwraca napis `"working well"`. Główny wątek programu oczekuje zakończenia pierwszego wątku, a następnie drukuje zwrócony przez niego napis.
- `pthdemo2` - drugi program demonstrujący bibliotekę wątków POSIX. Tworzy blokadę zamkniętą przez główny wątek oraz dwa dodatkowe wątki, próbujące wejść w sekcję krytyczną. Po pięciu sekundach główny wątek zwalnia blokadę i oczekuje zakończenia wątków, które synchronizują się za pomocą blokady.
- `pthdemo3` - trzeci program demonstrujący bibliotekę wątków POSIX. Tworzy blokadę, oraz zmienną warunkową (patrz 3.8.3). Następnie symuluje kolejkę przez licznik, dwa dodatkowe wątki symulują odczytywanie z kolejki, a jeden dodatkowy zapisywanie. Wątek czytający z kolejki używa zmiennej warunkowej do oczekiwania na nową wiadomość, jeżeli kolejka była pusta. Wątek zapisujący używa zmiennej do budzenia innych wątków czekających na zdarzenie. Program symuluje wysłanie ośmiu wiadomości do kolejki.
- `signal` - program demonstrujący obsługę sygnalów. Instaluje uchwyty sygnałów `SIGHUP` oraz `SIGUSR1`, a następnie wysyła je sam do siebie.
- `sysvmsg` - program testujący implementację kolejek wiadomości (patrz 3.9.3). Uruchamiając go z parametrem `create` tworzymy kolejkę wiadomości; z parametrem `send napis` wysyłamy napis do kolejki, a z parametrem `recv` program pobiera i drukuje napis z kolejki. Jeżeli kolejka jest pusta to program zostanie zablokowany przez jądro do czasu pojawienia się wiadomośći w kolejce. Wysyłanie sygnału przerwania przez terminal po naciśnięciu klawiszy CTRL+C nie odblokuje go, ponieważ obecna implementacja nie obsługuje przerywania czekania. Należy w takim wypadku przełączyć się na inny wirtualny terminal i wysłać jakąś wiadomość do kolejki.

  Powłoki systemowej można użyć do demonstracji kontroli zadań, dodając znak `&` na końcu polecenia - uruchamiający je w tle. Można zlecić nigdy nie kończące się polecenie `cat /dev/zero > /dev/zero &`, a następnie użyć polecenia `jobs` oraz `fg`. Polecenie można zakończyć naciskając CTRL+C, co spowoduje wygenerowanie sygnału przerwania `SIGINT` przez terminal.

Można również wydać polecenie `sleep 5 &`, i zobaczyć jak powłoka wykrywa zakończenie zadania w tle. Powłoka informuje o tego typu rzeczach dopiero po wydaniu jakiegoś polecenia przez użytkownika, ponieważ jest zablokowana przez jądro w oczekiwaniu na wejście z terminalu. Wpisując to polecenie można napisać dowolne inne po pięciu sekundach w celu otrzymania komunikatu.

Potoki można w efektowny sposób przetestować wydając polecenie `minigzip -d < /mnt/fd0/impala/syspack.tar.gz | tar tvf -`, które uruchami dwa programy połączone potokiem. Wejściem programu `minigzip` będzie plik `syspack.tar.gz`, a wejściem programu `tar` będzie wyjście poprzedniego programu. Polecenie testuje poprawność archiwum i drukuje jego zawartość.

Rozdział 3

Dokumentacja techniczna.

Nieniejszy rozdział opisuje techniczne aspekty naszego systemu operacyjnego. W tym rozdziale mówiąc system, będziemy mieli na myśli jedynie kod jądra systemu operacyjnego, a mówiąc użytkownik będziemy mieli na myśli jedynie kod programów działających pod kontrolą naszego systemu. Słowa klient będziemy używać w stosunku do mechanizmu lub procedury w systemie wykorzystującej inny mechanizm, np klientem procedury `str_len` jest każda procedura, która z niej korzysta. Klientem mechanizmu buforowanego wejścia-wyjścia jest implementacja systemu plików FAT12.

3.1 Zarządzanie pamięcią.

Ten podrozdział zawiera między innymi pobieżne opisy mechanizmów związanych z zarządzaniem pamięcią w systemie operacyjnym. O wiele bogatszy opis można znaleźć w pozycji [2], a dokładne omówienie aspektów technicznych w [3].

Komórki (bajty) pamięci komputera są adresowane 32 bitowymi liczbami[1], adresy używane przez sprzętową jednostkę pamięci nazywamy adresami fizycznymi, a adresy używane w instrukcjach procesora adresami wirtualnymi.

Pamięć wirtualna jest techniką umożliwiającą tworzenie wielu wirtualnych przestrzeni adresowych. W tym celu wykorzystywany jest tak zwany mechanizm stronicowania, realizowany przez procesor komputera. Stronicowanie polega na podziale pamięci fizycznej na bloczki o ustalonym rozmiarze, zwane ramkami [2]. Pamięć wirtualna jest również podzielona na takie bloczki, zwane stronami [3]. Stronicowanie to przypisywanie stronom odpowiednich ramek, co można zinterpretować jako ustalanie jaka ramka pamięci fizycznej kryje się za daną stroną pamięci wirtualnej. Adresy wirtualne z instrukcji procesora są tłumaczone sprzętowo na adresy fizyczne.

Jest to istotny element dla wielozadaniowości ponieważ zwykle programy nie współdzielą pamięci, a posługują się tymi samymi adresami. Łatwo się o tym przekonać pisząc prosty program w języku C drukujący na ekran adres jakiejś swojej zmiennej. Jeżeli uruchomimy jednocześnie wiele kopii naszego programu to każdy wydrukuje ten sam adres, mimo że każdy ma swoją własną pamięć.

Moduł pamięci wirtualnej jest podzielony na kilka warstw:

[1] Co daje 2^{32} różnych adresów, czyli możliwość zaadresowania 4GB pamięci.

[2] ramka - page frame

[3] strona - page

- `vm_pmap` (*page map*) - obsługa odwzorowania stron na ramki.
- `vm_seg` - zarządzanie wirtualnymi segmentami.
- `vm_space` - zarządzanie wirtualnymi przestrzeniami adresowymi.

Obsługa pamięci wirtualnej powstała pod wpływem materiałów omawiających zarządzanie pamięcią wirtualną w systemach SVR4[4] i Mach[4][5] Nazewnictwo procedur było wzorowane na drugim z wymienionych modułów.

3.1.1 Mapa pamięci.

Mapa pamięci jest z góry ustalona (rys 3.1). Dla programów użytkownika są przeznaczone adresy poniżej 3GB, zwane niższymi, a dla jądra adresy powyżej, zwane wyższymi. Każdy program wykonywany przez procesor wymaga stosu oraz sterty. Stertą nazywamy segment dynamicznie przydzielanej pamięci przez program podczas działania.

Mapa pamięci użytkownika jest narzucona przez obsługiwany format plików wykonywalnych A.out, omówiony w 3.10.2. Stosy wątków użytkownika są przydzielane przez system na końcu jego przestrzeni adresowej. Ten sposób postępowania jest uzasadniony tym, że sterta programu musi być ciągła (z punktu widzenia jądra), a nowe wątki, wraz z ich zapotrzebowaniem na stos, mogą przybywać podczas pracy programu.

Przestrzeń jądra wygląda podobnie, wpływ na nią wywarł format ELF omówiony w 3.10.1. Z tych samych względów stosy wątków jądra oraz stosy alternatywne wątków użytkownika są umieszczane w końcowych adresach.

Rozkład pamięci fizycznej różni się od przedstawionego powyżej (wirtualnego). Zawartość pamięci w adresach poniżej 1MB jest związana ze starszymi modelami procesorów oraz BIOSem (kod podprogramów obsługi przerwań w trybie rzeczywistym itp.), stąd na rysunku pojawiła się nazwa BIOS. Między innymi w tej pamięci jest odwzorowana pamięć karty graficznej, a system operacyjny umieszcza tam bufory dla DMA[6] szyny ISA, na której znajduje się kontroler stacji dyskietek.

3.1.2 Odwzorowania stron.

Moduł pamięci wirtualnej opisuje ramki pamięci za pomocą typu `vm_page_t`. Zawarte informacje to licznik odniesień, w ilu odwzorowaniach dana ramka się znajduje, oraz adres fizyczny. Na wewnętrzne potrzeby modułu przy niektórych stronach zapisywana jest także informacja o adresie wirtualnym strony, w jaki jest ta ramka odwzorowana.

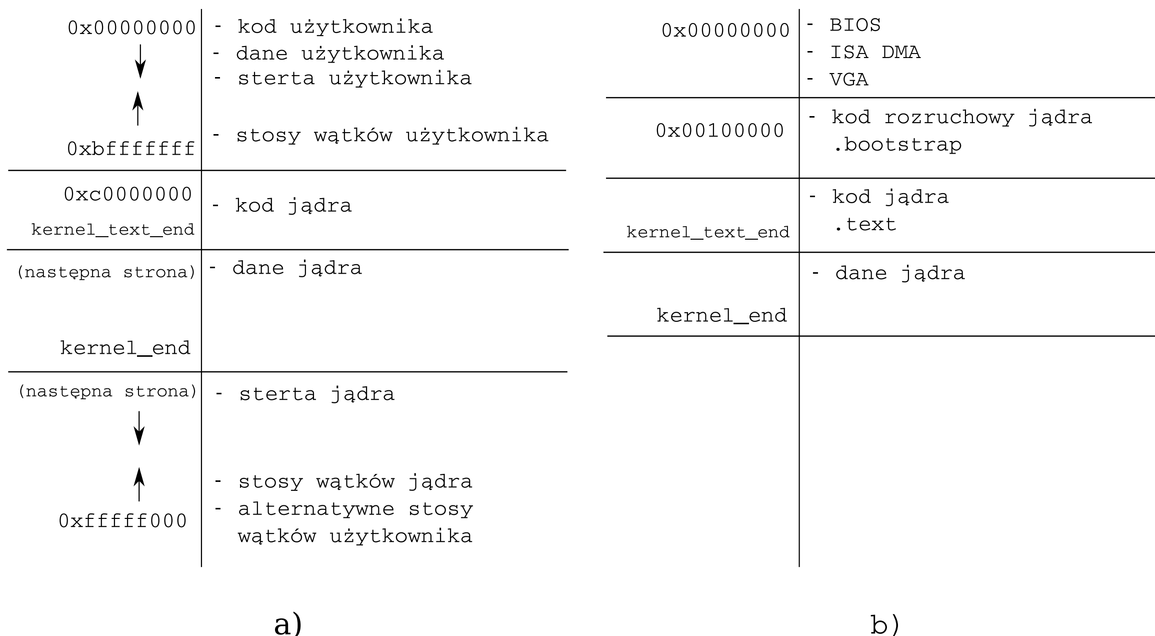
Opisy wszystkich ramek w komputerze są umieszczone w tablicy `vm_pages`. Jeżeli przy niektórych ramkach została zapisana informacja o adresie wirtualnym, to tablica może posłużyć też jako odwrotna tablica stron - tzn. do tłumaczenia adresu fizycznego na adres wirtualny.

Przy rozruchu systemu każda ramka znajdująca się za jądrem jest wpinana w listę wolnych ramek `vm_free_pages`.

[4] Jest to skrócona nazwa systemu jednej z wersji systemu UNIX: „System V: release 4".

[5] W dzisiejszych systemach operacyjnych pochodzących z UNIXa można znaleźć następców tych modułów: System Solaris wywodzi się z systemu piątego, a pamięć wirtualna systemu Mach przeniknęła do BSD.

[6] Direct Memory Access - technologia pozwalająca kopiować dane między pamięcią RAM a urządzeniem przez szynę, bez udziału procesora. Używanie DMA zwalnia programistę z ręcznego (udział procesora) wysyłania (odbierania) bajtów z (do) urządzenia.



Rysunek 3.1: Mapa pamięci a) wirtualnej, b) fizycznej. Strzałki oznaczają w którą stronę dany segment rośnie.

Tłumaczenie adresów przez procesor odbywa się w dwóch etapach. Pierwszy etap to segmentacja, gdzie adresy z instrukcji są tłumaczone na adresy liniowe. Procesor podczas tłumaczenia korzysta z tablicy deskryptorów, z której między innymi odczytuje informacje o adresie bazowym i długości segmentu. Adres liniowy powstaje poprzez przesunięcie adresu z instrukcji o adres bazowy danego segmentu[7]. Ta właściwość nie jest wykorzystywana w naszym systemie, dlatego wszystkie adresy bazowe wynoszą 0, dzięki czemu adres liniowy jest tożsamy z adresem instrukcji[8].

Drugim etapem jest przetłumaczenie adresu na adres fizyczny z użyciem katalogu stron. Sam katalog stron nie opisuje z powodów technicznych wszystkich stron w pamięci, ponieważ stron może być aż 2^{24} to taki katalog zajmowałby zbyt dużą ilość pamięci (biorąc pod uwagę, że każda przestrzeń adresowa ma swój katalog stron). Katalog zajmuje jedną stronę pamięci i jest tablicą o 1024 elementach. Każda pozycja w katalogu zawiera adres fizyczny tablicy stron opisującej 4MB pamięci. Ponieważ $1024 \cdot 4MB = 4GB$ to katalog stron opisuję całą pamięć jaką mogą adresować instrukcje procesora. Tablica stron ma taką samą budowę jak katalog, z tym że jej pozycje opisują fizyczne adresy (zatem każda pozycja tablicy stron opisuje 4kB pamięci) ramek.

Procesor posiada rejestr kontrolny (CR3), w którym trzyma fizyczny adres katalogu stron.

[7] Sama segmentacja spełnia jeszcze rolę ochronną, sprawdza czy program nie wyskoczył za segment oraz czy ma odpowiednie prawa dostępu.

[8] Z tego mechanizmu nie korzysta się w ogólnym zarządzaniu pamięcią, jedynie przy implementacji prywatnych segmentów dla wątków użytkownika. Prawdopodobnie jest to spowodowane tym, że język C nie wspiera wykorzystania tego mechanizmu. Przesunięcia spowodowałyby, że użycie adresów z segmentu stosu na kodzie operującym na segmencie danych byłoby nieprawidłowe. Stąd nie można byłoby używać np procedury `strlen` do tablic będących zmiennymi lokalnymi jak i do tablic będących zmiennymi globalnymi. Wsparcie wymagałoby wprowadzenia nowego typu wskaźników, w których oprócz samego adresu byłby zapisany deskryptor segmentu - stare kompilatory języka C na system DOS w tym celu rozszerzały język o słówko kluczowe `far`.

Każda przestrzeń adresowa posiada swój własny katalog stron, którego adres jest wpisywany w ten rejestr przy zmianie kontekstu. Warstwa pamięci wirtualnej odpowiedzialna za zarządzanie odwzorowaniami stron dla każdej ramki przydzielonej na katalog lub stronę zapisuje jej adres wirtualny, pod którym jest widziana w przestrzeni jądra. Dzięki temu operując na katalogu stron, w którym wskaźniki do tablic są adresami fizycznymi, może szybko znaleźć prawidłowy adres wirtualny.

Odwzorowanie stron jest opisane przez typ `vm_pmap_t`. Przed programistą systemu dwustopniowa konstrukcja katalogu jest ukryta, ponieważ nie jest to istotne w ogólnym zarządzaniu pamięcią oraz zrobiłoby moduł mniej przenośnym[9].

Operacje na odwzorowaniu zarządzają licznikiem odniesień do ramki, odpowiednio zwiększając oraz zmniejszając przy dodawaniu i kasowaniu wpisów. Ramki z wyzerowanym licznikiem trafiają ponownie na listę wolnych ramek i nadają się do ponownego użycia. Dodatkowymi operacjami są ręczne tłumaczenie adresów wirtualnych na fizyczne oraz kopiowanie stron pomiędzy odwzorowaniami, wykorzystywane przy współdzieleniu pamięci.

Jądro musi być odwzorowane w każdej przestrzeni adresowej, ponieważ w każdej chwili musi znajdować się w pamięci wirtualnej kod podprogramów obsługi przerwań oraz zleceń od programu użytkownika.

Odwzorowanie jądra w każdą przestrzeń adresową wiąże się z dwiema trudnościami technicznymi. Pierwsza, czyszczenie pamięci podręcznej procesora, przechowującej fragmenty tablic stron, przy każdej zmianie aktualnej przestrzeni adresowej. Ponieważ kod jądra jest w ciągłym użyciu (zegar systemowy oraz obsługa programów) to procesor będzie musiał ściągnąć z aktualnego katalogu i tablic stron wpisy, które przed chwilą wykasował. Druga trudność wynika z posiadania własnego katalogu przez każdą przestrzeń. Dodanie nowych tablic stron opisujących pamięć jądra wymaga edycji wszystkich katalogów.

Pierwsza trudność została rozwiązana sprzętowo przez firmę Intel w procesorach Pentium Pro (i686) przez wprowadzenie specjalnego atrybutu GP[10] informującego procesor. że dany wpis znajduje się we wszystkich tablicach stron opisujących te same 4MB pamięci. Druga trudność została rozwiązana przez przydzielenie wszystkich tablic stron mogących opisywać przestrzeń jądra podczas rozruchu systemu. Dzięki temu katalog stron każdej nowo utworzonej przestrzeni zawiera wpisy ze wszystkimi tablicami jakie mogą zostać użyte przez jądro i nie wymaga późniejszej edycji. Wadą tego rozwiązania jest to, że jądro może nigdy podczas swojej pracy nie użyć wszystkich tablic. Dla przestrzeni jądra o rozmiarze 1GB należy przydzielić 256 tablic stron, co łącznie daje koszt 4MB.

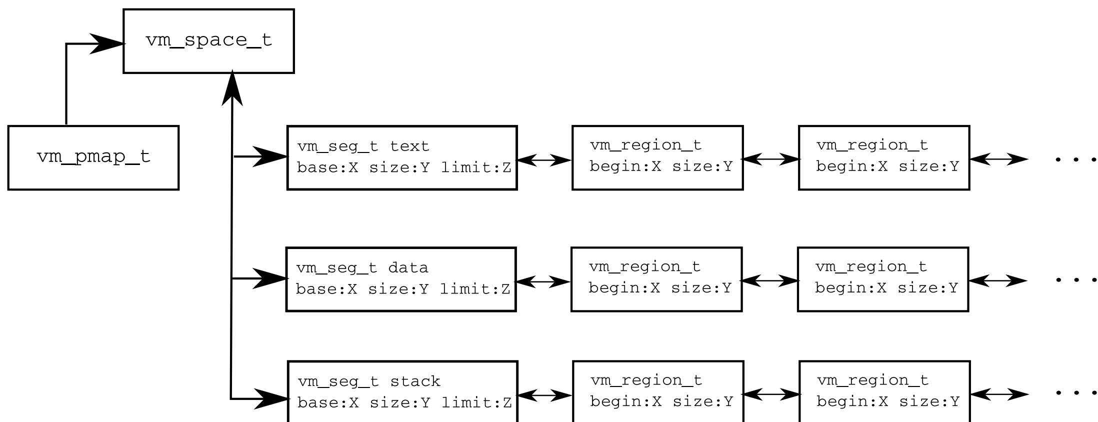
3.1.3 Wirtualne segmenty i przestrzenie adresowe.

Wirtualna przestrzeń adresowa jest dzielona na trzy wirtualne segmenty: segment tekstu (kodu), danych (sterta i dane z pliku) oraz stosów. Segmenty danych i stosu są we wzajemnej relacji, ponieważ ich wzrost przybliża koniec jednego do początku drugiego, zatem system musi pilnować momentu, w którym segmenty mogłyby się spotkać - taki moment oznacza koniec pamięci dla programu.

Przestrzeń adresowa jest opisywana przez typ `vm_space_t`, który zawiera w sobie odwzorowanie stron pamięci oraz opisy trzech wyżej wspomnianych segmentów.

[9]Rozszerzenie procesora pozwalające używać 48-bitowych adresów pamięci (zwiększając ograniczenie pamięci powyżej 4GB) wprowadza trójstopniową strukturę katalogu - dzięki ukryciu budowy zaprogramowanie obsługi takiego rozszerzenia wiązałoby się z mniejsza ilością zmian w kodzie.

[10]Global Page - strona globalna.



Rysunek 3.2: Techniczne ujęcie przestrzeni adresowej.

Segment jest opisywany przez adres bazowy, aktualny rozmiar oraz ograniczenie górne na rozmiar. Przydzielanie i niszczenie stosów podczas pracy programu i jądra oraz odwzorowywanie różnych fragmentów pamięci w stercie jądra i niszczenie tych odwzorowań powoduje fragmentację tych segmentów. Z tego powodu wirtualne segmenty nie muszą być ciągłymi przestrzeniami, lecz mogą być dynamicznie zmieniającymi się obszarami. W związku z tym segment zawiera w sobie listę ciągłych obszarów, opisywanych przez typ `vm_region_t`.

Rysunek 3.2 przedstawia opis przestrzeni adresowej używany w module pamięci wirtualnej.

3.1.4 Sterta jądra.

Stertą jądra zarządza algorytm opracowany przez Jeffa Bonwicka dla Sun Microsystems, przedstawiony na konferencji USENIX[1]. Ten algorytm, zwany alokatorem płytowym[11], przeniknął również do innych systemów operacyjnych jak FreeBSD, Linux, NetBSD.

Algorytm jest nadbudową na obsługę segmentu realizowaną przez pamięć wirtualną. Idea opiera się o pojęcię schowka (`kmem_cache_t`), który przechowuje w sobie elementy o ustalonym rozmiarze.

Schowek przechowuje elementy na płytach, które są tablicami elementów.

3.2 Sterowniki urządzeń.

Większość sterowników urządzeń w systemach UNIXowych tworzy tak zwane pliki urządzeń, przez które można obsługiwać dane urządzenie. W naszym systemie zarządzaniem takimi plikami zajmuje się specjalny system plików `devfs`, zamontowany w katalogu `/dev`. Sterowniki plików urządzeń dzielą się na trzy kategorie:

- urządzenia blokowe - sterowniki obsługujące takie urządzenia jak stacje dyskietek, dyski twarde itp.
- urządzenia znakowe - sterowniki obsługujące urządzenie, widziane jako strumień bajtów.
- terminale - specjalny rodzaj urządzeń znakowych.

[11]alokator płytowy - slab allocator

Każdy sterownik dostarcza systemowi tak zwaną deskę rozdzielczą (`devsw_t` - *device switch*), która zawiera implementację procedur obsługi pliku urządzenia. Deska rozdzielcza zawiera następujące pola:

- `d_open` - obsługa otwarcia pliku urządzenia.
- `d_close` - obsługa zamknięcia pliku urządzenia.
- `d_ioctl` - obsługa poleceń kontrolnych dla sterownika.
- `d_write` - obsługa zlecenia wyjścia ze strumienia znakowego, może blokować klienta na czas wykonywania operacji.
- `d_read` - obsługa zlecenia wejścia ze strumienia znakowego, może blokować klienta na czas wykonywania operacji.
- `d_strategy` - obsługa zleceń operacji wejścia-wyjścia na urządzeniach blokowych, działa asynchronicznie, tzn nie blokuje klienta na czas wykonywania operacji.
- `type` - informacja o kategorii, do jakiej urządzenie należy
  - `DEV_BDEV` - dla urządzeń blokowych.
  - `DEV_CDEV` - dla urządzeń znakowych.
  - `DEV_TTY` - dla terminali.

Podział na urządzenia znakowe i blokowe oraz rozróżnienie procedur obsługujących zlecenia wejścia-wyjścia jest koniecznie ze względu na wydajność oraz obsługę tych urządzeń. Na urządzeniach znakowych użytkownik może wykonywać te same operacje jak na plikach, ponieważ pliki są również widziane jako strumienie znakowe. Dozwolone są takie operacje jak przeczytanie jednego bajtu, dwóch kilobajtów czy pięciu megabajtów.

Urządzenia blokowe jak dyski twarde cechuje inna metodą dostępu, fizycznie są one podzielone na bloki danych (sektory). Stąd wszelkie transfery danych między pamięcią RAM a urządzeniem nie są tak elastyczne pod względem wielkości jak strumienie znaków.

Z powodu tych różnic jednolita obsługa wejścia-wyjścia dla tych dwóch rodzajów urządzeń musiałaby emulować urządzenia blokowe jako znakowe, co nie byłoby wydajne. Dzięki wyodrębnieniu oddzielnej procedury sterownik zawsze dostaje zlecenia, których długość jest wielokrotnością długości sektoru oraz może sam zadecydować w jakiej kolejności najlepiej je wykonać (co tłumaczy nazwę procedury - *strategia*). Zagadnienia wejścia-wyjścia urządzeń blokowych są omówione w rozdziale 3.5.

Operacje wejścia-wyjścia na terminalach działają na tej samej zasadzie co urządzeń blokowych, z tą różnicą że te operacje przechodzą dodatkową warstwę zwaną reżimem linii. Te zagadnienia są szerzej omówione w rozdziale 3.6.

W kodzie systemu większość sterowników, poza specjalnymi, jest wydzielona do oddzielnej biblioteki `libdev.a` (lokalizacja: `sys/dev/`). System operacyjny nie zna bezpośrednio wszystkich zaimplementowanych w niej sterowników, pobiera natomiast tablicę `devtab` zawierającą procedury inicjujące sterowniki. Specjalnymi sterownikami nie wydzielonymi do wymienionej biblioteki są wirtualne terminale emulowane przez konsolę, która jest częścią jądra.

System z plikami urządzeń wiąże deskryptor urządzenia `devd_t`, który zawiera w sobie takie informacje jak nazwa urządzenia, wskaźnik na prywatne dane sterownika oraz jego deskę

rozdzielczą. Jeden sterownik może stworzyć wiele deskryptorów urządzeń, w zależności od tego ile wykryje urządzeń, które może obsługiwać. Przykładowo, zaimplementowany kontroler stacji dyskietek, jeżeli wykryje dwie stacje dyskietek w komputerze, to tworzy odpowiednio dwa pliki urządzeń: `/dev/fd0` dla stacji `A:` oraz `/dev/fd1` dla stacji `B:`. System taką stację dyskietek widzi jako dwa różne deskryptory, które współdzielą ze sobą deskę rozdzielczą.

Nie wszystkie urządzenia w sprzęcie komputerowym są obsługiwane za pomocą tego modelu sterowników. Sterowniki niektórych urządzeń mogą być programowane, jako moduły dostarczające pewną funkcjonalność, a nie jako ogólnie dostępne pliki urządzeń. Ta uwaga dotyczy niskopoziomowych urządzeń, które są wykorzystywane przez jądro do dostarczania pewnych usług lub do implementacji innych sterowników. Udostępnienie ich jako pliki widziane dla użytkownika nie wnosiłoby żadnych korzyści, a dodało trudności implementacyjnych. Na przykład, sterownik szyny ISA udostępnia zbiór procedur `bus_isa_dma_` wykorzystywanych w sterowniku kontrolera stacji dyskietek. Sterownik zegara systemowego instaluje podprogram przerwania, który przy każdym tyknięciu uruchamia ogólną obsługę zegara w systemie. Obsługa przerwań sprzętowych jest omówiona w rozdziale 3.3.

3.2.1 Szyna ISA.

Sterownik szyny ISA jest dostarczany przez obsługę architektury sprzętu, jest on potrzebny w naszym systemie ponieważ kontroler stacji dyskietek znajduje się na niej. Zadaniem sterownika jest dostarczyć procedury obsługujące transfery DMA.

Za obsługę transferów DMA w komputerach domowych są odpowiedzialne dwa kontrolery Intel 8237A. Każdy z nich obsługuje cztery kanały, służące do wykonywania transferów. Każde urządzenia ma przypisany swój kanał, a przed rozpoczęciem transferu należy go odpowiednio zaprogramować.

Deskryptor kanału musi zostać przydzielony za pomocą procedury `bus_isa_dma_alloc`, a programowanie kanałów odbywa się przez `bus_isa_dma_prepare`. Ponieważ te kontrolery są starą technologią to obsługują jedynie transfery do fragmentów pamięci RAM których adres mieści się w 24 bitach. Taką pamięć nie zawsze można przydzielić, w szczególności po dłuższej pracy systemu, ten problem jest rozwiązany przez przydzielanie stałych buforów kanałom przy starcie systemu w adresach poniżej 1MB (zobacz rysunek 3.1). Po wykonaniu transferu należy wykonać procedurę `bus_isa_dma_finish`.

Para procedur `_prepare` i `_finish` kopiują odpowiednio dane pomiędzy stałym buforem kanału, a buforem danym przez klienta.

3.2.2 Sterownik stacji dyskietek.

Omówienie sterownika stacji dyskietek wymaga uprzedzenia kilku wiadomości z rozdziału 3.5. Jak wspomniano wcześniej procedura `d_strategy`. służąca do zlecenia operacji wejścia-wyjścia, posługuje się nagłówkami buforów. Nagłówki (`iobuf_t`), zawierają w sobie bufor operacji, logiczny numer bloku na urządzeniu, długość transferu oraz informację o kierunku transferu `BIO_READ` lub `BIO_WRITE`[12]. Sterownik informuje klientów o zakończeniu operacji za pomocą procedur `bio_done` oraz, w przypadku błędu, `bio_error`.

Procedura inicjalizująca sterownik `fdc_init` sprawdza w pamięci CMOS jakie są zainstalowane stacje dyskietek i odpowiednio na podstawie tych informacji tworzy pliki urządzeń

[12] Wszystkie kierunki odnoszą się do podmiotu, a nie pamięci. Stąd „czytanie” wszędzie oznacza czytanie z podmiotu (np. urządzenia, pliku) do pamięci, a nie czytanie z pamięci do podmiotu.

`fd0` i `fd1`. Pliki urządzeń nie są oddzielnymi instancjami sterownika, ponieważ zależą od tego samego kontrolera stacji dyskietek. Zatem procedury obsługi pliku urządzeń są pomostem do ogólniejszego sterownika FDC (*floppy disk controller*). W obecnej implementacji kolejka zleceń wejścia-wyjścia jest wspólna dla wszystkich stacji na danym kontrolerze.

Żadna z procedur sterownika kontrolera nie czeka na zakończenie zleconej przez siebie operacji. Kontroler po wykonaniu zadania informuje o jego zakończeniu zgłaszając przerwanie sprzętowe, a podprogram jego obsługi uruchamia dalej odpowiednie procedury. Dzięki temu podejściu sterownik działa w pełni asynchronicznie w stosunku do swoich klientów (dzięki czemu nie blokuje żadnego z nich).

Implementacja procedury `d_strategy` kolejkuje nagłówek buforu w kolejce zadań oraz jeżeli kolejka była pusta to rozpoczyna obsługę zlecenia. Jeżeli kolejka nie jest pusta to kontroler wykonuje jedno ze zleceń, po wykonaniu którego sam zacznie obsługiwać następne.

Obsługa zlecenia jest implementowana przez procedurę `fdc_work`, która w zależności od obecnego położenia głowicy zleca sterownikowi przesunięcie głowicy lub wykonanie transferu.

Zmiana położenia głowicy jest wykonywana przez procedurę `fdc_seek`, która zleca to zadanie bezpośrednio kontrolerowi. Po wykonaniu zadania jest zgłaszane przerwanie, którego obsługa zleca wykonanie transferu.

Transfer jest wykonywany przez procedurę `fdc_io`, która przeprogramowuje kanał DMA i zleca rozpoczęcie transferu kontrolerowi. Stacja dyskietek 1440kB może obsłużyć wszystkie sektory do końca ścieżki w jednym transferze. Jeżeli jednak zlecony transfer sterownika przekracza granicę ścieżki to musi zostać podzielony na mniejsze transfery częściowe. Obsługa zgłoszonego przerwania, informującego o zakończeniu operacji, informuje klienta sterownika o zakończeniu operacji używając `bio_done` lub wykonuje kolejny transfer częściowy za pomocą wyżej wymienionej procedury.

Transfery stacji dyskietek często kończą się błędami, które przy ponownej próbie nie występują. Sterownik obsługuje to nadając każdemu transferowi częściowemu licznik prób, który jeżeli zostanie przekroczony to operacja jest zakończona procedurą `bio_error` z numerem błędu `EIO`. Przy konieczności powtórzenia transferu częściowego jego długość jest obcinana do jednego sektora, obcięte sektory będą rozpatrywane w kolejnym transferze częściowym.

3.2.3 Sztuczne urządzenia.

W naszym systemie istnieją trzy sztuczne urządzenia:

- `null` - puste urządzenie znakowe
- `zero` - nieskończony strumień zer
- `md` (*memory disk*) - urządzenie blokowe działające w pamięci RAM. Było przez nas używane we wczesnych pracach nad wirtualnym systemem plików.

3.3 Przerwania.

Przerwaniem (*interrupt*) jest wywłaszczenie pracy procesora przez zdarzenie, które powoduje wykonanie kodu jego obsługi. Przerwania mogą być generowane przez:

- sprzęt komputerowy - nazywamy je wtedy przerwaniami sprzętowymi

- kod programu - nazywamy je wtedy przerwaniami programowymi
- procesor - nazywamy je wtedy pułapkami lub wyjątkami procesora

Obsługę przerwań opisuje tablica deskryptorów dostarczona procesorowi na początku inicjalizacji systemu. W niej są zapisane takie informacje jak adres podprogramu obsługi oraz poziom uprzywilejowania pracy procesora. Po zakończeniu obsługi przerwania procesor wraca do wykonywania wywłaszczonego zadania.

Przerwanie może nadejść w każdym momencie pracy, pomiędzy dowolnymi instrukcjami procesora. W systemie z podziałem czasu stwarza to ryzyko uszkodzenia struktur danych, które są jednocześnie modyfikowane przez wywłaszczony program oraz obsługę danego przerwania. Np. sterownik stacji dyskietek obsługując przerwanie może pobrać zlecenie wejściawyjścia z kolejki, która właśnie była modyfikowana przez wywłaszczony program, chcący zakolejkować kolejne zlecenie.

Problem jest rozwiązywany przez wprowadzenia poziomów uprzywilejowania dla przerwań. Przerwania z niższym priorytetem niż obecny są odwlekane. Do zwiększania priorytetu służą procedury `splXXX`, gdzie `XXX` jest nazwą poziomu. Zwracają one obecny poziom uprzywilejowania. Procedury nigdy nie zmniejszają poziomu uprzywilejowania, co zapobiega sytuacjom gdzie priorytet zostanie po cichu zmniejszony przez jedną z wykorzystanych procedur w kodzie mającym działać z większym. Priorytet jest przywracany za pomocą procedury `splx`.

Przerwania sprzętowe są obsługiwane przez chipset Intel 8259A[13], za pomocą niego jest również wykonane manipulowanie poziomami uprzywilejowania i odwlekania obsługi przerwań.

Obecnie wyszczególnionymi poziomami uprzywilejowania są:

- `TTY` - poziom odwleka wszystkie przerwania związane z obsługą terminali, klawiatur.
- `BIO` - poziom odwleka wszystkie przerwania związane z obsługą urządzeń blokowych.
- `SOFTCLOCK` - poziom odwleka wszystkie przerwania związane z obsługą czasomierzy, w tym zegar systemowy.
- `HIGH` - poziom odwleka wszystkie przerwania w systemie.

Manipulując poziomem uprzywilejowania należy zwrócić szczególną uwagę na to, że może to opóźniać obsługę urządzeń przez sterowniki, co może wpłynąć na wydajność systemu.

Poziom uprzywilejowania można interpretować również jako element tworzenia sekcji krytycznych blokujących sterowniki lub zegar systemowy uruchamiający program planisty.

Nieuważne mieszanie zmian poziomu uprzywilejowania z mechanizmami synchronizacji na wątkach może spowodować zakleszczenie całego systemu, ponieważ mechanizmy synchronizacji wątków będą czekać aż inny proces opuści sekcję krytyczną, a zmiana poziomu uprzywilejowania wyłączy program planisty, który odpowiada za zmianę aktualnie wykonywanego wątku.

Sterowniki urządzeń mogą instalować swoje podprogramy obsługi za pomocą procedury `irq_install_handler`, przypisując od razu odpowiedni poziom uprzywilejowania.

Przerwania będące wyjątkami procesora są przez niego rzucane w szczególnych przypadkach jak nieprawidłowy dostęp do pamięci czy wykonanie nieprawidłowej instrukcji. System

[13] Tak naprawdę ten chipset nie jest już produkowany, jest obecnie emulowany przez mostek na płycie głównej. W nowoczesnych procesorach jest zastąpiony nowym kontrolerem wbudowanym w procesor.

Impala zostaje zatrzymany przy każdej pułapce procesora, oprócz pułapki związanej z nieodpowiednim dostępem pamięci. W takim wypadku, w zależności od tego kto wykonał nieprawidłowy dostęp są podejmowane różne akcje. Jeżeli wykonał go proces użytkownika to jest do niego dostarczany sygnał `SIGSEGV`, jeżeli jądro to system jest zatrzymywany.

Przerwania programowe są wykorzystywane do komunikacji użytkownika z systemem, mechanizm jest omówiony szerzej w 3.9.1.

3.4 Strumieniowe wejście-wyjście.

3.5 Buforowane wejście-wyjście.

Operacje wejścia-wyjścia są na urządzeniach blokowych opisywane przez nagłówki buforów (`iobuf_t`). Każdy nagłówek buforu zawiera w sobie następujące pola:

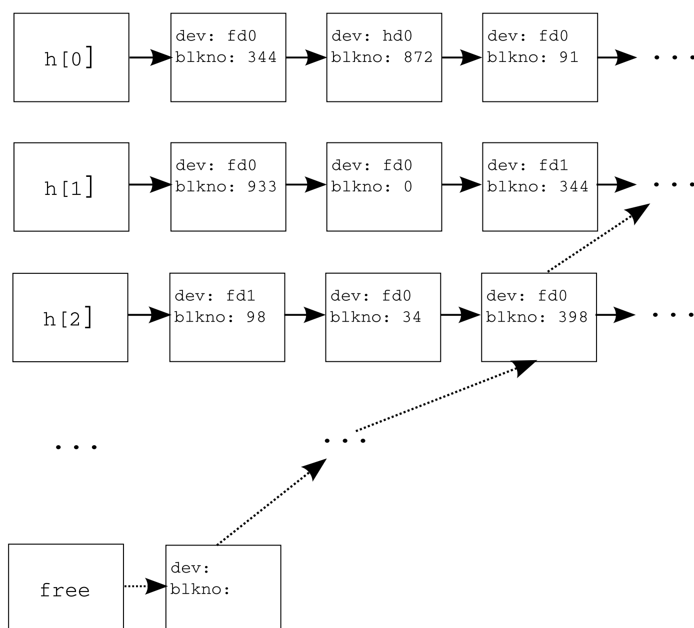
- `addr` - adres bufora
- `size` - długość bufora w bajtach
- `bcount` - długość bufora w blokach
- `blkno` - logiczny numer bloku
- `flags` - znaczniki
- `errno` - numer błędu
- `oper` - kierunek transferu `BIO_READ` lub `BIO_WRITE`
- `resid` - ilość pozostałych bajtów do przesłania (na potrzeby sterowników)
- `dev` - deskryptor urządzenia
- `sleepq` - śpiąca kolejka

Zdefiniowane znaczniki `iobuf_t.flags`:

- `BIO_DONE` - operacja wejścia-wyjścia zakończona
- `BIO_ERROR` - wystąpił błąd podczas ostatniej operacji wejścia-wyjścia
- `BIO_VALID` - dane w buforze są prawidłowe
- `BIO_CACHE` - nagłówek jest w tablicy haszującej
- `BIO_BUSY` - nagłówek jest zajęty przez klienta

System udostępnia mechanizm buforowanego wejścia-wyjścia, który jest warstwą pomiędzy swoimi klientami (modułami obsługi systemów plików) a sterownikami urządzeń. Mechanizm posługuje się stałą ilością nagłówków buforów, które oprócz opisywania operacji dla sterowników służą do zapamiętywania ich wyników. Zapamiętane dane mogą ograniczyć potrzebę korzystania z urządzenia.

Należy zwrócić uwagę, że buforowane są operacje wejścia-wyjścia, nie sam nośnik urządzenia. Ta subtelna różnica objawia się tym, że jeden bufor może pamiętać ciąg stu sektorów



Rysunek 3.3: Przykładowa tablica haszująca. Bloczki z pierwszej kolumny oznaczają nagłówki list, a z pozostałych kolumn nagłówki buforów. Listy $h[i]$ oznaczają listy tablicy haszującej, a `free` oznacza listę wolnych nagłówków.

zaczynając od dwudziestego, a drugi bufor może pamiętać ciąg dziesięciu sektorów zaczynając od pięćdziesiątego tego samego urządzenia. Mechanizm nie zapewnia, że buforowanie „wspólnych" sektorów zawiera te same dane. Odpowiedzialność tworzenia sensownych operacji wejścia-wyjścia spada na klientów tego mechanizmu.

3.5.1 Tablica haszująca.

W celu przyspieszenia szukania buforów w pamięci system posługuje się tablica haszującą, która korzysta z funkcji haszującej zależnej od adresu deskryptora urządzenia oraz pierwszego bloku operacji.

$$\begin{aligned} hash(\texttt{dev}, \texttt{blkno}) &= h(\texttt{adres-deskryptora(dev)} \times (\texttt{blkno}+1)) \\ h(k) &= ((ak+b) \bmod p) \bmod m \end{aligned}$$

Nagłówki są nawlekane na dwie listy, listę wolnych nagłówków jeżeli z danego buforu nikt nie korzysta oraz na listę tablicy haszującej. Przykładowa tablica jest przedstawiona na rysunku 3.3.

Bufor znajduje się na wolnej liście tylko wtedy, gdy nie jest używany przez żadnego klienta. Może on też jednocześnie znajdować się w tablicy haszującej, co oznacza że buforuje poprzednio wykonaną operację.

Tablica haszująca oraz lista wolnych buforów znajduje się w strukturze `bufhash`.

3.5.2 Cykl życia nagłówka buforu.

Stała ilość buforów wymaga ustalenia rygorystycznych zasad posługiwania się nimi. Każdy nagłówek może być dany jednemu klientowi.

Klient może otrzymać bufor za pomocą dwóch procedur `bio_getblk` oraz `bio_read`. Obydwie pobierają te same argumenty: urządzenie na którym ma być wykonany transfer, numer logiczny bloku oraz długość, która musi być wielokrotnością sektora danego urządzenia.

Procedura `bio_getblk` przeszukuje najpierw tablicę haszującą w celu znalezienia danej operacji. Jeżeli nagłówek jest używany przez innego klienta, to klient o niego proszący jest usypiany na śpiącej kolejce `iobuf_t.sleepq`. Po obudzeniu procedura jest restartowana, ponieważ dany nagłówek buforu mógł zmienić buforowaną operację.

Jeżeli odpowiedni bufor nie zostanie znaleziony w tablicy haszującej to nagłówek jest przydzielany z listy wolnych buforów, jeżeli wzięty nagłówek znajduje się również w tablicy haszującej (to znaczy buforuje inną operację niż my szukamy) to należy go z niej usunąć.

Zdobyty nagłówek buforu jest organizowany przez procedurę `buf_alloc`, która sprawdza czy nie należy przydzielić buforu lub zmienić rozmiaru obecnie przydzielonego. Następnie jest naznaczany flagą `BIO_BUSY` i zwracany klientowi.

Procedura `bio_read` jest nakładką na wyżej omówioną, jeżeli bufor nie jest naznaczony flagą `BIO_VALID` to nagłówek jest przekazywany jako zlecenie do sterownika urządzenia. W takim wypadku klient jest blokowany na czas wykonania zlecenia.

Operacja zapisu może zostać wykonana za pomocą operacji `bio_write`, po wykonaniu działania klient musi zwrócić bufor mechanizmowi używając procedury `bio_release`.

Dla sterowników urządzeń są dostarczone procedury `bio_done` i `bio_error` informujące klientów o zakończeniu operacji zleconej przed dany nagłówek. Obydwie procedury budzą klienta oczekującego na bufor za pomocą `bio_wait`.

3.6 Interfejs terminali.

Programy użytkownika wymagają od systemu ujednoliconego dostępu do urządzeń umożliwiających im komunikację z użytkownikiem. Urządzenia służące do takiej komunikacji nazywamy terminalami. Terminale można podzielić na następujące grupy:

1. Terminal zewnętrzny, komunikujący się z komputerem poprzez port szeregowy bądź modem

2. Terminal zintegrowany z komputerem, komunikacja następuje np. poprzez dzieloną pamięć. (Klawiatura, monitor)

3. Terminal sieciowy - komunikacja np. poprzez Ethernet

Wszystkie te urządzenia widoczne są dla użytkownika w ujednoliconej formie - jako urządzenia terminalowe. W systemie Impala z terminalami związana jest struktura `tty_t`. Rejestrowanie nowego urządzenia terminalowego w systemie następuje poprzez funkcję `tty_create`. Jako argumenty przyjmuje ona nazwę nowego urządzenia, dowolną strukturę z prywatnymi danymi urządzenia, oraz wskaźnik do funkcji obsługującej zapis na tym urządzeniu. W ten sposób, system terminali stanowi nakładkę na inne urządzenia, implementującą ich typowe funkcjonalności w zunifikowany sposób.

3.6.1 Reżim linii.

Aby zapewnić zgodność programów Unixowych z oferowanym przez nas interfejsem terminali, został on zaprojektowany zgodnie ze standardem POSIX dotyczącym tej kwestii. Standard reguluje jak ma przebiegać wejście, wyjście oraz zmiana ustawień terminala.

Zmiana ustawień.

Najważniejsze ustawienia terminala przechowywane są w strukturze `termios`. Zawiera ona następujące pola:

tcflag_t c_iflag	Flagi konfigurujące zachowanie wejścia
tcflag_t c_oflag	Flagi konfigurujące zachowanie wyjścia
tcflag_t c_lflag	Ogólne flagi ustawiające tryb pracy terminala
tcflag_t c_cflag	Flagi związane z obsługą połączenia
cc_t c_cc[NCCS]	Znaki specjalne

Przeznaczenie poszczególnych pól zostanie przybliżone w kolejnych podrozdziałach. Biblioteka C udostępnia funkcje do zapisu i odczytu aktualnych ustawień terminala - są to odpowiednio `tcsetattr` i `tcgetattr`. Funkcje to zostały zaimplementowane w oparciu o wywołanie systemowe `ioctl`.

Otwieranie terminala, uprawnienia procesów.

Terminal otwierany jest jak zwykły plik, przy pomocy wywołania systemowego `open`. Aby umożliwić wielu programom jednoczesne korzystanie z jednego terminala, oraz umożliwić kontrolę zadań w powłokach takich jak `ash`, wprowadzona została dodatkowa organizacja procesów.

Każdy proces może posiadać swój powiązany terminal kontrolujący. Ogólnie rzecz biorąc, może on korzystać tylko z tego terminala. Procesy zostały podzielone na sesje, oraz w ramach sesji na grupy. Wszystkie procesy w ramach sesji, które mają ustawiony terminal kontrolujący, mają ustawiony ten sam terminal. Tak więc terminal jest powiązany z sesją. Terminal może być związany z tylko jedną sesją i vice versa.

W ramach sesji procesy tworzą rozłączne grupy, z których jedna może być wyszczególniona jako grupa procesów pierwszoplanowych terminala. Procesy z tej grupy jako jedyne mają dostęp do wejścia z terminala. Co do zapisu na terminal, możliwy jest on także spoza grupy procesów pierwszoplanowych, jednak to wymaga dodatkowych środków (w postaci blokowania lub ignorowania sygnału `SIGTTOU`).

Sesja oraz grupa procesów posiadają swój identyfikator, równy identyfikatorowi procesu, który jako pierwszy do danego bytu należał. Proces taki zwany jest odpowiednio liderem sesji i liderem grupy. Do tworzenia nowej sesji wykorzystuje się funkcję `setsid`. Terminal kontrolujący procesu ustawiany jest automatycznie, w momencie otwierania go, o ile proces otwierający nie posiada już terminala kontrolującego, jest liderem sesji, oraz terminal ten nie jest jeszcze związany z żadną sesją. Terminal kontrolujący, sesję, oraz grupę proces dziedziczy po ojcu w wywołaniu `fork`.

Pobieranie i modyfikacja grupy procesu realizowane są poprzez `getpgid` i `setpgid`. Wybór grupy procesów pierwszoplanowych terminalu następuje poprzez wywołanie `ioctl` na deskryptorze pliku terminala, z poleceniem `TIOCSPGRP` (zobacz także `tcsetpgrp` i `tcgetpgrp`).

Zapisywanie do terminala.

Programy przekazują dane na wyjście terminala za pomocą jednej z funkcji biblioteki C, zazwyczaj z rodziny `printf`. Funkcja `printf` wywołuje poprzez przerwanie systemową funkcję `sc_write`. Ta z kolei, poprzez vnode związany z deskryptorem pliku przekazuje bufor z danymi użytkownika do funkcji obsługi `tty_write` urządzenia znakowego związanego z terminalem.

Zanim będzie mógł nastąpić faktyczny zapis danych przy pomocy funkcji zarejestrowanej w procedurze `tty_create`, funkcja `tty_write` weryfikuje, czy piszący proces ma do tego prawo, oraz w zależności od ustawień wykonuje końcowe przekształcenia na danych użytkownika. Wykonywane przekształcenia uzależnione są od wartości `c_oflag`. Możliwe operacje to miedzy innymi zamiana znaków `CR` (powrót karetki) na znaki `NL` (nowej linii) i zamiana znaków `NL` na parę `CR-NL`.

Odczyt z terminala.

Urządzenie wejściowe otrzymawszy dane, przekazuje je do bufora powiązanego terminala poprzez funkcję `tty_input`. Użytkownik uzyskuje dostęp do tych danych przy pomocy funkcji bibliotecznych takich jak `scanf`, korzystających z wywołania systemowego `read`. Dane uzyskane w ten sposób to ciąg znaków ASCII.

Wyobraźmy sobie następującą sytuację: program pyta użytkownika, o podanie imienia, ten jednak, w połowie wpisywanego tekstu popełnił błąd i skorygował go przy użyciu klawisza backspace. Program jest zainteresowany jedynie poprawionym wpisem, a nie ciągiem znaków zawierających błędne dane oraz kod ASCII klawisza backspace. Jest to na tyle częsta sytuacja, że schemat obsługi wejścia, w którym sterownik dba o obsługę zmian w ramach jednej lini wejścia został uwzględniony w standardzie POSIX jako element interfejsu terminali. Oczywiście niektóre programy chcą znać kody wszystkich naciskanych klawiszy, bez konieczności czekania na znak nowej lini, tak więc i ta sytuacja musi być obsługiwana. Pierwszy tryb według POSIX zwiemy trybem kanoniczym, drugi surowym. Tryb pracy terminala uzależniony jest od obecności flagi `ICANON` w polu `c_lflag` struktury opisującej konfigurację terminala.

Tryb kanoniczny.

Domyślnie terminal znajduje się w trybie kanonicznym. W trybie tym rozpoznawane są znaki specjalne, ustalone w polach tabeli `c_cc`. Należą do nich EOF, EOL, ERASE, INTR, KILL, QUIT, START, STOP, SUSP i TIME. Sterownik dokonuje edycji linii w przypadku rozpoznania znaku ERASE (usunięcie ostatniego znaku) bądź KILL (usunięcie całej lini). Jeżeli pole `c_lflag` zawiera flagę `ISIG`, wystąpienie znaków INTR, QUIT oraz SUSP powoduje wysłanie do grupy procesów pierwszoplanowych odpowiednio sygnału `SIGINT`, `SIGQUIT`, `SIGTSTP`.

Procedura `read` zwraca wynik, tylko w przypadku gdy w buforze wejściowym terminala istnieje linia zakończona znakiem NL, EOF bądź EOL. Jeżeli linia nie jest jeszcze gotowa, proces zasypia w oczekiwaniu na nią. Zwrócony bufor zawiera co najwyżej jedną linię z wejścia.

W trybie kanonicznym na wejściu wykonywane jest wstępne przetwarzanie, według wartości `c_iflag`. Możliwe operacje to m.in. ignorowanie znaków `CR`, zamiana znaku `CR` na znak `NL` i na odwrót.

Tryb surowy.

W trybie surowym znaki nie są dodatkowo przetwarzane. Poprzez ustawienie pól `c_cc[VMIN]` i `c_cc[VTIME]` użytkownik może kontrolować minimalną ilość znaków, jaka zostanie zwrócona przez `read`, oraz czas, jaki procedura ma czekać na kolejny znak (bądź całość wejścia - przy `MIN`= 0).

3.6.2 Konsola.

Jedynym w tej chwili zaimplementowanym w Impali terminalem jest konsola - zestaw złożony z klawiatury i wyświetlacza podłączonego do karty graficznej. Urządzenia te są widoczne w systemie jako kilka osobnych wirtualnych konsoli, o plikach `/dev/ttyvX`, gdzie `X` jest numerem urządzenia. Przełączanie pomiędzy tymi konsolami następuje po naciśnięciu odpowiedniego klawisza funkcyjnego `Fx`.

Obsługa klawiatury.

Niskopoziomowa obsługa klawiatury przebiega następująco:

- Każde wciśnięcie i zwolnienie klawisza powoduje wygenerowanie przerwania klawiatury.
- Procedura obsługi tego przerwania rozpoznaje rodzaj zdarzenia, odczytując jego kod - "scancode" - z odpowiedniego portu układu kontrolera klawiatury (i8042).
- Na podstawie scancode wyznaczany jest unikalny kod klawisza - "keycode".
- Na podstawie kodu klawisza, przechowywanych w sterowniku informacji o naciśniętych klawiszach specjalnych takich jak shift, alt i ctrl oraz "keymapy" odwzorowującej te informacje w znak (ewentualnie ciąg znaków) ASCII, wyznaczany jest wynik naciśnięcia klawisza w postaci, jakiej oczekuje użytkownik.
- Wynik z poprzedniego kroku przekazywany jest do aktywnej wirtualnej konsoli oraz powiązanego z nią terminala poprzez procedurę `vcons_input_[char/string]`.

Obsługa karty graficznej.

Niskopoziomowa komunikacja z kartą graficzną w Impali polega na:

- Początkowym zainicjalizowaniu karty graficznej, realizowanym poprzez odpowiednią sekwencję zapisów i odczytów z portów karty. W kroku tym ustawiany jest m.in. kursor sprzętowy. Ponieważ nie ma możliwości wyłączenia migania kursora sprzętowego, zostaje on ukryty. Kursor widoczny na ekranie jest emulowany programowo.
- Bufor ramki karty graficznej jest odwzorowany w pamięć pod adresem fizycznym `0xb8000`. Jako, że w Impali niskie adresy są zarezerwowane na przestrzeń użytkownika, ten adres fizyczny jest z kolei odwzorowywany w stercie jądra. Wyświetlenie znaku w pewnym miejscu ekranu polega na zapisaniu go w odpowiednim miejscu pamięci. Atrybuty znaków takie jak ich kolor dla znaku z komórki pamięci x ustawiane są w komórce $x + 1$.

ESC[2J	Czyści cały ekran
ESC3C	Przesuwa kursor o 3 pozycje w prawo
ESC[1;5H	Ustawia kursor w piątej kolumnie pierwszego wiersza
ESC[5;7;1m	Włącza mruganie, pogrubienie oraz zamienia kolor tła z kolorem znaku
ESC[6n	Żąda informacji o aktualnym położeniu kursora

Tabela 3.1: Przykładowe sekwencje sterujące VT100. ESC oznacza znak \033

3.6.3 Emulacja terminala.

Aby umożliwić programom użytkownika bardziej zaawansowaną kontrolę nad zawartością ekranu, wirtualne konsole udają, że są fizycznym terminalem. Konsola w Impali emuluje VT100, popularny terminal stworzony przez firmę Digital Equipment Corporation. Emulacja ta polega na rozpoznawaniu sekwencji sterujących tego terminala oraz odpowiednim reagowaniu na nie. Podobnie, wejście z klawiatury przedstawiane jest użytkownikowi w postaci, jaką by otrzymał pracując na terminalu VT100 i korzystając z jego klawiatury.

Dzięki takim działaniom progamy mają możliwość m.in. ustawiać pozycję kursora na ekranie, przewijać ekran, kasować jego zawartość, pobierać informacje o położeniu kursora i o wspieranych funkcjonalnościach, zmieniać tryb pracy terminala i atrybuty wypisywanych znaków.

3.6.4 Termcap.

Jako, że istnieje wiele terminali, różniących się sekwencjami sterującymi, rozmiarem ekranu i innymi szczegółami ich działania, wynikła potrzeba udostępniania procesowi informacji o terminalu na jakim aktualnie działa. Identyfikator tego terminala przechowywany jest w zmiennej środowiskowej `TERM`. Dostęp do informacji o konkretnych sekwencjach sterujących możliwy jest poprzez następujące funkcje biblioteki C: `tgetstr`, `tgetnum`, `tgetflag`. Zanim jednak będziemy mogli skorzystać z tych funkcji, konieczne jest załadowanie informacje o wybranym terminalu za pomocą funkcji `tgetent`. Informacje uzyskane z wymienionych funkcji mogą służyć jako wejście do procedury `tgoto`, wypełniającej sekwencję sterującą o wymagane argumenty, oraz procedury `tputs` wysyłającej sekwencję do terminala.

Baza danych dla termcap typowo przechowywana w pliku, jest u nas dostępna jako wartość zmiennej środowiskowej `TERMCAP`.

3.7 Wirtualny system plików (VFS).

3.7.1 Pliki od strony użytkownika.

Interfejs plików jest jednym z najważniejszych interfejsów jakie programy mają do dyspozycji. W systemach Unixowych pliki reprezentują całą gamę bytów:

- Pliki w standardowym rozumieniu, jako pewnej długości ciąg bajtów. Jest to rodzaj pliku, z jakim użytkownik najczęściej ma kontakt, korzystając z sytemów plików takich jak FAT, NTFS, UFS, ext3 itd.
- Urządzenia - zarówno fizycznie istniejące w komputerze, jak i wirtualne.
- Potoki FIFO.

- Pliki jako wirtualne byty służące do przekazywania różnych informacji z jądra systemu.

Niezależnie od tego, co kryje się pod plikiem, od strony użytkownika posiada on jednolity interfejs. Najważniejszymi funkcjami wchodzącymi w jego skład są: `open`, `close`, `read`, `write`, `ioctl` `fcntl`, `lseek` i `fstat`. Otwarty plik w programie jest identyfikowany przy pomocy deskryptora pliku - zazwyczaj małej liczby nieujemnej. Jądro systemu posiada związaną z każdym procesem tabelę, służącą do tłumaczenia deskryptora pliku na strukturę `file_t`. Wiele deskryptorów plików może wskazywać na tą samą strukturę. Oto, jak wygląda `file_t`:

vnode_t	*f_vnode	Wskaźnik na v-węzeł związany z plikiem
off_t	f_offset	Aktualna pozycja kursora w pliku
int	f_refcnt	Licznik referencji dla tej struktury
int	f_flags	Flagi, początkowo ustawione przez open

Tak więc wszystkie deskryptory wskazujące na ten sam plik dzielą informacje takie jak aktualna pozycja kursora oraz flagi pliku. Do pobrania struktury `file_t` na podstawie deskryptora służy funkcja `f_get`. Tablica deskryptorów plików została zrealizowana jako lista kawałków tablicy o pojemności 32 wpisów.

3.7.2 Pliki od strony jądra.

Ważnym zadaniem jądra systemu operacyjnego jest stworzenie takich warunków, w których pliki pochodzące z różnych systemów plików mogły być obsługiwane przez pojedynczy, uniwersalny interfejs. Takie właśnie zadanie spełnia VFS - wirtualny system plików. Zarówno pojęcie pliku jak i systemu plików zostało ujęte w ramy ściśle zdefiniowanych struktur jądra, ukrywających prawdziwą implementację powiązanych z nimi operacji. Pliki są reprezentowane przez strukturę `vnode_t`, natomiast systemy plików poprzez `vfs_t`. Projekt VFS-a zawarty w Impali bazowany był na tym z SVR4.

V-węzły

Aby móc zrozumieć funkcjonowanie wirtualnego systemu plików, ważną jest znajomość struktury, do której odnosi się większość operacji. Oto jak wygląda v-węzeł:

int	v_type	typ v-węzła
int	v_flags	flagi v-węzła
int	v_refcnt	licznik referencji
vfs_t	*v_vfs_mounted_here	system plików tutaj zamontowany
vfs_t	*v_vfs	system plików tego vnode
vnode_ops_t	*v_ops	wskaźnik do `vnode_ops` z tego fs
devd_t	*v_dev	urządzenie, jeśli to v-węzeł urządzenia
void	*v_private	prywatne dane systemu plików, np. i-węzeł
mutex_t	v_mtx	blokada do synchronizacji

Impala posiada następujące typy v-węzłów:

VNODE_TYPE_REG	-	zwykły plik
VNODE_TYPE_DIR	-	plik reprezentujący katalog
VNODE_TYPE_DEV	-	plik reprezentujący urządzenie
VNODE_TYPE_LNK	-	plik będący dowiązaniem symbolicznym
VNODE_TYPE_FIF	-	plik reprezentujący potok

Operacje możliwe do wykonania na v-węźle ukryte są w strukturze `vnode_ops_t` dostarczanej dla każdego v-węzła przez związany z nim system plików. Struktura ta zawiera wskaźniki do funkcji wykonujących wszystkie przewidziane przez nas operacje, jakie są możliwe do wykonania na pliku. Szczegółowo została ona przedstawiona w tabeli 3.2.

Dla wygody korzystania z v-węzłów dla każdej z operacji wprowadzono odpowiednie makro `VOP_XXX(v, ...)` wykonujące operację `XXX` na v-węźle `v`. Dla dalszego ułatwienia pracy z v-węzłami, wprowadzono następujące ogólne procedury:

- `vnode_opendev` - otwiera vnode urządzenia o podanej nazwie
- `vnode_rdwr`, `vnode_urdwr` - wykonują operacje odczytu i zapisu do danego v-węzła
- `vnode_stat` - pobiera różne informacje o pliku
- `vnode_isatty` - sprawdza czy v-węzeł związany jest z terminalem
- `vnode_access_ok` - weryfikuje uprawnienia danego procesu do dostępu do pliku z zamiarem wykonania podanych operacji
- `vnode_alloc` - przydziela nową, pustą strukturę v-węzła
- `vrele` - zwalnia daną referencję do v-węzła
- `vref` - tworzy nową referencję do v-węzła

W kwestii zliczania referencji, przyjęta przez nas strategia zakłada, że każda funkcja zwracająca w wyniku v-węzeł, musi zwrócić go ze zwiększoną już liczbą referencji (tj. wykonanym `vref`). Dzięki temu, eliminujemy jedną z możliwych sytuacji wyścigu. Procedura wywołująca staje się w ten sposób właścicielem zwróconego wskaźnika do v-węzła. Jeżeli nie będzie już więcej z niego korzystała, musi wykonać na nim `vrele`.

Kolejnym ustaleniem, jest to, że funkcja, która dostała v-węzeł jako argument, może bez przeszkód z niego korzystać, o ile nie zachowuje tego wskaźnika na później. Każde klonowanie wskaźnika wymaga wywołania funkcji `vref`. Wyraźnie oznaczone funkcje mogą jednak przejmować prawo własności do referencji od strony wywołującej. Jest to wykorzystywane dla wygody, w sytuacjach, gdy po wykonaniu procedury dalszy dostęp do obiektu nie jest zazwyczaj potrzebny.

Podobna strategia została zastosowana w odniesieniu do struktury `file_t`. Odpowiednie funkcje mają nazwy `frele` i `fref`.

W przypadku zwalniania ostatniej referencji do v-węzła, wykonywana jest na nim operacja `VOP_INACTIVE`, mająca na celu poinformowanie i-węzła (zależnej od sytemu plików części v-węzła) o tej sytuacji. Umożliwi to wykonanie końcowych czynności, takich jak zwolnienie i-węzła.

3.7.3 System plików.

Struktura reprezentująca zamontowane systemy plików jest analogiczna do struktury v-węzła, jednakże związane z nią operacje są mniej liczne:

vnode_open_t	*vop_open	Otwieranie pliku
vnode_create_t	*vop_create	Tworzenie nowego pliku
vnode_close_t	*vop_close	Zamykanie pliku
vnode_read_t	*vop_read	Odczytywanie z pliku
vnode_write_t	*vop_write	Zapisywanie do pliku
vnode_ioctl_t	*vop_ioctl	Wykonywanie dodatkowych operacji
vnode_seek_t	*vop_seek	Sprawdzanie, czy dana pozycja jest prawidłowa
vnode_truncate_t	*vop_truncate	Zmiana rozmiaru pliku
vnode_getattr_t	*vop_getattr	Pobieranie różnych atrybutów pliku
vnode_setattr_t	*vop_setattr	Ustawianie atrybutów pliku
vnode_lookup_t	*vop_lookup	Poszukiwanie v-węzła o podanej nazwie, począwszy od danego v-węzła katalogu
vnode_mkdir_t	*vop_mkdir	Tworzenie nowego katalogu
vnode_getdents_t	*vop_getdents	Pobieranie zawartości katalogu
vnode_readlink_t	*vop_readlink	Odczytywanie nazwy pliku, na który wskazuje dowiązanie symboliczne
vnode_symlink_t	*vop_symlink	Tworzenie dowiązania symbolicznego
vnode_access_t	*vop_access	Sprawdzanie praw dostępu do pliku
vnode_sync_t	*vop_sync	Synchronizowanie stanu pliku w pamięci ze stanem na trwałym nośniku
vnode_inactive_t	*vop_inactive	Informowanie pliku, że ostatnia referencja do tego v-węzła jest właśnie usuwana
vnode_lock_t	*vop_lock	Blokowanie pliku
vnode_unlock_t	*vop_unlock	Odblokowywanie pliku
vnode_rmdir_t	*vop_rmdir	Usuwanie katalogu
vnode_link_t	*vop_link	Podwiązywanie istniejącego pliku pod nową nazwą
vnode_unlink_t	*vop_unlink	Usuwanie wpisu z katalogu

Tabela 3.2: Struktura v-węzła - `vnode_t`

vfs_ops_t	*vfs_ops	definicje operacji związanych z tym systemem plików
vnode_t	*vfs_mpoint	v-węzeł który przykryliśmy montując ten s. plików
devd_t	*vfs_mdev	urządzenie, używane przez ten s. plików
void	*vfs_private	prywatne dane systemu plików
vfs_conf_t	*vfs_conf	struktura definiująca typ zamontowanego systemy plików
list_node_t	L_mountlist	węzeł z listy zamontowanych fs

Operacje, możliwe do wykonania na systemie plików to montowanie systemu plików, odmontowywanie go, synchronizowanie jego stanu z pamięcią trwałą, oraz pobranie v-węzła katalogu głównego systemu plików. Podobnie jak przy v-węzłach, dla wygody korzystania z tych operacji utworzone zostały makra, o nazwach `VFS_XXX`. W chwili obecnej żaden z systemów plików nie posiada zaimplementowanej obsługi odmontowywania systemu plików. Do montowania systemu plików istnieje dodatkowa funkcja `vfs_mount` korzystająca z makra `VFS_MOUNT` i wykonująca wszelkie niezbędne czynności potrzebne do zamontowania systemu plików na podstawie v-węzła punktu montowania, informacji o nazwie systemu plików oraz strukturze urządzenia które należy zamontować.

Zaimplementowane w Impali systemy plików to:

- MFS - pamięciowy system plików, zamontowany jako korzeń drzewa katalogów.
- FAT12 - system plików potrzebny do odczytania danych z dyskietki. Dyskietka jest domyślnie montowana w systemie plików pod katalogiem `/mnt/fd0/`.
- devfs - system pełniący rolę pojemnika na pliki urządzeń, a także pośrednika między interfejsem plików a interfejsem urządzeń. Domyślnie zamontowany w `/dev/`.
- fifofs - system plików na użytek plików potoków. Nie obsługuje on funkcji montowania.

Wszystkie te systemy plików kompilowane są do pojedynczej biblioteki `libfs.a` udostępniającej tablicę funkcji inicjalizacji poszczególnych systemów plików `fstab`. Funkcje zawarte w tej tabeli są automatycznie wykonywane podczas inicjalizacji wirtualnego systemu plików.

Jedną z ważniejszych funkcji, udostępnionych przez podsystem VFS, jest `vfs_lookup`. Służy ona do zlokalizowania v-węzła pliku na podstawie ścieżki dostępu. Implementuje ona standardowy schemat rozwiązywania ścieżek.

3.8 Wielozadaniowość.

Zadania procesora są reprezentowane przez wątki (`thread_t`). Każdy z wątków dostaje kwant czasu na wykorzystanie procesora, dzięki czemu użytkownik może odnieść wrażenie, że uruchomione przez niego zadania działają jednocześnie. Za przydział czasu procesora jest odpowiedzialny program planisty omówiony w 3.8.5.

Wątek posiada w sobie między innymi następujące informacje:

- kontekst, służący do zachowywania jego stanu
- proces do, którego przynależy wątek
- adres procedury wejściowej, od której rozpoczyna się działanie zadania
- adres stosu użytkownika i jego rozmiar

- adres stosu alternatywnego i jego rozmiar
- przestrzeń adresowa, w której działa to zadanie (wspólna dla wszystkich wątków w obrębie tego samego procesu)

Wątki procesora wykonujące kod użytkownika potrzebują dwóch stosów. Pierwszy stos jest przeznaczony do standardowego użytku. Stos alternatywny natomiast jest wykorzystywany wtedy, kiedy procesor wywłaszcza zadanie obsługując przerwanie. Wątki wykonujące kod systemu utożsamiają ze sobą te dwa stosy.

Stosy użytkownika są jego prywatnymi stosami, tzn. istniejącymi jedynie w jego przestrzeni adresowej, natomiast stosy alternatywne znajdują się w przestrzeni jądra, dzięki czemu są zawsze dostępne dla systemu.

Adres procedury wejściowej jest wykorzystywany jeżeli dany wątek nie miał dotąd ani razu przydzielonego czasu procesora - np. został dopiero utworzony. W takim przypadku jego działanie rozpoczyna się od adresu procedury wejściowej, w przeciwnym od adresu zapisanego w kontekście.

Kontekst procesora zawiera w sobie takie informacje jak ramka przerwania, którą zapisuje procesor na stos alternatywny przy wywłaszczeniu przez przerwanie oraz pomocniczy schowek na rejestry procesora.

3.8.1 Zmienianie kontekstu procesora.

Mechanizm zmiany kontekstu procesora jest oparty o mechanizm obsługi przerwań przez procesor. Przed rozpoczęciem omawiania mechanizmu zmian kontekstu wyszczególnimy kilka faktów, związanych z pracą procesora.

- Procesor podczas wywoływania procedury kładzie na stos adres aktualnie wykonywanej instrukcji, tak aby po zakończeniu procedury mógł wrócić do kodu ją wywołującego, zapisane w ten sposób adresy na stosie tworzą „ślad wywołań".
- Stos w procesorze jest definiowany przez adres w rejestrze `esp`, a operacje na stosie pobierają i zapisują dane pod tym adresem odpowiednio go modyfikując.

Pod obsługę zegara systemowego, którego przerwanie jest generowane z częstotliwością 100Hz, jest podpięty program planisty, który jeżeli zdecyduje się zmienić kontekst procesora, tzn. wykonywany przez procesor wątek, to przekazuje sterownie do procedury `thread_switch` przekazując jej deskryptor aktualnie wykonywanego wątku (`curthread`) oraz wątku na jaki ma nastąpić zmiana.

Wspomniana procedura zapisuje aktualne rejestry procesora do pomocniczego schowka w kontekście w deskryptorze aktualnie wykonywanego wątku, a następnie wczytuje zachowane wcześniej rejestry ze schowka drugiego wątku.

Ponieważ w rejestrach procesora są zapisane adresy lokalizujące stos to powyższa operacja powoduje „podmianę" aktualnie wykonywanego stosu procesora na ten, jaki był w momencie zapisywania rejestrów drugiego wątku (to znaczy dokładnie wtedy, kiedy planista uruchomił `thread_switch` podczas pracy drugiego wątku). Ponieważ na stosie jest zapisany ślad wywołań, to po podmianie stosu procesor zachowa się dokładnie tak, jak by się zachował wykonując drugi wątek po zachowaniu jego rejestrów. Bardzo podobny mechanizm istnieje w języku C w procedurach `setjmp` i `longjmp`. Kończąc po kolei procedury procesor wróci

do tej, którą wywołał procesor do obsługi przerwania zegara podczas pracy drugiego wątku, a jej zakończenie przywróci stan procesora z zachowanej ramki przerwania, zawierającej stan drugiego wątku. Ta operacja jest prawidłowa, ponieważ ramki przerwań są zapisywane na stosach alternatywnych, które są dostępne w przestrzeni jądra.

Jeżeli drugi wątek nie miał dotąd przydzielonego czasu procesora, to procedura zmiany kontekstu nie miała okazji zachować jego rejestrów w schowku, a procesor zapisać ramkę przerwania na jego stosie alternatywnym. W takim wypadku uruchamiana jest procedura `thread_resume`, która ręcznie buduje ramkę stosu, aby oszukać procesor, że powraca do obsługi zadania, które wcześniej wywłaszczył. Jako adres wywłaszczonej instrukcji jest podawana procedura wejściowa.

W schowku na rejestry procesora jest zapisywany również rejestr kontrolny procesora `cr3` zawierający adres katalogu stron, tak więc wczytując adresy z tego schowka następuje również podmiana wirtualnej przestrzeni adresowej użytkownika.

3.8.2 Rozwidlanie procesów.

Jedyną drogą stworzenia nowego procesu w systemie UNIX jest rozwidlenie istniejącego procesu. Zgodnie z semantyką nadaną przez standard nowy proces dziedziczy po rodzicu kopię jego deskryptorów plików, przestrzeni adresowej oraz środowiska.

Jeżeli w procesie rodzica działało wiele wątków to nowy zawiera tylko jeden, który jest kopią wątku zlecającego systemowi zadanie rozwidlenia.

Za obsługę rozwidlenia odpowiedzialna jest procedura `proc_fork`. Po stworzeniu nowego wątku w systemie kopiowany jest stos alternatywny oraz kontekst wątku wywołującego. Skopiowany stos zawiera ramkę przerwania z zachowanym stanem wątku wywołującego, co umożliwia oszukania procesora w podobny sposób jak w procedurze `thread_resume` i uruchomienie nowy wątek z pożądanym stanem.

3.8.3 Synchronizacja.

Ważnym elementem implementacji mechanizmów synchronizacji są niepodzielne instrukcje procesora, tzn takie których nie można wywłaszczyć w trakcie ich wykonywania. Instrukcje porównania oraz zapisu generowane przez kompilator spełniają tę właściwość. Moduł obsługi platformy sprzętowej musi dostarczyć jeszcze procedurę `atomic_change_int` modyfikującą komórkę pamięci i zwracającą jej starą wartość w sposób niepodzielny. Gdyby procedura nie spełniała żądanej własności, to pomiędzy pobraniem starej wartości, a zapisaniem nowej mogłoby zostać wykonane wywłaszczenie, podczas którego wartość komórki pamięci została zmodyfikowana. W takim wypadku wszelkie decyzje podjęte na podstawie starej wartości, zwróconej przez procedurę, mogłyby być błędne.

Najprostszym mechanizmem synchronizacji są wirujące zamki, wykorzystujące wprost niepodzielne instrukcje procesora. Taki zamek jest opisywany przez jedną komórkę pamięci przyjmującą wartość `SPINLOCK_LOCK` lub `SPINLOCK_UNLOCK` odpowiednio do stanu blokady.

Operacja zamknięcia używa wspomnianej wcześniej procedury aby ustawić stan blokady na `SPINLOCK_LOCK`. Jeżeli stara wartość komórki pamięci była równa `SPINLOCK_UNLOCK` to z niepodzielności użytej procedury wynika, że nikt inny nie mógł odczytać ani zmodyfikować jej wartości i udało się pomyślnie zmienić stan blokady. Jeżeli stara wartość komórki pamięci była równa `SPINLOCK_LOCK`, to również z niepodzielności tej procedury wynika, to

że blokada została zamknięta przez kogoś innego. W takim wypadku procedura zamknięcia zamka kręci się w miejscu (wiruje), dopóki nie uda się jej zamknąć blokady.

Ten sposób synchronizacji nie zapewnia, że wątki nie będą głodzone (oczekiwać w nieskończoność), dodatkowo oczekiwanie na zwolnienie blokady jest aktywne co marnuje czas pracy procesora.

Innym rodzajem blokady są zamki `mutex_t` (*mutual exclude*), których klienci są usypiani na czas założonej blokady. Zamki wewnętrznie używają kolejki FIFO do kontrolowania kolejności budzenia wątków, chcących wejść do chronionej sekcji krytycznej - ta strategia eliminuje problem głodzenia. Wewnętrzna struktura danych tej blokady jest chroniona za pomocą wirujących zamków. Idea tych zamków odróżnia je od semaforów tym, że z tą blokadą jest związana informacja o właścicielu, tzn wątku który zamknął blokadę - w przeciwieństwie do semaforów jedynie on ma prawo do odblokowania.

Zmienna warunkowa jest mechanizmem ściśle powiązanym z mechanizmem blokad. Jej zadanie to umożliwienie biernego oczekiwania na zdarzenie. Jest on sprzężony z wybraną blokadą w celu wykonywania niepodzielnej operacji jej zwolnienia oraz uśpienia wątku.

Mechanizm jest standardowo wykorzystywany do implementacji kolejek, sama blokada może jedynie chronić sekcje krytyczne procedur wysłania i odebrania wiadomości do kolejki, lecz uniemożliwia oczekiwanie w przypadku braku wiadomości do odebrania. Mechanizm wykorzystuje się tak, że gdy nie ma wiadomości do odebrania to klient jest usypiany, a blokada zwalniana. Dzięki niepodzielności nie jest możliwa sytuacja, w której pomiędzy uśpieniem a zwolnieniem blokady ktoś dostarczy wiadomość do kolejki.

Dostarczenie zdarzenia budzącego jest możliwe do jednego wątku lub do wszystkich naraz, może je dostarczyć jedynie właściciel blokady sprzężonej z zmienną warunkową. Zdarzenie jest dostarczane przy wyjściu z sekcji krytycznej przez klienta je zgłaszającego. Budzony wątek dostaje od razu zamknięty zamek, dzięki czemu powraca on do swojej sekcji krytycznej.

Nasza implementacja zamków `mutex_t` zawiera w sobie mechanizm zmiennej warunkowej.

3.8.4 Biblioteka wątków użytkownika.

Istnieją trzy modele realizacji wątków po stronie użytkownika:

- M:1 - wiele wątków użytkownika jest zarządzanych przez niego samego, a jądro systemu operacyjnego widzi wszystkie jako jeden wątek (nie wie o ich istnieniu).
- M:N - wątki użytkownika są łączone w grupy, które są widziane jako pojedyncze wątki dla jądra.
- 1:1 - każdy wątek użytkownika jest widziany przez jądro.

W naszym systemie wybraliśmy trzeci model. Do zarządzania wątkami dostarczone są odpowiednie wywołania systemowe pozwalające tworzyć nowe wątki, oczekiwać na ich zakończenie oraz zarządzać blokadami.

Identyfikatorami wątków jakimi posługuje się biblioteka są adresy deskryptorów wątków w jądrze, ponieważ jądro nie może ufać dostarczonym adresom od użytkownika (podczas wskazywania identyfikatorem, którego wątku tyczy się dana operacja) każdy proces posiada listę swoich wątków. Identyfikator wątku jest weryfikowany poprzez test bycia obecnym na tej liście.

Blokadami tworzonymi przez użytkownika są blokady `mutex_t` w jądrze systemu. Pomysł z identyfikatorami i badaniem ich poprawności jest ten sam, co wyżej omówiony.

Standard *POSIX Threads* definiuje procedury i semantykę wątków użytkownika. W naszym systemie zaimplementowaliśmy podzbiór tego standardu pozwalający na prostym zarządzaniu wątkami, tworzeniu blokad oraz zmiennych warunkowych.

Wewnętrzne struktury biblioteki są chronione przez wirujące zamki.

3.8.5 Szeregowanie zadań.

Zaimplementowany w systemie algorytm szeregowania zadań został oparty na algorytmach planistów zastosowanych w systemach 4.3BSD oraz SVR4. Dokładny opis tych algorytmów znajduje się w [4].

Omówimy tutaj jedynie ogólny zarys planisty oraz różnice w stosunku do pierwowzoru. Planista przydziela procesom priorytet na podstawie ich ostatniego zachowania w systemie (tzn. zużycia czasu procesora oraz innych statystyk) oraz poziomu uprzejmości (nice). Priorytet jest liczbą z zakresu od 0 do 127, procesy o mniejszej wartości priorytetu są traktowane jako ważniejsze, można więc rozumieć tę wartość jako karę nakładaną na proces. Wszystkie wątki w ramach procesu traktowane są jednakowo i wszystkie posiadają priorytet równy priorytetowi procesu. Przestrzeń priorytetów podzielona jest na 32 grupy po 4 wartości w każdej grupie, kolejka Q_i odpowiada za zbiór priorytetów $i, i+1, i+2, i+3$. Procesy przynależą do odpowiednich kolejek, za rozsiewanie ich odpowiada okresowo (co `SCHED_RESCHEDULE` kwantów czasu) uruchamiana funkcja `__resched`, która dokonuje podziału na podstawie listy `run_queue`, w której znajdują się wątki gotowe do uruchomienia, po wykonaniu tej funkcji następuje wybranie niepustej kolejki zawierającej najniższe priorytety i w ramach tej kolejki planista realizuje podejście zwane algorytmem karuzelowym. Algorytm karuzelowy jest prostym sposobem szeregowania zadań polegającym na organizacji procesów w listę cykliczną, a następnie przydzielanie każdemu z wątków kolejno jednakowego kwantu czasu. Procesy znajdujące się poza tą kolejką nie są brane pod uwagę aż do następnego wywołania funkcji `__resched`, która realizuje także uaktualnianie priorytetu procesów wg. wzoru:

$$\text{priorytet} = 2 \cdot \text{poziom_uprzejmości} + \text{wykorzystanie_procesora}/2$$

Wszystkie opisywane przez wzór wartości przechowywane są w bloku kontrolnym procesu. Algorytm umożliwia preferowanie pewnych zadań przez użytkownika systemu za pomocą poziomu uprzejmości, oraz uwzględnia wykorzystanie procesora przez poszczególne procesy, co umożliwia wybór zadań interakcyjnych (tj. edytory tekstu) przed zadaniami obliczeniowymi (tj. kompilatory).

3.9 Pomniejsze usługi jądra.

3.9.1 Wywołania systemowe.

Proces użytkownika nie posiada żadnych efektywnych praw pozwalających mu na samodzielne modyfikowanie zawartości plików i katalogów, rozmiaru używanej przez siebie pamięci czy uruchamianie innych programów - nie posiada nawet praw pozwalających mu samodzielnie zakończyć swoje działanie. Wykonanie takich operacji jest zlecane przez użytkownika do systemu za pomocą mechanizmu wywołań systemowych.

Użytkownik przekazuje parametry systemowi kładąc je na stos, tak jakby miał wywołać procedurę w języku C, lecz zamiast instrukcji wywołującej ustawia numer żądanej operacji

w rejestrze procesora `eax` i uruchamia instrukcję generującą odpowiednie przerwanie programowe[14]. Sterowanie zostaje przekazane przez procesor do procedury `ISR_syscall`, która przekazuje odpowiednio sterowanie dalej do procedury `syscall`, niezależnej już od platformy sprzętowej.

Po przywróceniu sterowania do programu w rejestrze `eax` znajduje się wynik operacji, a w rejestrze `ecx` numer błędu.

3.9.2 Sygnały.

Sygnały są podstawowym mechanizmem systemów UNIXowych służącym do powiadamiania procesów o zajściu zdarzeń systemowych czy komunikacji między wątkami. Istnieje wiele niekompatybilnych ze sobą implementacji sygnałów w systemach UNIXowych, ze względu na spory chaos na tej płaszczyźnie zdecydowaliśmy się na implementację zgodną ze standardem POSIX wzorowaną na systemach rodziny BSD.

System sygnałów jest bardzo prostym i intuicyjnym mechanizmem, działa on na zasadzie poinformowania procesu o nadejściu jednego z 32 sygnałów i podjęciu przez ten proces odpowiedniej akcji. Procesy mogą same decydować co zrobić z sygnałem, który został do nich dostarczony, możliwe jest podjęcie akcji domyślnej, zignorowanie sygnału oraz własna obsługa sygnału. Wyjątkiem są sygnały SIGSTOP i SIGKILL, które nie mogą zostać zignorowane lub zamienione własną obsługą sygnału. Akcja zapisywana jest w bloku kontrolnym procesu w strukturze `sigaction` odpowiedzialnej za przechowywanie adresu obsługi sygnału. Adres może przyjmować dwie wartości specjalne `SIG_DFL`, `SIG_IGN` odpowiadające odpowiednio domyślnej obsłudze, zignorowaniu sygnału, lub być wskaźnikiem na funkcję obsługi sygnału zdefiniowanej w programie przez programistę. Wielokrotne dostarczenie jednego sygnału do procesu zanim proces zdąży sygnał obsłużyć jest równoznaczne z jednokrotnym wysłaniem sygnału. Sygnały w systemie dostarczane są do procesu do bloku kontrolnego procesu, sygnał może obsłużyć dowolny jego wątek, ustawiając w bloku kontrolnym jego obsłużenie i następnie podejmując akcje odpowiednią dla niego na podstawie tablicy `p_sigact`. Obsługę sygnału można rozumieć jako przerwanie działania programu, przechowanie kontekstu, wykonanie procedury obsługi a następnie powrót do miejsca, w którym nastąpiło przerwanie i odtworzenie przechowanego kontekstu. Sprawdzanie czy nadszedł nowy sygnał realizowane jest przy zmianie kontekstu oraz powrocie z wywołań systemowych. Ponieważ możliwe jest przerywanie działania procedur obsługi sygnałów, to jądro udostępnia specjalny interfejs obsługi sygnałów przez procedury `sigenter` i `sigreturn`, które organizują procedury obsługi sygnałów w stos działający analogicznie do stosu przy klasycznym wywoływaniu procedur.

3.9.3 Kolejki wiadomości.

Kolejki wiadomości są jednym z trzech mechanizmów komunikacji między procesami wywodzącej się z Systemu V. Obok nich istnieją również mechanizmy pamięci dzielonej oraz semaforów, nie zostały one jednak zaimplementowane.

Użytkownik identyfikuje zasoby komunikacji między procesami za pomocą kluczy, które można porównać z plikami. Charakteryzują je takie same prawa dostępu oraz operacje tworzenia i usuwania.

[14]W nowoczesnych procesorach są dostarczone specjalne instrukcje do obsługi wywołań systemowych, zmniejszające narzut związany z obsługą przerwań, na architekturze x86 są to instrukcje `sysenter` oraz `sysexit`.

Znając klucz użytkownik może pobrać identyfikator kolejki wiadomości za pomocą której może się wymieniać wiadomościami z innymi procesami.

3.10 Szczegóły techniczne.

3.10.1 Obraz jądra.

Obraz jądra jest zapisany w formacie ELF (*Executable and Linkable Format*). Format charakteryzuje zmienna ilość sekcji w pliku, gdzie każda ma swoją nazwę. Obraz jądra zapisany w tym formacie zawiera następujące sekcje:

- `.bootstrap` - kod rozruchowy jądra.
- `.text` - kod jądra.
- `.data` - dane jądra.
- `.rodata` - dane jądra, przeznaczone tylko do odczytu.

Różne dodatkowe sekcje mogą zostać dołączone przez kompilator.

Dużym udogodnieniem tego formatu jest możliwość przypisania każdej sekcji dwóch różnych adresów ich lokalizacji. Jeden to adres wirtualny, mówiący gdzie dana sekcja znajduje się w wirtualnej przestrzeni adresowej. Drugi to adres fizyczny informujący gdzie dana sekcja będzie znajdować się w pamięci fizycznej.

Systemy operacyjne, używające tego formatu do obrazów programów, nie używają pola z adresem fizycznym, ponieważ jest on dynamicznie ustalany podczas ładowania programów. Używa go natomiast program ładujący GRUB przy wybieraniu lokalizacji w pamięci gdzie daną sekcję załadować.

3.10.2 Obrazy programów.

Obrazy programów w naszym systemie są trzymane w starym formacie plików wykonywalnych `A.out` (assembler output). Obecnie ten format jest wyparty przez format ELF, lecz jego prosta budowa czynią go idealnym kandydatem do prostego systemu operacyjnego. Obraz w tym formacie zawiera w pliku dwie sekcje.

- `.text` - kod programu.
- `.data` - dane programu.

Początek sekcji danych jest zaokrąglony do adresu pierwszej strony po sekcji tekstu, która rozpoczyna się pod adresem 0.

3.10.3 Rozruch systemu.

Skompresowany obraz jądra (`/boot/impala.gz`) jest ładowany przez program GRUB, zainstalowany w sektorze rozruchowym dyskietki.

Z działania jądra w wysokich adresach (3.1) wynika trudność techniczna przy ładowaniu systemu. Program ładujący nie przygotowuje mechanizmu stronicowania (pamięci wirtualnej), w związku z czym ogólny kod jądra jest bezużyteczny. Ponieważ format ELF umożliwia

tworzenie wielu sekcji z możliwością rozróżnienia adresu fizycznego od wirtualnego to problem został rozwiązany przez wprowadzenie specjalnej sekcji `.bootstrap`. Sekcja w odróżnieniu od standardowej sekcji `.text` jest przystosowana do pracy w środowisku, gdzie adresy wirtualne odpowiadają adresom fizycznym.

Kod rozruchowy z tej sekcji konfiguruje mechanizm stronicowania aby odwzorować kod jądra w wysokich adresach, a następnie przekazuje sterowanie kodu rozruchowego z sekcji tekstu.

W pierwszej kolejności uruchamiana jest inicjalizacja modułu platformy sprzętowej. Następnie sterowanie jest przekazywane do procedury `kmain` niezależnej od platformy.

3.10.4 System budowania.

System budowania systemu jest wzorowany na tym z systemu BSD. Idea polega napisaniu ogólnych skryptów dla programu `make`, które sparametryzowane mogą budować biblioteki, programy i jądro. Skrypty budujące system włączają w siebie te ogólne i je parametryzują w zależności od zadania jakie chcą wykonać.

Ta technika zwiększa wygodę budowania systemu, ponieważ cała obsługa jest zapachnięta do kilku plików, a w pozostałych częściach drzewa dostępne są tylko krótkie i łatwe w zarządzaniu deklaracje zadań. Poprawianie błędów tyczy się jedynie tych głównych skryptów, a efekt powiela się we wszystkich ich klientach.

Przykładowy skrypt `Makefile` budujący bibliotekę ze sterownikami jądra:

```
LIBRARY= libdev
CFLAGS=-D__KERNEL ${_K_FLAGS}

SRCS=\
    devtable.c\
    fdc/fdc.c\
    md/md.c\
    pseudo/null.c\
    pseudo/zero.c\

include ${IMPALA_MK}/lib.mk
```

Ostatnia linia w skrypcie włącza odpowiedni skrypt, a definiowane zmienne są jego parametrami.

Ponieważ nasz zespół posługiwal się różnymi wariantami programu `make` to byliśmy zmuszeni zrezygnować z wygodnych udogodnień oferowanych przez nie, z powodu braku wzajemnej kompatybilności. Źródła głównych skryptów znajdują się w katalogu `mk/`.

Bibliografia

[1] Jeff Bonwick and Sun Microsystems. The Slab Allocator: An Object-Caching Kerne Memory Allocator. In USENIX Summer, strony 87–98. 1994. http://www.usenix.org/publications/library/proceedings/bos94/full_papers/bonwick.ps

[2] Abraham Silberschatz. Peter B. Galvin. Podstawy systemów operacyjnych. Wydanie czwarte. WNT. Warszawa. 2001.

[3] Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide Part 1. Intel Corporation. 2008.

[4] UNIX: Jądro systemu. Nowe horyzonty. WNT. Warszawa. 2005

[5] Opis powłoki w standardzie POSIX. http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html

[6] Opis powłoki w podręczniku FreeBSD. http://www.freebsd.org/cgi/man.cgi?query=sh&manpath=FreeBSD+7.1-RELEASE

Dodatek A

Licencja.

- Owoc prac nad systemem Impala jest udostępniony na niniejszej licencji.

```
Impala Operating System

Copyright (C) 2009 University of Wroclaw. Department of Computer Science
   http://www.ii.uni.wroc.pl/
Copyright (C) 2009 Mateusz Kocielski, Artur Koninski, Pawel Wieczorek
   http://bitbucket.org/wieczyk/impala/
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
 notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
 notice, this list of conditions and the following disclaimer in the
 documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

Razem z systemem dystrybuowane są programy objęte innymi licencjami, więcej informacji w pliku `COPYRIGHT` na nośniku z systemem oraz w kodzie źródłowym.