AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Inżynierii Metali i Informatyki Przemysłowej

PROJEKT INŻYNIERSKI

pt.

„Implementacja podstaw systemu operacyjnego zgodnego z POSIX”

Imię i nazwisko dyplomanta:	**Grzegorz Gliński**
Kierunek studiów:	**Informatyka Stosowana**
Profil dyplomowania:	**Systemy informatyki przemysłowej**
Nr albumu:	**210616**
Opiekun:	dr inż. Krzysztof Wilk

Podpis dyplomanta: Podpis promotora:

Kraków 2011

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszy projekt inżynierski wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Kraków, dnia ……..… Podpis dyplomanta…….

Spis treści

1. Wstęp

Praca ta jest ogólną dokumentacją, opisującą stworzony wcześniej projekt. Celem projektu było stworzenie jądra systemu, które implementowało by podstawowe funkcje standardu POSIX. Zachowanie zgodności z tym standardem znacznie upraszcza późniejsze rozwijanie systemu oraz uruchamianie na nim aplikacji, które pisane były z myślą o innych systemach zgodnych z tym standardem (przykładem takiej aplikacji jest pakiet kompilatorów GNU Gcc). W chwili obecnej poza znanymi systemami operacyjnymi (jak Microsoft Windows czy Linux), istnieje wiele systemów operacyjnych mniej lub bardziej rozbudowanych pisanych głównie przez pojedyncze osoby [1]. Przy tworzeniu systemu od zera, bardzo często zakłada się że nie będzie on systemem do wszystkiego (nie będzie nadawał się jednocześnie do multimediów, na serwer oraz do obliczeń numerycznych). Pozwala to na wybór odpowiednich algorytmów (np. szeregowania zadań) oraz optymalizację kodu systemu pod konkretne zadanie jakie docelowo ma tworzony system spełniać. Takie podejście pozwala na maksymalne wykorzystanie posiadanych zasobów sprzętowych oraz minimalizację czasu wymaganego do wykonania danej operacji. System taki może znaleźć zastosowanie jako platforma do działania programów z zakresu informatyki stosowanej jak na przykład komputerowa symulacja procesów czy sztuczna inteligencja.

2. Budowa systemu operacyjnego

Systemem operacyjnym nazywamy całość oprogramowania zarządzającego sprzętem komputerowym oraz tworzącego środowisko do uruchamiania i kontroli aplikacji [2]. Nowoczesny system operacyjny musi realizować funkcję takie jak:

- planowanie przydziału czasu procesora
- zarządzanie pamięcią operacyjną oraz urządzeniami
- synchronizację oraz współdzielenie urządzeń i pamięci pomiędzy procesami
- kontrola uprawnień użytkowników oraz poszczególnych aplikacji

Nowoczesny system operacyjny możemy podzielić na kilka części:

- Jądro systemu operacyjnego
- Sterowniki urządzeń
- Powłoka

Jądro systemu operacyjnego (ang. *kernel*) jest to najważniejsza część systemu. To właśnie jądro jest pierwszym elementem ładowanym do pamięci przez program rozruchowy. Po załadowaniu musi ono zainicjować podstawowe urządzenia (jak na przykład procesor czy pamięć operacyjna), przygotować środowisko pracy dla sterowników urządzeń oraz aplikacji użytkownika. To właśnie jądro dostarcza API (ang. *Application Programming Interface*), poprzez które aplikacje oraz sterowniki komunikują się z jądrem oraz między sobą. Istnieje kilka typów jąder, podzielonych ze względu na budowę i założenia. Najczęściej spotykane są dwa typy:

- jądro monolityczne – sterowniki urządzeń pracują w przestrzeni adresowej jądra systemu. Jest to rozwiązanie wydajne i stosunkowo łatwe w implementacji. Jednak poprzez współdzielenie pamięci z jądrem, wadliwie napisany sterownik może spowodować niestabilne zachowanie całego systemu (przykładem jądra monolitycznego jest Linux)
- mikrojądro – głównym założeniem mikrojądra jest przeniesienie większości sterowników i usług do przestrzeni użytkownika. Wynikiem tego jest większa stabilność oraz bezpieczeństwo systemu. Wadą jest spadek wydajności oraz bardziej skomplikowana implementacja (przykładem mikrojądra jest GNU Hurd).

Jądro współczesnego systemu operacyjnego możemy podzielić na kilka modułów:

- kod inicjujący (wykonywany tylko raz, podczas ładowania systemu)
- manager urządzeń (odpowiedzialny za komunikację ze sterownikami oraz zarządzanie sprzętem)
- manager pamięci (odpowiedzialny za przydzielanie i zwalnianie pamięci dla jądra aplikacji)
- planista (odpowiedzialny za przydział pracy procesorów)
- system plików (odpowiedzialny za szeroko pojęte zarządzanie plikami: otwieranie, zamykanie, kasowanie, tworzenie, itp.)
- synchronizacja oraz komunikacja między procesami (odpowiedzialny za synchronizację do struktur jądra, pamięć dzieloną, itp.)

Sterowniki urządzeń są to moduły, które do różnych urządzeń tej samej klasy (np. karty sieciowe) tworzą ujednolicony interfejs z którego może korzystać jądro systemu. Dzięki stworzeniu takiego interfejsu jądro nie musi wiedzieć z jakim konkretnym urządzeniem ma do czynienia (musi znać jedynie typ urządzenia, na przykład: urządzenie magazynujące dane

czy karta graficzna) co czyni system przenośnym między różnymi konfiguracjami sprzętowymi.

Powłoka (ang. *shell*) jest to aplikacja której celem jest stworzenie interfejsu umożliwiającego komunikację użytkownika z jądrem systemu operacyjnego. Istnieje kilka typów powłok:

- tekstowe (ang. *CLI – Command line interface*), korzystanie z tego typu powłoki polega na wpisywaniu poleceń, które są odpowiednio interpretowane. Przykładem takiej powłoki jest GNU Bash,
- graficzne (ang. *GUI – Graphical user interface*), ten typ powłoki obsługiwany jest głównie za pomocą urządzenia wskazującego (na przykład myszy) a korzystanie z niego polega na wybieranie odpowiednich akcji przez klikanie na elementy interakcyjne nazywane kontrolkami. Przykładem tego typu powłoki jest Explorer z systemu Microsoft Windows.

3. Architektura systemu

3.1. Standard POSIX

POSIX (ang. *Portable Operating System Interface for Unix*) – w znaczeniu dosłownym: przenośny interfejs systemu operacyjnego Unix. Jest to nazwa rodziny standardów określonych przez IEEE (standard IEEE 1003), mającej na celu zdefiniowania interfejsu programistycznego (API), interfejsu użytkownika (polecenia systemowe) oraz cech powłoki systemu operacyjnego [3]. Nazwa POSIX została zaproponowana przez Richarda Stallmana. Standard ten dotyczy przede wszystkim wszystkich systemów operacyjnych klasy UNIX. Poza tym implementacje tego standardu występują w takich systemach operacyjnych jak Linux, Solaris, Mac OS X, QNX czy FreeBSD. W przypadku systemów operacyjnych z rodziny Microsoft Windows istnieją środowiska pozwalające na korzystanie z interfejsu programistycznego definiowanego przez POSIX (na przykład cygwin).

3.2. Jądro systemu

Jądro systemu zostało zaprojektowane oraz zaimplementowane zgodnie ze modelem monolitycznym [2] przy dodatkowym założeniem że, kod jądra nie będzie zawierał żadnych sterowników urządzeń poza niezbędnymi (jak na przykład sterownik karty graficznej VGA w trybie tekstowym). Dodatkowym celem, było podzielenie jądra na kod zależny oraz

niezależny od architektury sprzętowej. Dzięki temu możliwe jest uruchomienie systemu na różnych architekturach (w chwili obecnej na Intel IA-32 oraz AMD64).

3.3. Uruchamianie systemu na architekturze x86

Aby umożliwić ładowanie systemu przez różne programy rozruchowe, zostało ono stworzone w zgodności ze standardem multiboot [4]. Jest to realizowane poprzez umieszczenie w kodzie jądra dodatkowej struktury przechowującej dane niezbędne do załadowania systemu przez program ładujący. Podczas pisania i testowania jądra do ładowania systemu wykorzystywany był program GNU Grub.

Po załadowaniu z dysku jądra oraz kilku podstawowych modułów bootloader aktywuje chroniony tryb pracy procesora, a następnie skacze do funkcji **_start** jądra (zdefiniowanej w pliku src/sys/arch/x86/entry_32.S). Funkcja ta została napisana w języku assembler. Zadaniem tej funkcji jest przygotowanie środowiska pracy dla jądra systemu. Ustawia ona stos dla jądra, inicjuje stronnicowanie pamięci [5]. Następnie inicjowana jest konsola tekstowa, przetwarzane są informacje otrzymane od programu ładującego (między innymi mapa pamięci fizycznej). Następnie konfigurowane są parametry pacy procesora. Odbywa się to już przez funkcję w języku C o nazwie **cpu_init()**. To właśnie ta funkcja konfiguruje takie rejestry procesora [5] jak ***gdtr*** (z ang. *Global Descriptor Table Register*) czy ***idtr*** (*Interrupt Descriptor Table Register*) oraz inicjuje struktury używane do planowania przydziału czasu procesora. Następnie kalibrowana jest funkcja opóźnienia oraz jeżeli jest obsługiwany inicjowany jest Lokalny APIC [5] (ang. *Advanced Programmable Interrupt Controller*) procesora. Na samym końcu inicjowany jest ko-procesor matematyczny. Po powrocie z funkcji **cpu_init()** inicjowane są części jądra niezależne od architektury oraz linkowane z jądrem moduły załadowane przez program ładujący [6]. Po wykonaniu tego kroku jądro jest gotowe do uruchamiania aplikacji. Wykonywana jest funkcja **start_init()**, której celem jest stworzenie procesu dla pierwszej aplikacji (programu init). Jako punkt wejścia dla tego procesu (adres od którego rozpocznie się wykonywanie kodu jest podawana funkcja o nazwie **init()**. Przed uruchomieniem programu w trybie użytkownika funkcja ta musi zrealizować kilka czynności:

- Zamontować główny system plików
- Zainicjować deskryptory plików dla standardowego wejścia i wyjścia
- Następnie jeśli zdefiniowany jest parametr init w linii poleceń jądra, próbujemy wykonać zdefiniowaną aplikację.

- Próbujemy wykonać kolejno: /sbin/init oraz /bin/sh
- Jeżeli nie udało się wykonać żadnej z aplikacji, wyświetlamy informację o błędzie krytycznym

W chwili obecnej program /sbin/init, który jest wykonywany domyślnie uruchamia jedynie program powłoki (/bin/sh) oraz w przypadku jego zakończenia uruchamia go od nowa.

3.4. Zarządzanie zadaniami

Zarządzanie zadaniami w systemie zostało zaimplementowane z podziałem na procesy oraz wątki [7], przy czym kolejkowaniu, usypianiu czy budzeniu podlegają jedynie wątki a nie całe procesy. Każdy wątek w systemie jest opisywany przez poniższą strukturę:

```
struct thread
{
        list_t list; /* Lista wątków w danej kolejce */
        list_t t_list; /* Lista wątków w danym procesie macierzystym */

        tid_t tid; /* ID wątku */

        struct proc * proc; /* Proces macierzysty */

        int state; /* Stan wątku */
        int timeout; /* Czas do końca wykonania/czekania */
        int prio; /* Priorytet */

        int waitpid;
        int exit_code; /* Kod zakończenia */

        struct ctx ctx; /* Kontekst procesu */
};
```

Możliwe stany wątków w systemie to:

```
#define THREAD_STATE_RUNNING 0 /* Wątek aktualnie działający */
#define THREAD_STATE_READY   1 /* Wątek gotowy do wykonania */
#define THREAD_STATE_WAITING 2 /* Wątek oczekujący na jakiejś
blokadzie */
#define THREAD_STATE_NEW     3 /* Wątek stworzony funkcją
thread_init() */
```

```
    #define THREAD_STATE_BLOCKED 4 /* Wątek zablokowany */
    #define THREAD_STATE_ZOMBIE  5 /* Wątek zombie – czekający na
odczytanie informacji o zakończeniu przez proces macierzy funkcją wait() */
```

Wątki grupowane są w kolejki zaimplementowane na zasadzie listy dwukierunkowej. W systemie istnieje jedna globalna kolejka wątków gotowych do wykonania. W przypadku innego stanu wątków nie ma globalnej listy która by dane wątki zbierała. Każdy mutex czy semafor posiada własną listę wątków oczekujących na danej blokadzie. Jak widać w strukturze opisującej wątek nie ma informacji o przestrzeni adresowej czy otwartych plikach. Te dane zostały opisane w strukturze opisującej proces. Wygląda ona następująco:

```
struct proc
{
        list_t list; /* Lista procesów w systemie */
        list_t list_siblings; ./* Lista rodzeństwa */

        pid_t pid; /* ID procesu */
        uid_t uid; /* Identyfikatory użytkowników i grup */
        gid_t gid;
        uid_t euid;
        gid_t egid;

        list_t thread_list; /* Lista wątków */
        atomic_t threads; /* Ilość wątków */
        struct thread * main; /* Główny wątek */

        /* Lista procesów potomnych */
        list_t childs_list;
        struct proc * parent; /* Proces – rodzic */

        /* Dane VFS */
        struct vnode * vnode_root;
        struct vnode * vnode_current;
        struct filedes * filedes[OPEN_MAX];
        mode_t umask;

        struct vm_space * vmspace; /* Przestrzeń adresowa */

        spinlock_t lock; /* Blokada struktury */
};
```

Jak widać w tej strukturze przechowywane są wszystkie dane wspólne dla wątków jak przestrzeń adresowa, otwarte pliki, identyfikatory użytkownika, katalog roboczy, itp. Dodatkowo zostało zdefiniowane pole ***main*** które zawiera adres struktury opisującej główny wątek. W przypadku odwoływania się bezpośrednio do procesu przez funkcje systemowe jak **kill** czy **waitpid** odwołujemy się właśnie do głównego wątku w procesie.

Jako mechanizmy służące do synchronizacji wątków [7] w jądrze zostały zaimplementowane trzy mechanizmy:

- blokady wirujące (ang. *spin lock*)
- mutex (ang. ***Mut****ual* ***Ex****clusion*, wzajemne wykluczanie)
- semafory

Blokady wirujące są to najprostsze mechanizmy. Ich działanie polega na aktywnym oczekiwaniu na spełnienie warunku (zwolnienie blokady). Aby zapewnić poprawność działania blokady, zostały wykorzystane tzw. operacje atomowe, a dokładnie operacja sprawdź i ustaw. Implementacja operacji atomowych jest zależna od zastosowanej architektury. Musi ona jednak zapewniać to że operacja wykona się jednocześnie tylko na jednym procesorze oraz że będzie to jeden cykl pracy procesora (nie zajdzie międzyczasie np. przerwanie). Blokady wirujące używane są wszędzie tam gdzie wymagane jest zablokowanie obiektu na krótki czas lub nie możemy pozwolić sobie na przełączenie wątków.

Mutex (zwany też semaforem binarnym) jest to bardziej rozbudowany typ blokady, w którym aktywne oczekiwanie zostało zastąpione wstrzymaniem wykonywania wątku wywołującego aż do zwolnienia blokady. Jest do wydajniejsze przy blokowaniu na dłuższy czas, gdyż nie marnujemy czasu procesora na aktywne oczekiwanie wykonując w czasie oczekiwania inne wątki.

Semafor jest najbardziej rozbudowaną strukturą. Jego działanie jest analogiczne do jego pierwotnego odpowiednika stosowanego w kolei. Tak długo jak semafor jest podniesiony wątek może wejść w sekcję chronioną semaforem. Podczas wywoływania operacji blokowania za każdym razem zmniejszany jest licznik semafora. Gdy licznik ten osiągnie wartość zero kolejne wątki są usypiane aż któryś wątek zwolni blokadę (zwiększy licznik semafora). Mutex jest specyficznym typem semafora, z licznikiem początkowo ustawionym na jeden. Poza blokowaniem struktur semafory mogą być wykorzystywane np. do oczekiwania na jakieś zdarzenie. W tym wypadku licznik semafora domyślnie ustawia się na wartość zero. Wątki oczekujące na zdarzenie próbują blokować semafor, ale przez to,

że ma on wartość zero, są blokowane. Przy nadejściu zdarzenia zwiększa się licznik semafora, co powoduje obudzenie wątku oczekującego i ponowne zmniejszenie licznika.

Do zarządzania procesami w systemie, korzysta się z funkcji POSIX:

- **pid_t fork(void);**
- **int waitpid(int pid, int * status, int option);**
- **int execve(char * path, char * argv[], char * envp);**

Funkcja **fork()** tworzy kopię procesu wywołującego kopiując przestrzeń adresową, listę otwartych plików, itp. Proces potomny różni się od procesu macierzystego jedynie identyfikatorem procesu (PID) oraz identyfikatorem procesu-rodzica (PPID). Od momentu powrotu z funkcji **fork()** oba procesy zaczynają wykonywać się równolegle. Funkcja **fork()** zwraca:

- -1 – w przypadku błędu
- > 0 – identyfikator procesu potomnego
- 0 – w przypadku gdy nastąpił powrót z funkcji **fork()** do nowo utworzonego procesu

Funkcja **waitpid()** służy do oczekiwania na zakończenie procesu potomnego. Parametr status jest wskaźnikiem na zmienną w której należy zapisać kod zakończenia procesu. W przypadku gdy status ustawiony jest na ***NULL***, wartość ta nie jest zapisywana. Parametr ***option*** służy do określenia sposobu działania funkcji. Domyślna wartość to 0. Na chwilę obecną jedyną opcją jest ***WNOHANG***, która powoduje natychmiastowy powrót z funkcji **waitpid()**, nawet jeśli nie ma żadnych martwych procesów potomnych. Procesy na które chcemy oczekiwać określa się parametrem pid. Możliwe wartości pid to:

- > 0 – czekamy na proces o identyfikatorze równym ***pid***
- 0 – czekamy na dowolny proces z tej samej grupy do której należy proces wywołujący
- -1 – czekamy na dowolny proces o identyfikatorze grupy równej grupie procesu wywołującego
- < -1 – czekamy na dowolny proces z grupy o identyfikatorze grupy równym wartości bezwzględnej z pid

Funkcja **execve()** służy do wykonywania nowych aplikacji. Aktualna przestrzeń adresowa jest niszczona. W to miejsce tworzona jest nowa do której jest ładowana aplikacja z dysku z pliku o nazwie zdefiniowanej w parametrze ***path***. Parametr ***argv*** przetrzymuję tablice wskaźników na łańcuchy znaków, które zostaną przekazane jako parametry do nowo

wywoływanej aplikacji. Tablica na jest zakończona wskaźnikiem na adres pusty (***NULL***). Parametr ***envp*** przechowuje wskaźnik na tablicę zawierającą zmienne środowiskowe, które przechowywane są jako napisy w formacie NAZWA=WARTOŚĆ. Ta tablica jest również zakończona wskaźnikiem na adres pusty. Gdy funkcja zakończy się powodzeniem rozpoczyna się wykonywanie nowej aplikacji. W przypadku błędu zwracane jest -1.

Przełączanie zadań realizowane jest przez wywołanie funkcji **schedule()**. Funkcja ta wywoływana jest co określony czas poprzez przerwanie zegarowe. Na każdym procesorze jest ona wywoływana osobno. Funkcja ta wykorzystuje algorytm szeregowania Round Robin z dodatkowym polem określającym na jaki maksymalny czas proces otrzyma kontrolę nad procesorem [7] oraz z dodatkowym warunkiem że jeśli nie ma żadnych wątków gotowych do uruchomienia wybierany jest ukryty wątek bezczynności.

Implementacja funkcji schedule wygląda w następujący sposób:

```
void schedule(void)
{
 struct thread * old;

 /* Sprawdzamy czy przerwania są wyłączone */
 ASSERT(intr_disable() == 0);

 /* Jeżeli wywłaszczenie aktualnego procesu zostało wyłączone,
kończymy funkcję */
 if (atomic_get(&SCHED->preempt_disabled) > 0)
       return;

 /* Jeżeli aktualne zadanie jest uruchomione */
 if (SCHED->current->state == THREAD_STATE_RUNNING)
 {
       if (SCHED->current->timeout-- > 0) /* Zmniejszamy i sprawdzamy
czas do wywyłaszczenia */
                  return;

       /* Zmieniamy stan na gotowe */
       SCHED->current->state = THREAD_STATE_READY;
 }
 old = SCHED->current;

 /* Blokujemy kolejkę zadań gotowych */
```

```
	spinlock_lock(&_ready_queue_lock);

	/* Jeżeli aktualne zadanie nie jest wątkiem bezczynności oraz ma stan
"Gotowe do wykonania", dodajemy do kolejki oczekujących */
	if ((old->tid > 0) && (old->state == THREAD_STATE_READY))
		list_add(_ready_queue.prev, &old->list);

	/* Jeżeli lista oczekujących na wykonanie jest pusta, uruchamiamy
wątek bezczynności (IDLE) */
	if (LIST_IS_EMPTY(&_ready_queue))
	{
		SCHED->current = SCHED->idle;
		SCHED->current->timeout = 0;
	}
	else /* W przeciwnym wypadku wybieramy pierwszy wątek z kolejki */
	{
		SCHED->current = (struct thread *)_ready_queue.next;
		list_remove(&SCHED->current->list);
		SCHED->current->timeout = SCHED->current->prio;
	}

	/* Zmieniamy stan na uruchomione */
	SCHED->current->state = THREAD_STATE_RUNNING;

	spinlock_unlock(&_ready_queue_lock);

	/* Jeżeli musimy, zmieniamy proces */
	if (old != SCHED->current)
	{
		SCHED->ctxsw++; /* Aktualizacja statystyk */
		/* Przestrzeń adresową zmieniamy tylko przy zmianie procesu */
		if(old->proc->vmspace != SCHED->current->proc->vmspace)
			vm_space_switch(SCHED->current->proc->vmspace);

		/* Zmieniamy kontekst */
		ctx_switch(&old->ctx, &SCHED->current->ctx);
	}
}
```

Jak widać funkcja **schedule()** nie wie nic o procesach o stanie innym niż uruchomione lub gotowe do uruchomienia. Dzięki temu może istnieć dowolna ilość mini kolejek wątków

na przykład kolejka wątków oczekujących na dostęp do dysku czy terminala. Do budzenia uśpionych wątków oraz dodawania ich do kolejki gotowych do uruchomienia wykorzystywana jest funkcja **sched_wakeup()**. Jako parametr przyjmuje ona wątek, który powinien zostać obudzony.

W przypadku gdy chcemy mieć pewność że dany fragment kody wykona się w całości od razu, bez przełączania zadań, możemy użyć blokady wywłaszczenia danego zadania. Służą do tego funkcje **sched_preempt_disable()** - wyłącza wywłaszczenie zadania oraz **sched_preempt_enable()** która działa w sposób przeciwny. Blokada ta używana jest głównie w funkcjach które operują na strukturze procesu lub wątku i modyfikują newralgiczne punkty jak na przykład przestrzeń adresowa (podmiana przestrzeni adresowej przy wykonywaniu **execve()**). Dzięki zaimplementowaniu takiego tupu blokady nie musimy wyłączać przerwań podczas tego typu operacjach.

3.5. Zarządzanie pamięcią

Moduł zarządzania pamięcią operacyjną możemy podzielić na klika mniejszych części:

- zarządzanie pamięcią fizyczną
- zarządzanie pamięcią jądra (sterta jądra)
- alokator pamięci systemowej
- zarządzanie pamięcią tymczasową jądra
- zarządzanie pamięcią aplikacji

Mapa pamięci fizycznej została zaimplementowana jako bitmapa, gdzie bit zapalony oznacza stronę pamięci zajętą a bit wyzerowany stronę dostępną do użycia. Dodatkowo zostało wydzielone kilka stref pamięci fizycznej:

- Pamięć dostępna dla DMA (obszar pamięci który może być wykorzystywany przy transferach bez udziału procesora, dla architektury IA-32 jest to pierwsze 16MB RAM [2])
- Pamięć niska (obszar pamięci który jest mapowany bezpośrednio w przestrzeni adresowej jądra, dla wersji 32b jest to 512MB a dla wersji 64b cała dostępna pamięć fizyczna)
- Pamięć wysoka (obszar pamięci niedostępny bezpośrednio, występuje tylko w niektórych architekturach)

Moduł zarządzający udostępnia do alokowania / zwalniania następujące funkcje:

paddr_t phys_alloc(size_t len, unsigned flags); - funkcja alokuje obszar pamięci o długości len. Dodatkowo można określić flagi w parametrze flags, gdzie flags to suma bitowa następujących możliwości:

- ***PHYS_ZONE_NORMAL*** – zaalokowana strona może leżeć w pamięci niskiej
- ***PHYS_ZONE_DMA*** – zaalokowana strona może leżeć w obszarze DMA
- ***PHYS_ZONE_HIGH*** – zaalokowana strona może leżeć w obszarze pamięci wysokiej
- ***PHYS_ALLOW_WAIT*** – pozwól na uśpienie wątku który próbuje zaalokować stronę
- ***PHYS_ALLOW_IO*** – pozwól na operację wejścia/wyjścia podczas alokowania (np. zapis innnej strony do pamięci wymiany)

void phys_free(paddr_t start, size_t len); - funkcja oznacza obszar o długości ***len*** rozpoczynający się od adresu ***start*** jako dostępny dla systemu.

Do zarządzania pamięcią jądra systemu operacyjnego, został wykorzystany dostępny na licencji Public Domain alokator napisany przez Doug Lea [8]. Kod został odpowiednio zmodyfikowany na potrzeby jadra systemu (między innymi zostały dodane blokady odpowiednich struktur). Dzięki stworzeniu odpowiedniego pliku zawierającego makrodefinicie preprocesora, w jądrze do zarządzania pamięcią używa się funkcji z prefiksem „k” które są odpowiednikami ich wersji z biblioteki standardowej. Najczęściej wykorzystywane są:

- **void * kalloc(size_t len)** – alokuje pamięć o rozmiarze len
- **void kfree(void * ptr)** – zwalnia obszar rozpoczynający się od adresu wskazywanego przez wskaźnik ptr
- **void * krealloc(void * ptr, size_t len)** – zmienia rozmiar zaalokowanego wcześniej bloku pamięci wskazywanego przez ptr.

Alokator pamięci systemowej jest to uniwersalny alokator, wykorzystywany zarówno przez alokator pamięci tymczasowej jądra jak i przez moduł zarządzający pamięcią aplikacji. Został on zaimplementowany jako dwie listy bloków. Jedna z list przetrzymuje bloki wolne a druga używane. Obie listy są posortowane pod względem adresów bloków. Listy bloków grupowane są w ciągłe obszary pamięci. Do tworzenia obszaru słuzy funkcja: **struct sma_area * sma_area_create(addr_t start, size_t size, void * do_alloc, void * do_resize, void * do_free);** która tworzy nowy obszar rozpoczynający się od adresu ***start***, zajmujący długość ***size***. Pozostałe trzy parametry to wskaźniki na funkcję wywoływane

przy zalokowaniu, zmianie rozmiaru lub zwalnianiu pojedynczego bloku. Do operowania na obszarze pamięci służą funkcje:

- **struct sma_block * sma_alloc(struct sma_area * area, addr_t addr, size_t size, unsigned flags);** - funkcja alokuje z obszaru wskazywanego przez ***area***, blok o rozmiarze ***size*** i preferowanym adresie początkowym ***addr***.
- i**nt sma_free(struct sma_area * area, addr_t start, size_t size);** - funkcja zwalnia cały lub fragment bloku w obszarze ***area*** zaczynający się od start o długości ***size***
- **struct sma_block * sma_getblock(struct sma_area * area, addr_t addr);** - funkcja zwraca wskaźnik na strukturę opisującą blok w alokatorze, znajdujący się w obszarze ***adrea*** i zawierający adres ***addr***

Pamięć tymczasowa jądra jest to obszar pamięci wirtualnej, pod który możemy mapować pamięć fizyczną która jest potrzebna tylko przez jakiś czas. Jest to wykorzystywane np. przez funkcję kopiującą strony przy kopiowaniu przy zapisie. Jest ona również dość często używana w sterownikach urządzeń (do mapowania pamięci urządzenia w przestrzeni adresowej jądra systemu). Pamięć ta została zaimplementowana w oparciu o alokator pamięci systemowej. Jądro może alokować lub zwalniać tą pamięć poprzez wywoływanie następujących funkcji:

- **void * kmem_alloc(size_t len);** - alokuje obszar o rozmiarze len
- **void kmem_free(void * ptr, size_t len);** - zwalnia obszar od adresu wskazywanego przez ***ptr*** o długości ***len***
- **void * kmem_map_page(paddr_t phys);** - alokuje obszar o rozmiarze jednej strony a następnie mapuje stronę o adresie fizycznym ***phys*** pod zaalokowany adres i zwraca wskaźnik na ten adres
- **void kmem_unmap_page(void * ptr);** - funkcja odmapowywuje adres wskazywany przez ***ptr*** a następnie zwalnia obszar o rozmiarze jednej strony na którą wskazuje wskaźnik ***ptr***

Manager pamięci aplikacji został oparty o alokator pamięci systemowej. Każda aplikacja posiada własną przestrzeń adresową opisaną strukturą:

```
struct vm_space
{
      void * dataptr; /* Dane specyficzne dla architektury (np.
katalog stron) */
```

```
            struct mutex mutex; /* Blokada struktury */
            struct sma_area * area; /* Wskaźnik do obszaru alokatora
pamięci
systemowej (SMA) */
            atomic_t refs; /* Uzywane przy vfork() */
      };
```

Do zarządzania przestrzenią adresową służą funkcje:

- **struct vm_space * vm_space_create(void);** - funkcja tworzy nową, pustą przestrzeń adresową oraz mapuje w niej pamięć jądra
- **void vm_space_destroy(struct vm_space * vmspace);** - funkcja niszczy przestrzeń adresową oraz wszelkie struktury zależne od niej
- **void vm_space_switch(struct vm_space * newspace);** - funkcja zmienia przestrzeń adresową na podaną jako parametr.
- **int vm_space_clone(struct vm_space * dest, struct vm_space * src);** - funkcja służy do klonowania przestrzeni adresowej. W rzeczywistości nie jest kopiowana pamięć a jednynie niezbędne struktury.

W implementacji modułu zarządzającego pamięcią użytkownika zostały zaimplementowane dla algorytmy optymalizujące jego wydajność: kopiowanie przy zapisie oraz wczytywanie na żądanie. Kopiowanie przy zapisie polega na tym, że po wykonaniu funkcji **vm_space_clone()** obie przestrzenie adresowe posiadają zamapowane te same fizyczne strony w trybie tylko do odczytu. W momencie wywołania zapisu na stronie, wywoływany jest wyjątek, w którym następuje kopiowanie zawartości oryginalnej strony do nowo zaalokowanej strony fizycznej. Wczytywanie na żądanie polega na tym że, gdy chcemy zamapować plik na pamięć, zapisujemy w strukturze tylko strukturę opisującą go. Dane z pliku ładowane są w momencie odwołania do danej strony. Dzięki temu zyskujemy dużą optymalizację uruchamiania aplikacji (ładowane są tylko potrzebne fragmenty kodu/danych).

Interfejs zarządzający pamięcią użytkownika został zaprojektowany tak aby zapewnić jak największą zgodność z standardem POSIX [7]. Posiada on dwie główne funkcje:

- **int vm_mmap(void * start, size_t length, uint16_t flags, int fd, loff_t offset, struct vm_space * vmspace, void ** addr);**
- **int vm_unmap(void * start, size_t length, struct vm_space * vmspace);**

Funkcja **vm_mmap** jest odpowiednikiem funkcji **mmap()** ze standardu POSIX.

Jej zadaniem jest zamapowanie pamięci. Przyjmuje ona następujące parametry:

- ***start*** – preferowany początek mapowania (jeśli jest równy ***NULL*** to nieokreślony)
- ***lenght*** – rozmiar mapowanego obszaru w bajtach
- ***flags*** – flagi mapowania. Jest to suma bitowa następujących wartości:
  - ***PROT_NONE*** – obszar nie będzie dostępny
  - ***PROT_READ*** – obszar dostępny do odczytu
  - ***PROT_WRITE*** – obszar dostępny do zapisu
  - ***PROT_EXEC*** – obszar dostępny do wykonania kodu
  - ***MAP_SHARED*** – zawartość mapowania będzie wspólna dla wszystkich aplikacji które mapują dany obiekt
  - ***MAP_PRIVATE*** – zmiany w zawartości mapowanej pamięci nie będą widoczne dla innych aplikacji mapujących dany obiekt
  - ***MAP_FIXED*** – adres start jest jedynym akceptowalnym adresem mapowania. Jeżeli nie da się utworzyć obszaru od podanego adresu, funkcja zwróci błąd
  - ***MAP_ANONYMOUS*** – zamapowana pamięć zostanie wypełniona zerami (nie jest mapowany żaden obiekt a jednynie pusty obszar pamięci)
- ***fd*** – w przypadku mapowania pliku na pamięć, numer deskryptora mapowanego pliku. Ignorowane jeśli ustawiono ***MAP_ANONYMOUS***
- ***offset*** – przesunięcie mapowania względem początku pliku (używane tylko podczas mapowania pliku na pamięć)
- ***vmspace*** – przestrzeń adresowa w której ma zostać zamapowana pamięć
- ***addr*** – wskaźnik informujący pod jakim adresem umieścić rzeczywisty początkowy adres mapowania.

Funkcja **vm_unmap()** odmapowywuje zamapowaną wcześniej pamięć (można zwolnić całość lub fragment utworzonego wcześniej mapowania). Jako parametr przyjmuje:

- ***start*** – wskaźnik na początek obszaru do zwolnienia
- ***lenght*** – rozmiar obszaru w bajtach
- ***vmspace*** – wskaźnik na strukturę opisującą przestrzeń adresową w której chcemy zwolnić pamięć

Zarówno w funkcji **vm_mmap()** oraz **vm_unmap()** istnieje kilka reguł dotyczących parametrów:

- parametry ***start***, ***lenght*** oraz ***offset*** muszą być zawsze wyrównanie do rozmiaru

pojedynczej strony (zdefiniowanej poprzez makro ***PAGE_SIZE***)

- parametr ***flags*** musi zawierać co najmniej jedną z wartości: ***PROT_NONE***, ***PROT_READ***, ***PROT_WRITE*** lub ***PROT_EXEC***
- w przypadku gdy nie jest ustawione ***MAP_ANONYMOUS***, parametr ***fd*** musi zawierać poprawny numer deskryptora pliku.

3.6. System plików

W systemie został zaimplementowany system plików znany z systemów z rodziny UNIX. Polega on na istnieniu jednego głównego katalogu (ang. *root*), w którym tworzone są obiekty systemu plików (pliki, katalogi, pliki urządzeń, kolejki fifo, łącza, itp.). Dodatkowo każdy katalog może być punktem montowania innego systemu plików (np. innej partycji na dysku). Dzięki temu aplikacja korzystająca z pliku nie musi wiedzieć na jakiej partycji jest plik z którego korzysta. Pozwala to też na wprowadzenie jednego wspólnego interfejsu dla wszystkich systemów plików. Podstawowym pojęciem w implementacji modułu zarządzającego plikami jest pojęcie węzła wirtualnego (ang. *virtual node*, w skrócie *vnode*). W zastosowanej implementacji vnode przechowuje wszystkie informacje poza nazwą pliku o dowolnym obiekcie systemu plików (pliku, katalogu, itp.). Nazwa pliku nie jest przechowywana, ponieważ do jednego obiektu może być przypisane kilka nazw. Struktura vnode wygląda następująco:

```
struct vnode
{
	list_t list; /* Lista węzłów w punkcie montowania */
	ino_t ino; /* Numer i-węzła */
	mode_t mode; /* Prawa dostępu do obiektu oraz jego typ */
	nlink_t nlink; /* Ilość dowiązań do obiektu */
	loff_t size; /* Rozmiar w bajtach */
	uid_t uid; /* Identyfikator właściciela obiektu */
	gid_t gid; /* Identyfikator grupy będącej właścicielem obiektu */
	dev_t dev; /* Numer urządzenia na którym znajduje się obiekt */
	blkcnt_t blocks; /* Ilość bloków jaką zajmuje dany obiekt */
	blksize_t block_size; /* Rozmiar pojedynczego bloku obiektu */
	dev_t rdev; /* Uzywane w przypadku plików urządzeń: numer urządzenia
na który wskazuje plik urządzenia */
	struct mountpoint * mp; /*Punkt montowania na którym znajduje się
obiekt*/
	struct mountpoint * vfsmountedhere; /* Używane gdy obiekt jest
```

```
punktem montowania, wskazuje na strukturę opisującą punkt montowania */
      void * data; /* Dane specyficzne dla systemu plików, do użycia przez
dany sterownik */
      struct vnode_ops * ops; /* Wskaźnik na listę funkcji jaką możemy
wykonać na danym węźle */
      atomic_t refcount; /* Licznik odwołań do struktury */
      struct mutex mutex; /* Blokada struktury */
      list_t pagecache; /* Używane przez manager pamięci do trzymania listy
stron z pliku */
};
```

Aby wykonać jakąkolwiek operację na obiekcie systemu plików, korzystamy z funkcji znajdujących się w strukturze pod adresem wskazywanym przez pole ***ops*** w strukturze vnode. Nigdy nie wywołujemy bezpośrednio funkcji ze sterownika. Możliwe operacje to:

```
struct vnode_ops
{
      int (*open)(struct vnode * vnode);
      int (*close)(struct vnode * vnode);
      ino_t (*lookup)(struct vnode * vnode, char * name);
      ssize_t (*read)(struct vnode * vnode, void * buf, size_t len, loff_t
offset);
      ssize_t (*write)(struct vnode * vnode, void * buf, size_t len, loff_t
offset);
      int (*mkdir)(struct vnode * vnode, char * name, mode_t mode, uid_t
uid, gid_t gid);
      int (*creat)(struct vnode * vnode, char * name, mode_t mode, uid_t
uid, gid_t gid);
      int (*symlink)(struct vnode * vnode, char * name, char * path);
      int (*readlink)(struct vnode * vnode, char * buf, size_t len);
      int (*getdents)(struct vnode * vnode, struct dirent * dirp, size_t
len, loff_t * offset);
      int (*sync)(struct vnode * vnode);
      int (*mknod)(struct vnode * vnode, char * name, mode_t mode, dev_t
rdev, uid_t uid, gid_t
gid);
      int (*unlink)(struct vnode * vnode, char * name);
};
```

- **open** – ładuje z systemu plików informacje o obiekcie do struktury ***vnode***. Wymaga wypełnionych pól: ***mp*** oraz ***ino***
- **close** – zamyka obiekt, zapisuje wszystkie informacje z węzła na dysk, w przypadku gdy nie ma żadnych odwołań do obiektu usuwa go z dysku
- **lookup** – wyszukuje z danym węźle (katalogu) numer węzła dla obiekcie o podanej nazwie pliku. Jeśli plik nie istnieje zwraca 0
- **read** – odczytuje z obiektu do pamięci wskazywanej przez ***buf***, ilość bajtów określoną przez ***len***, począwszy od przesunięcia względem początku plików o wartość parametru ***offset***
- **write** – analogicznie do **read**, jednak zapisuje pamięć spod adresu wskazywanego przez ***buf*** na dysk
- **mkdir** – tworzy katalog o podanej nazwie i uprawnieniach w podanym obiekcie (katalogu)
- **creat** – analogicznie do **mkdir**, jednak zamiast katalogu tworzy regularny plik
- **symlink** – analogicznie do **mkdir**, jednak zamiast katalogu tworzy dowiązanie symboliczne
- **readlink** – odczytuje zawartość obiektu (dowiązania symbolicznego) do pamięci wskazywanej przez adres ***buf*** i maksymalnej długości ***len***
- **getdents** – odczytuje wpisy w podanym katalogu (ang. *directory entries*).
- **sync** – wymusza zapisanie na dysk informacji z węzła
- **mknod** – tworzy plik specjalny (np. plik urządzenia) w podanym katalogu o podanej nazwie, uprawnieniach oraz typie. W przypadku plików urządzeń wymagane jest podanie numeru urządzenia w parametrze ***rdev***
- **unlink** – usuwa z katalogu wpis o podanej nazwie. Zmniejszany jest również licznik odwołań do obiektu. W przypadku gdy licznik tez osiągnie wartość 0, obiekt jest usuwany z fizycznie z dysku podczas wywołania funkcji **close**

Aby maksymalnie uprościć pracę na plikach, zostały wprowadzone funkcje operujące na systemie plików które są odpowiednikami funkcji z biblioteki standardowej języka C dla jądra. Są to:

```
int sys_access(char * path, int mode, struct proc * proc);
int sys_chdir(char * path, struct proc * proc);
int sys_chroot(char * path, struct proc * proc);
int sys_open(char * path, int flags, mode_t mode, struct proc * proc);
```

```
int sys_close(int fd, struct proc * proc);
int sys_fstat(int fd, struct stat * stat, struct proc * proc);\
int sys_stat(char * path, struct stat * stat, struct proc * proc);
int sys_lstat(char * path, struct stat * stat, struct proc * proc);
ssize_t sys_read(int fd, void * buf, size_t len, struct proc * proc);
ssize_t sys_write(int fd, void * buf, size_t len, struct proc * proc);
loff_t sys_lseek(int fd, loff_t off, int whence, struct proc * proc);
int sys_getdents(int fd, struct dirent * buf, unsigned count, struct
proc * proc);
int sys_fcntl(int fd, int cmd, void * arg, struct proc * proc);
int sys_dup(int fd, struct proc * proc);
int sys_dup2(int oldfd, int newfd, struct proc * proc);
int sys_ioctl(int fd, int cmd, void * arg, struct proc * proc);
int sys_mount(char * src, char * dest, char * fstype, int flags, void
* data, struct proc * proc);
int sys_umount(char * dest, int flags, struct proc * proc);
int sys_mknod(char * path, mode_t mode, dev_t dev, struct proc *
proc);
int sys_chmod(char * path, mode_t mode, struct proc * proc);
int sys_mkdir(char * path, mode_t mode, struct proc * proc);
int sys_unlink(char * path, struct proc * proc);
int sys_link(char * oldpath, char * newpath, struct proc * proc);
int sys_truncate(const char *path, loff_t length, struct proc * proc);
int sys_ftruncate(int fd, loff_t length, struct proc * proc);
int sys_pipe(int fds[2], struct proc * proc);
```

Każda z tych funkcji jako ostani parametr przyjmuje wskaźnik na strukturę opisującą proces wywołujący. Jest to potrzebne do ustalenia listy otwartych plików, katalogu głównego oraz roboczego czy wartości maski uprawnień dla tworzenia nowych plików.

3.7. Wywołania systemowe

Wywołania systemowe są to funkcje, poprzez które aplikacje użytkownika komunikują się z jądrem systemu operacyjnego. Każda funkcja posiada przypisany unikalny numer przez który jest identyfikowana. W systemie sposób wywoływania funkcji zależny jest od architektury. Dla architektury x86, do wywoływania funkcji systemowych korzysta się z przerwania 128 (80h). Numer wywoływanej funkcji przekazywany jest w rejestrze ***eax***. Parametry zaś umieszcza się kolejno w rejestrach: ***ebx***, ***ecx***, ***edx***, ***esi***, ***edi*** [2]. Jak widać powoduje to powstanie limitu maksymalnie pięciu argumentów, co jednak jest wystarczającą

ilością. W kodzie jądra została wprowadzona globalna tablica wskaźników na funkcje realizujące odpowiednie wywołania. Dzięki temu niezależnie od architektury dana funkcja ma zawsze ten sam numer. W chwili obecnej zdefiniowane jest 52 funkcje systemowe. Pełną listę funkcji i ich numerów można znaleźć w pliku nagłówkowym sys/syscall.h. Jest to plik dołączony do biblioteki standardowej języka C. Funkcje systemowe przed wykonaniem odpowiedniej akcji, muszą sprawdzić czy otrzymały poprawne argumenty (np. czy adres wskaźnika leży w przestrzeni użytkownika). Jest to konieczne aby zapobiec dostępowi do pamięci jądra. Niesprawdzenie argumentów może skutkować że złośliwy program może nadpisać fragment pamięci jądra co może prowadzić do przejęcia kontroli nad systemem. Wszystkie funkcje z kontrolą argumentów są poprzedzone prefiksem _ (podkreślenie, np. **_sys_exit**). Takie rozróżnienie jest wymagane, ponieważ jądro oraz moduły również korzysta z funkcji bez prefiksów do wywoływania swoich funkcji. Np. aby otworzyć plik, moduł jądra skorzysta z funkcji **sys_open()**. Natomiast jeśli żądanie otwarcia pliku zgłasza aplikacja, pierwsze wywoływana jest funkcja **_sys_open()**. W niej następuje kontrola parametrów i jeśli parametry są poprawne wywoływana jest funkcja **sys_open()**. Przykładowa implementacja funkcji systemowej wygląda następująco:

```
static int _sys_open(char * path, int flags, mode_t mode)
{
        /* Wykonujemy kontrolę adresu pod którym zapisana jest nazwa
pliku, jeśli jest niepoprawny zwracamy błąd */
        if (!IS_IN_USERMEM(path))
              return -EFAULT;

        /* Wywołujemy rzeczywistą funkcję jądra */
        return sys_open(path, flags, mode, SCHED->current->proc);
}
```

4. Technologia wykonania

System został napisany w większości w języku ANSI C [9]. Dodatkowo gdzie nie dało się tego uniknąć lub było to wygodniejsze został użyty język Assembler [10] w składni AT&T.

Dzięki zastosowaniu języka C, większość elementów jądra (jak na przykład obsługę wirtualnego systemu plików czy zarządzania urządzeniami) można przenosić na różne

architektury. W chwili obecnej dostępne są wersje dla procesorów i686 i nowszych oraz dla Amd 64 (x86_64). Jednak w miejscach gdzie niezbędne było wykorzystanie języka niskiego poziomu (obsługa przerwań, operacje na portach I/O) został wykorzystany Assembler. Dodatkowo pliki źródłowe systemu zostały podzielone na zależne i niezależne od architektury. Pliki assemblerowe oraz wstawki tego języka w kodzie C, znajdują się tylko w pierwszej grupie plików źródłowych.

Do kompilacji wykorzystywane są narzędzia GNU Binutils oraz GNU Gcc z nałożonymi autorskimi łatkami dostosowującymi format plików wynikowych do architektury systemu. Jako narzędzie ułatwiające budowanie wykorzystywany jest program GNU Make oraz kilka innych programów (jak na przykład: GNU AWK, sed, grep, itp.).

System budowania został tak zaprojektowany aby umożliwić niezależną kompilację dowolnych elementów systemu przy założeniu że moduły jądra kompilowane są z wykorzystaniem pliku Makefile jądra. Najbardziej rozbudowanym modułem systemu budowania jest plik Makefile dla jądra oraz modułów. Jego zadaniem jest:

- Wykrycie czy budujemy system poprzez kompilacją skrośną
- Wygenerowanie na podstawie pliku konfiguracyjnego jądra takich plików jak:
  - .version.c – zawierającego informacje o wersji systemu
  - include/autoconf.h – plik nagłówkowy z informacjami o konfiguracji jądra (włączone funkcje, itp.).
- Wstępne skompilowanie i zlinkowanie obrazu .kernel.rel
- Wygenerowanie na podstawie pliku .kernel.rel pliku .symtab.c który zawiera informacje o symbolach udostępnianych przez jądro modułom
- Końcowe linkowanie jądra do pliku kernel.sys

W pliku Makefile dla jądra zdefiniowane są następujące cele:

- all – buduje jądro oraz moduły zdefiniowane w pliku konfiguracyjnym
- clean – czyści katalogi z plików tymczasowych
- install – instaluje jądro
- modules – kompiluje moduły zdefiniowane w pliku konfiguracyjnym
- modules_install – instaluje moduły zdefiniowane w pliku konfiguracyjnym
- kernel.sys – kompiluje oraz linkuje plik z obrazem jądra systemu

Dla każdego budowanego modułu istnieje plik opisujący w jakiś sposób moduł powinien być zbudowany. Plik nazywa się Kbuild i znajduje się zawsze w katalogu z plikami źródłowymi modułu. Przykładowy plik Kbuild wygląda tak:

```
# Pliki źródłowe modułu
MOD_SOURCES=main.c chan.c pio.c patapi.c pata.c part.c
# Nazwa pliku wynikowego
TARGET=ata.ko
```

Plik ten zawiera jedynie niezbędne informacje, bez których zbudowanie modułu było by niemożliwe. Wszelkie pozostałe reguły oraz zmienne definiowane są w pliku Makefile jądra systemu operacyjnego. Pozwala to na uproszczenie plików opisujących modułu oraz uniknięcie dublowania reguł oraz zmiennych w plikach Kbuild dla każdego modułu.

W przypadku systemu budowania aplikacji, został stworzony szablon pliku Makefile, znajdujący się w katalogu src/scripts o nazwie application.mak. Jest on używany do kompilacji, linkowania oraz instalacji wszystkich aplikacji dostarczanych razem z systemem. Dzięki zastosowaniu takiego szablonu plik Makefile zostaje znacznie uproszczony.

Przykładowy plik Makefile dla aplikacji sbin/init wygląda tak:

```
# Definiujemy katalog główny źródeł (ścieszka do katalogu src)
TOPDIR=../..
# Definiujemy w jakim katalogu aplikacja ma się znajdować po zainstalowaniu
APPDIR=sbin
# Nazwa aplikacji, jest to jednocześnie nazwa pliku wykonywalnego
APPNAME=init
# Lista wszystkich plików źródłowych oddzielonych spacją
SOURCES=init.c
# Jeżeli aplikacja używa dodatkowych bibliotek to umieszczamy je tutaj jako parametry dla linkera ld np. -lncurses
LDADD=
# Dodatkowe flagi kompilatora/linkera (zgodne ze składnią parametrów dla gcc/ld)
LDFLAGS=
CFLAGS=
# Na samym końcu pliku Makefile ładujemy szablon, który zdefiniuje odpowiednie
```

```
# cele i reguły dla GNU make
include $(TOPDIR)/scripts/application.mak
```

Dzięki zastosowaniu takiego systemu szablonów pliki Makefile są znacznie prostsze oraz unika się powtarzania w każdym pliku tych samych reguł.

5. Opis załączonej płyty CD

Załączona płyta CD zawiera zarówno kod źródłowy całego projektu jak i obraz dysku z zainstalowanym systemem oraz emulator QEMU. Kompletny kod źródłowy znajduje się na płycie CD w pliku archiwum idylla-src.tar.bz2. Do skompilowania systemu we własnym zakresie wymagane są narzędzia do kompilacji skrośnej. W celu zbudowania takiego kompilatora należy uruchomić skrypt mktoolchain.sh z parametrem all znajdujący się w katalogu toolchain.

W celu uruchomienia systemu z załączonego obrazu dysku twardego należy uruchomić plik qemu.bat z głównego katalogu płyty CD. Po uruchomieniu systemu w trybie normalnym program init powinien uruchomić powkłokę (program GNU Bash). Po załadowaniu powłoki możemy rozpocząć pracę z systemem. Najczęściej używane polecenia to:

- cd katalog – zmienia katalog roboczy na podany
- pwd – wypisuje na ekran aktualny katalog roboczny
- ls – wypisuje listę obiektów w danym katalogu na ekran
- cat plik – wypisuje zawartość pliku o podanej nazwie na standardowe wyjście
- clear – czyści ekran
- tty – wypisuje na ekran nazwę aktualnego terminala
- history – pokazuję historie wpisanych poleceń
- exit – kończy pracę powłoki
- nano – uruchamia edytor tekstu nano
- mount – montuje podany system plików (aktualnie obsługiwany jest tylko ramfs)

W załączonym obrazie systemu zostały zainstalowane następujące aplikacje oraz biblioteki pochodzące ze źródeł zewnętrznych:

- GNU GRUB, wersja 0.97
- GNU Bash, wersja 4.1.0
- GNU Binutils, wersja 2.20.1

- GNU Gcc, wersja 4.5.0
- GNU Nano, wersja 2.2.5
- GNU Ncurses, wersja 5.7

Nie wszystkie aplikacje posiadają pełną funkcjonalność, głównym problemem jest brak zaimplementowanej funkcji systemowej select. Przez to nie działają np. klawisze strzałek w edytorze GNU nano.

5.1. Wymagania sprzętowe:

- procesor klasy i686 lub nowszy (dla wersji 32 bitowej) lub zgodny z amd64 (dla wersji 64 bitowej)
- co najmniej 32MB pamięci operacyjnej
- karta graficzna zgodna z VGA
- dysk IDE (lub SATA w trybie emulacji IDE) zawierający partycję sformatowaną w systemie plików ext2 zawierającą pliki systemu operacyjnego
- klawiatura PS/2

5.2. Obsługiwany sprzęt oraz funkcje:

- Pamięć operacyjna: 4GB (wersja 32 bitowa) lub 128TB (wersja 64 bitowa)
- Procesory: do 256 procesorów zgodnych z i686 lub amd64
- Karta graficzna: zgodna z VGA, tryb tekstowy
- Urządzenia wejściowe: Klawiatura zgodna z PS/2 (lub USB w trybie emulacji PS/2)
- Dyski twarde: IDE (lub SATA w emulacji IDE)
- Napędy optyczne: zgodne z ATAPI
- Systemy plików: ext2 (niepełne wsparcie zapisu na dysk)
- Format aplikacji: Statycznie linkowane pliki ELF

6. Podsumowanie

Głównym problemem napotkanym podczas tworzenia systemu był problem rozwiązania synchronizacji do wszystkich elementów jądra aby zmaksymalizować wydajność oraz uniknąć zakleszczeń. Problem ten był dlatego taki trudny do rozwiązania gdyż, debugowanie działającego jądra przy uruchomionych kilku wątkach programami do tego przeznaczonymi

było praktycznie niemożliwe. Dlatego do wyszukiwania błędów wykorzystywane były głównie funkcje wypisujące na ekran wartości poszczególnych zmiennych oraz wykonujące zrzut stosu pokazując dzięki temu kolejność wywoływania funkcji przez jądro. Nie wszystkie problemy zostały rozwiązane, co może objawiać się od czasu do czasu fatalnymi błędami oraz paniką jądra (ang. *kernel panic*).

Kolejnym krokiem w rozwoju systemu będzie poprawa wszelkich znalezionych błędów, implementacja zapisu na systemie plików ext2. Następnym celem będzie stworzenie sterowników do popularnych urządzeń oraz systemów plików (system plików ISO9660, FAT, obsługa stacji dyskietek). Międzyczasie prowadzone będą prace nad przeniesieniem większej liczby aplikacji oraz zapewnienia jak największej zgodności ze standardem POSIX. Powinno to pozwolić na skompilowanie systemu pod nim samym, a tym samym uniezależnić go od innych systemów. Na chwilę obecną nie jest to cel daleki, same kompilatory jak GCC czy Binutils działają już w systemie. Potrzebne jest tylko kilka narzędzi jak awk, Make, itp.

Literatura:

1. Projects - OSDev Wiki, http://wiki.osdev.org/Projects
2. Silberschatz A., Galvin P.B., Gagne G., Podstawy Systemów Operacyjnych, Wydawnictwa Naukowo-Techniczne, Warszawa 2006
3. POSIX.1 FAQ, http://www.opengroup.org/austin/papers/posix_faq.html
4. Multiboot Specification version 0.6.96, http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
5. IA-32 Intel Architecture Software Developer's Manual Vol. I-III, Intel 2005
6. System V Application Binary Interface – DRAFT, http://www.sco.com/developers/gabi/latest/contents.html
7. Uresh V., Jądro systemu UNIX – nowe horyzonty, Wydawnictwa Naukowo-Techniczne, Warszawa 2001
8. malloc.c, ftp://gee.cs.oswego.edu/pub/misc/malloc.c
9. C (język programowania), http://pl.wikipedia.org/wiki/C_(język_programowania)
10. Assembler, http://pl.wikipedia.org/wiki/Asembler