OS–9 – modułowy, wielozadaniowy system czasu rzeczywistego

materiały do wykładu

Marek Wnuk

OS–9 firmy Microware jest kompletnym, wielozadaniowym, modułowym systemem czasu rzeczywistego. Zawiera nie tylko jądro czasu rzeczywistego (*real-time kernel*) wraz z towarzyszącymi mu modułami, ale również moduły obsługi plików (*file managers*) i sterowniki urządzeń (*device drivers*) umożliwiające tworzenie elastycznego systemu wejścia - wyjścia. Udostępnia mechanizmy podziału czasu (*time sharing*) i wielozadaniowości (*multi-tasking*). Jest standardowo wyposażony w Unix-o podobny interfejs użytkownika (*shell*, hierarchiczny system plików, komendy ułatwiające administrowanie systemem). W wersji profesjonalnej (*Professional OS–9*) jest wyposażony w kompilatory języków wyższego rzędu, debuggery i narzędzia komunikacyjne umożliwiające współpracę z innymi systemami. Modułowa budowa OS–9 umożliwia przystosowanie rozmiarów i możliwości systemu docelowego do potrzeb konkretnej aplikacji, bez zamykania drogi ewentualnej dalszej rozbudowy. Jest dostępny w wersji *Industrial OS–9* jako przeciwny biegun bardzo szerokiego spektrum zastosowań (od prostych sterowników - w wersji zROMowanej do dużych systemów pracujących w sieciach). Warto wspomnieć, ze jedno z jego wcieleń zyskuje coraz większą popularność w CD-i jako CD-RTOS.

1 Modułowa budowa OS–9

1.1 Moduły pamięciowe OS–9

Koncepcja modułów pamięciowych OS–9 stanowi sedno wielu zalet, które pozwalają na stosowanie tego systemu w szerokim zakresie aplikacji. Pozwala ona systemowi operacyjnemu panować nad programami, wspólnymi obszarami danych, składnikami systemu operacyjnego. Niezależnie od ich bezwzględnego położenia w pamięci, można się do nich odwoływać przez nazwę, podobnie jak do plików na dysku.

Moduł OS–9 może zawierać program, dane, część systemu operacyjnego. Zaczyna się nagłówkiem, a kończy cykliczną sumą kontrolną (CRC).

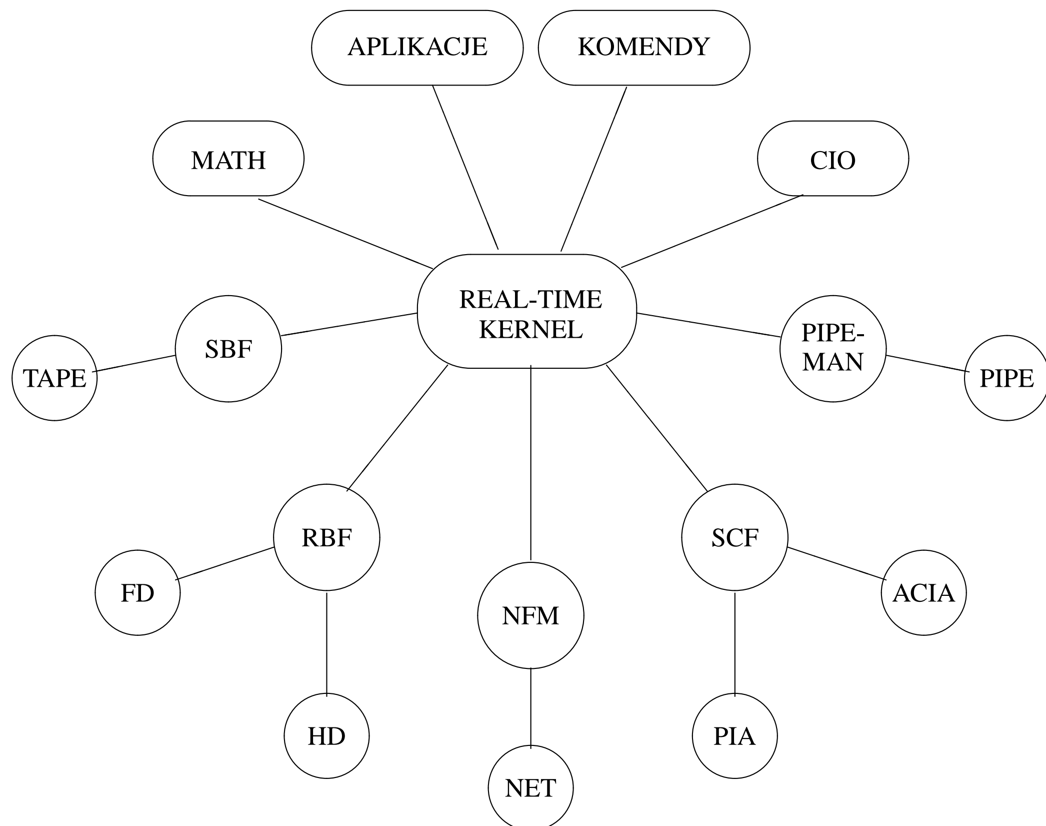
Nagłówek zawiera informacje o module, niezbędne do jego rozpoznania i prawidłowego wykorzystania w systemie. Pozwala on znaleźć moduł w pamięci przy starcie systemu, zainicjować odpowiednie obszary danych i sprawdzić jego poprawność, dzięki sumie kontrolnej i CRC. Zawiera też wskaźnik do nazwy modułu.

Moduł jest dostępny przez nazwę, musi ona być unikalna wśród modułów załadowanych do pamięci, choć na dysku (w plikach) może się powtarzać.

Adresy wszystkich załadowanych aktualnie modułów są zebrane w kartotece modułów (*module directory*), która jest tablicą w pamięci obsługiwaną przez system operacyjny. Każdy jej element zawiera:

- adres modułu
- licznik bieżących użytkowników (*link count*)
- identyfikator grupy
- sumę kontrolną nagłówka modułu

Moduł jest dostępny w systemie dzięki funkcji systemowej F$Link (*link to module*), której dostarcza się nazwę modułu. Jądro systemu (*kernel*) przegląda moduły w kartotece, adres modułu wskazuje na jego nagłówek, w którym z kolei znajduje się wskaźnik do nazwy. Jej zgodność z nazwą zadaną kończy poszukiwanie.



Rysunek 1: Struktura modułowego systemu czasu rzeczywistego OS–9

Moduły mogą znajdować się w pamięci stałej (ROM), lub być ładowane przy starcie systemu, lub w dowolnej chwili podczas jego pracy. Podczas ładowania jądro systemu rezerwuje pamięć zgodnie z rozmiarem pliku, wpisuje jego zawartość i przeszukuje zapisany obszar w celu wykrycia poprawnych modułów. Znalezione moduły są dołączane do kartoteki modułów.
Każde użycie modułu (F$Link) powoduje zwiększenie licznika (*link count*), a każde zakończenie używania (F$Unlk) - zmniejszenie. Moduły o własności *sticky* pozostają w kartotece również przy zerowej wartości licznika, co pozwala uniknąć wielokrotnego ładowania z pliku często używanych modułów. Przy braku miejsca w pamięci modułu o zerowym liczniku użyć są usuwane. Większość podstawowych programów użytkowych firmy Microware ma własność *sticky*. Moduły bez tej własności są usuwane przy zerowaniu licznika użyć.
Programy występują wyłącznie w formie modułów. Mogą być ładowane przez system w dowolny obszar pamięci. Są one napisane w sposób niezależny od lokalizacji (*position-independent code*). Obszary danych są odrębne od obszarów programu i mogą być umieszczane w dowolnym miejscu pamięci. Pozwala to na ROM-owalność i współużywalność

modułów programowych.
System operacyjny OS–9 jest również podzielony na moduły, co ułatwia jego dopasowanie do konkretnych potrzeb wynikąjących z aplikacji. Moduły umieszczone w ROM i pliku ładowanym przy starcie systemu (*bootfile*) są automatycznie włączane do kartoteki modułów, nie ma więc potrzeby specjalnego budowania systemu, a jedynie trzeba dodawać lub wymieniać moduły.

1.2 Budowa modułu (na przykładzie modułu typu *program*)

Nagłówek modułu i CRC są automatycznie tworzone przez linker w trakcie kompilacji lub asemblacji programu. Składa się on z części wspólnej dla wszystkich modułów zakończonej sumą kontrolną i części specyficznej dla każdego typu modułu (zawartego w części wspólnej).

Przykład: moduł programowy

```
-----------------------------------------------------------------------------
adres   wielkosc    nazwa w C   nazwa w asm
-----   --------    ---------   -------------- WSPOLNA CZESC NAGLOWKA
$0000   word (2)    _msync      M$ID           Znacznik modulu (\$4AFC)
$0002   word (2)    _msysrev    M$SysRev       Numer wersji systemu
$0004   long (4)    _msize      M$Size         Wielkosc modulu
$0008   long (4)    _mowner     M$Owner        Wlasciciel modulu
$000C   long (4)    _mname      M$Name         Wskaznik nazwy modulu
$0010   word (2)    _maccess    M$Accs         Zezwolenia na dostep
$0012   word (2)    _mtylan     M$Type,M$Lang  Typ i jezyk
$0014   word (2)    _mattrev    M$Attr,M$Revs  Atrybuty i wersja modulu
$0016   word (2)    _medit      M$Edit         Numer edycji modulu
$0018   long (4)    _musage     M$Usage        Wskaznik komentarzy
$001C   long (4)    _msymbol    M$Symbol       Wskaznik tablicy symboli
$0020   word (2)    _mident                    Kod identyfikacyjny modulu
$0022               _mspare                    Zarezerwowane
$002e   word (2)    _mparity    M$Parity       Suma kontrolna naglowka
-----   --------    ---------   -------------- ROZSZERZENIE NAGLOWKA (Prog)
$0030   long (4)    _mexec      M$Exec         Wskaznik startu programu
$0034   long (4)    _mexcpt     M$Excpt        Wskaznik obslugi TRAP
$0038   long (4)    _mdata      M$Mem          Wielkosc obszaru danych
$003C   long (4)    _mstack     M$Stack        Wielkosc stosu
$0040   long (4)    _midata     M$IData        Wskaznik inicjacji danych
$0044   long (4)    _midref     M$IRefs        Wskaznik inicjacji wskaznikow
-----   --------    ---------   -------------- WLASCIWY PROGRAM
$0048
...
-----   --------    ---------   -------------- CYKLICZNA SUMA KONTROLNA
        long (4)                               CRC modulu
-----------------------------------------------------------------------------
```

1.3 Moduły w plikach

Moduły mogą być umieszczone w plikach pojedynczo, lub grupami. Często (ale nie koniecznie) nazwa pliku jest zgodna z nazwą modułu. Program load używa funkcji systemowej F$Load do zarezerwowania pamięci na ładowany plik, załadowania go, sprawdzenia (F$ValMod) modułów znalezionych w zapisanym obszarze. Licznik użyć (*link count*) pierwszego modułu z pliku jest ustawiany w kartotece modułów na 1, pozostałych - na 0.

Unlink (F$UnLink) zmniejsza o 1 licznik użyć modułu. Jeśli w ten sposób osiąga on 0, to moduł (o ile nie jest *sticky*) jest usuwany z kartoteki, a zajmowana przezeń pamięć jest zwalniana. Moduły *sticky* wymagają zmniejszenia link count do wartości -1.
Uruchamianie nowego procesu (F$Fork) powoduje poszukiwanie modułu o zadanej nazwie w kartotece modułów pamięciowych, a w przypadku jego braku - załadowanie pliku o tej nazwie z katalogu programów (*execution directory*) i uruchomienie pierwszego modułu z tego pliku (niezależnie od jego nazwy). Moduły nie mające zezwolenia na uruchamianie (*pe,ge,ue*) są odrzucane z błędem E_NEMOD - moduł nieuruchamialny.

1.4 Pamięć w OS–9

Jądro systemu OS–9 (*kernel*) dynamicznie przydziela pamięć i zwraca do puli wolnej pamięci w miarę potrzeby. Nie jest używany mechanizm pamięci wirtualnej, programy używają pośredniego adresowania względem bazowych rejestrów adresowych.
Jądro podczas uruchamiania procesu przydziela mu pamięć statyczną i obszar stosu według modułu głównego (zawierającego pierwotny program). Podczas działania, program może dynamicznie żądać przydzielania pamięci i zwalniać ją (do 32 obszarów łącznie). W miarę przydzielania pamięci dokonyawne jest (w miarę możliwości) scalanie sąsiadujących obszarów.

1.5 Pamięć kolorowana (*coloured memory*)

Microware wprowadza typy ("kolory") pamięci i priorytety dostępu, by ułatwić wpływanie przez użytkownika systemu na decydowanie o kolejności i sposobie wykorzystywania zasobów pamięci RAM. Niektóre części pamięci komputera są szybsze (na płycie procesora), inne - wolniejsze (poprzez magistralę kasety). Ponadto, niektóre obszary mogą mieć specjalne własności (dwubramowa pamięć karty video, bufor komunikacyjny pakietu podrzędnego, pamięć z podtrzymaniem bateryjnym).
Lista kolorowych obszarów pamięci jest umieszczona w module init, łatwe jest więc konfigurowanie pamięci przez użytkownika.
Dwie usługi systemowe:
_srqmem() (F$SRqMem) i _srqcmem() (F$SRqCMem)
pozwalają w programach żądać konkretnego, lub dowolnego typu (koloru) pamięci. Można również użyć koloru pamięci jako parametru przy funkcjach F$Load i F$DatMod.
Pamięć o zadanym kolorze jest przydzielana według priorytetów w danym typie (kolorze), a bez specyfikacji koloru - zgodnie z priorytetami wszystkich typów łącznie.

1.6 Przydzielanie pamięci (*memory allocation*)

Jądro systemu OS–9 korzysta z oddzielnych list wolnej pamięci dla każdego priorytetu każdego koloru. Każdy wolny obszar ma w nagłówku:

```
$0000   long (4)    adres nastepnego elementu
$0004   long (4)    adres poprzedniego elementu
$0008   long (4)    wielkosc obszaru (w bajtach)
```

Jądro obsługuje również tablicę obszarów znalezionych przy starcie systemu, opisująca adres początku i końca, numer koloru, priorytet i atrybuty każdego obszaru pamięci.

1.7 Ochrona pamięci (*inter-task memory protection*)

OS–9 nie korzysta z translacji adresów (pamięć wirtualna). Sprzęt do zarządzania pamięcią (MMU) jest używany do zabezpieczania obszarów pamięci przed niepożądanym dostępem z innego procesu. Dla procesorów rodziny 68xxx posiadających MMU istnieje specjalny

moduł ssm (*system security module*), który musi być w pamięci podczas startu systemu (w ROM lub *bootfile*).
Uruchamiany proces ma przydzieloną pamięć statyczną i stos, a także pamięć zajętą przez jego główny moduł (*primary module*). Każdy proces ma swą mapę pamięci, która jest modyfikowana przy przydzielaniu i zwalnianiu obszarów przez program. Obszar tworzonych, lub przyłączanych modułów daych jest dodawany do mapy pamięci procesu zgodnie z zezwoleniami na dostęp zawartymi w nagłówku modułu.

2 System wejścia–wyjścia w OS–9

W OS–9 różne urządzenia zewnętrzne są widoczne dla użytkownika w sposób ujednolicony. Do obsługi każdego urządzenia niezbędne są trzy moduły:

- deskryptor urządzenia (*Device Descriptor* - np. **term**, **d0**, **dd**)
- moduł obsługi plików (*File Manager* - np. **SCF**, **RBF**, **SBF**)
- sterownik urządzenia (*Device Driver* - np. **rbVMSC**)

Deskryptor urządzenia zawiera nazwy pozostałych modułów. System wykonuje operacje na fizycznym urządzeniu na podstawie wywołania tego urządzenia przez nazwę deskryptora.
Różne klasy urządzeń mają oczywiście różne zestawy operacji, które można na nich wykonać (np. `dir`, `format` są oczywiste dla dysku, niedostępne dla terminala). Mimo to unifikacja systemu we-wy pozwala łatwo zastępować jedne urządzenia innymi bez modyfikacji programów.

2.1 Struktura wejścia–wyjścia w OS–9

System wejścia/wyjścia OS–9 jest skonstruowany hierarchicznie. Wszystkie wywołania systemowych funkcji we/wy trafiają do jądra systemu (moduł **kernel**), skąd są kierowane do odpowiedniego dla tej klasy urządzeń modułu obsługi plików (*file manager*). Wykonanie fizycznych operacji na konkretnym urządzeniu we/wy jest zlecane odpowiedniemu modułowi sterownika urządzenia (*device driver*).
Dopuszczalne jest istnienie wielu modułów obsługi plików i sterowników urządzeń. Każdy moduł obsługi plików przystosowany jest do pewnej klasy urządzeń. Przykłade m może być RBF (*Random Block File manager*) obsługujący urządzenia o blokowym dostępie swobodnym (dyski), lub SCF (*Sequential Character File manager*) przeznaczony do obsługi urządzeń operujących strumieniem znaków (terminale, drukarki).
File manager wykonuje tylko operacje o charakterze logicznym, co pozwala zachować szeroki zakres jego przydatności. Fizyczne przesyłanie danych wykonywane jest na jego zlecenie przez sterownik urządzenia - moduł programowy napisany specjalnie dla konkretnego fizycznego urządzenia we/wy, jak UART, kontroler dysków, czy karta sieciowa.
Każde urządzenie jest opisane przez specjalny moduł danych - deskryptor urządzenia (*device descriptor*). Urządzenie jest w systemie znane pod nazwą deskryptora poprzedzoną przez ”/”. Przykład: moduł **term** opisuje urządzenie **/term**. Deskryptor zawiera nazwę modułu obsługi plików i modułu sterownika urządzenia, adres fizycznego urządzenia, zestaw parametrów i opcji definiujących konkretne własności urządzenia we/wy.
Taka konstrukcja systemu wejścia/wyjścia pozwala na użycie tego samego sterownika do wszystkich urządzeń korzystających z takiego samego sprzętu pzez zdefiniowanie osobnych deskryptorów z innymi adresami i wektorami obsługi przerwań. Można też na jednym fizycznym urządzeniu zdefiniować wiele urządzeń logicznych (deskryptory o różnych nazwach), różniących się parametrami, modułami obsługi plików, lub nawet modułami

sterowników. Na przykład, dwa deskryptory dla tego samego portu szeregowego mogą pozwalać na współpracę z terminalem i drukarką, na pracę w trybie asynchronicznym i synchronicznym (w tym przypadku zazwyczaj różnią się modułem sterownika). Inny przykład to deskryptory dysków elastycznych o różnych formatach możliwych do obsłużenia w tym samym napędzie.

2.2 Podsystem wejścia–wyjścia

Podsystem we/wy składa się z modułu obsługi plików, sterownika urządzenia, deskryptora urządzenia i obszaru pamięci statycznej danego urządzenia. Wszystkie podsystemy są umieszczone w tablicy urządzeń (*device table*). Urządzenie we/wy może być miejscem składowania danych (jak dysk), lub interfejsem z otoczeniem (jak port szeregowy). Poza tym dostępne są urządzenia abstrakcyjne, nie mające fizykalnego charakteru, a istniejące w postaci reprezentacji w pamięci RAM. Takimi urządzeniami są łącza (*pipes*) i pseudodyski (*RAM disks*). Obsługa różnie zrealizowanych urządzeń przez jądro systemu (*kernel*) jest jednakowa.
System OS–9 dynamicznie inicjuje i zakańcza pracę podsystemów we/wy. Kernel dokonuje odpowiednich operacji w chwili otwierania ścieżki dostępu do pliku na danym urządzeniu wykonując funkcję I$Attach (wbudowana w jądro). Powoduje ona dołączenie (*link*) modułu deskryptora urządzenia (co zwiększa jego ilość użyć - *link count*). Jeśli w tablicy urządzeń nie ma deskryptora o tym samym adresie, to jądro tworzy podsystem we/wy:

1. pobiera nazwy modułu obsługi plików i sterownika z deskryptora;
2. przyłącza te moduły;
3. tworzy nowe miejsce w tablicy urządzeń;
4. ustawia ilość użyc urządzenia na 1;
5. rezerwuje miejsce w RAM na pamięć statyczną urządzenia;
6. uruchamia procedurę inicjacji (ze sterownika);
7. w przypadku powodzenia - wypełnia miejsce w tablicy urządzeń, przy błędzie - odłącza urządzenie;

2.3 Deskryptory urządzeń

Moduł deskryptora urządzenia jest małym modułem danych. Zawiera on część standardową i część opcjonalną, zależną od typu urządzenia (modułu obsługi plików). Część standardowa jest umieszczona po nagłówku:

```
-----   --------    ---------   --------------  ROZSZERZENIE NAGLOWKA (Desc)
$0030   long (4)    _mport      M$Port          Adres portu
$0034   byte (1)    _mvector    M$Vector        Wektor obslugi przerwan
$0035   byte (1)    _mirqlvl    M$IRQLvl        Poziom przerwania
$0036   byte (1)    _mpriority  M$Prior         Priorytet przerwania
$0037   byte (1)    _mmode      M$Mode          Zakres mozliwosci urzadzenia
$0038   word (2)    _mfmgr      M$FMgr          Adres nazwy mod. obsl. plikow
$003a   word (2)    _mpdev      M$PDev          Adres nazwy sterownika
$003c   word (2)    _mdevcon    M$DevCon        Adres obszaru parametrow spec.
...
$0046   word (2)    _mopt       M$Opt           Wielkosc obszaru opcji
-----   --------    ---------   -------------   OBSZAR OPCJI
```

```
...
-----------------------------------------------------------------------

    /* Device descriptor module */
    typedef struct {
        struct modhcom  _mh;                    /* common header info */
            char *              _mport;         /* device port address */
            unsigned char       _mvector;       /* trap vector number */
            unsigned char       _mirqlvl;       /* irq interrupt level */
            unsigned char       _mpriority;     /* irq polling priority */
            unsigned char       _mmode;         /* device mode capabilities */
            short               _mfmgr;         /* file manager name offset */
            short               _mpdev;         /* device driver name offset */
            short               _mdevcon;       /* device configuration offset */
            unsigned short      _mdscres[4];    /* (reserved) */
            unsigned short      _mopt;          /* option table size */
            unsigned char       _mdtype;        /* device type code */
    } mod_dev;
```

Moduł sterownika urządzenia (*Device Driver*) zawiera w rozszerzeniu nagłówka adresy siedmiu procedur obsługi sprzętu, które się nań składaja (`Init, Read, Write, GetStat, SetStat, Term i Except`). *File Manager* odwołuje się do tych procedur za pośrednictwem ich wskaźników w czasie realizacji funcji wejścia/wyjścia.

```
    -----------------------------------------------------------------------------
    adres   wielkosc    nazwa w C   nazwa w asm
    -----   --------    ---------   -------------   ROZSZERZENIE NAGLOWKA (Desc)
    $0030   long (4)    _mexec      M$Exec          Wskaznik startu programu
    $0034   long (4)    _mexcpt     M$Excpt         Wskaznik obslugi TRAP
    $0038   long (4)    _mdata      M$Mem           Wielkosc obszaru danych
    $003C   long (4)    _mdinit     M$DInit         wskaznik procedury Init
    $003C   long (4)    _mdread     M$DRead         wskaznik procedury Read
    $003C   long (4)    _mdwrite    M$DWrite        wskaznik procedury Write
    $003C   long (4)    _mdgetstat  M$DGetStat      wskaznik procedury GetStat
    $003C   long (4)    _mdsetstt   M$DSetStt       wskaznik procedury SetStat
    $003C   long (4)    _mdterm     M$DTerm         wskaznik procedury Term
    $003C   long (4)    _mderror    M$DError        wskaznik procedury Except
    -----   --------    ---------   -------------   WLASCIWE PROCEDURY
    ...
    ----------------------------------------------------------------------------

    /* device driver module */
    typedef struct {
        struct modhcom _mh;     /* common header info */
        long    _mexec,         /* offset to execution entry point */
                _mexcpt,        /* offset to exception entry point */
                _mdata;         /* data storage requirement */
        short   _mdinit,        /* offset to init routine */
                _mdread,        /* offset to read routine */
                _mdwrite,       /* offset to write routine */
                _mdgetstat,     /* offset to getstat routine */
                _mdsetstt,      /* offset to setstat routine */
                _mdterm,        /* offset to terminate routine */
                _mderror;       /* offset to exception error routine */
    } mod_driver;
```

2.4 Ścieżki i pliki

Ścieżka (*path*) pozwala na dostęp na poziomie logicznym do danych, komend, statusu urządzenia z programu. Zamiast bezpośredniego komunikowania się z urządzeniem, program otwiera (*open*) ścieżkę dostępu przy pomocy odpowiedniego wywołania systemowego. System zwraca numer ścieżki, przy pomocy ktorego odbywają się wszystkie dalsze operacje (odczyt, zapis, sterowanie ...). Po zakończeniu dostępu program wykonuje operację zamknięcia ścieżki (*close*).
Ścieżka jest reprezentowana przez strukturę w pamięci zwaną deskryptorem ścieżki (*path descriptor*), która jest tworzona przy otwieraniu i usuwana przy zamykaniu ścieżki. Przy zakończeniu procesu jądro samo zamyka wszystkie otwarte w nim ścieżki.
Plik jest strukturą danych w urządzeniu mającym pamięć. Pozwala to na przechowywanie wielu zbiorów danych na jednym urządzeniu i ich modyfikowanie. Moduły obsługi plików są tworzone tak, by jak najbardziej upodobnić (z punktu widzenia programu) pliki na różnych typach urządzeń do siebie.
Nazwa ścieżki (*pathlist*) jest napisem określającym urządzenie i/lub plik. Jest ona używana przy otwieraniu ścieżki. Nazwa ścieżki może zawierać wiele elementów oddzielonych separatorami "/":
/d0 - katalog główny dysku opisanego deskryptorem **d0**;
/h1/SUPWA/ex - plik **ex** w podkatalogu **SUPWA** katalogu głównego dysku **h1**;
Jeśli nazwa ścieżki nie zaczyna się od "/", to jest uważana za ścieżkę względną, począwszy od bieżącego katalogu danych, lub programów.

2.5 Systemowe funkcje wejścia-wyjścia

I$Attach *Attach I/O Device* Przyłącza urządzenie zapewniając jego inicjację, pamięć statyczn/a, umieszczenie w tablicy.

I$Detach *Detach I/O Device* Odłącza urządzenie.

I$Dup *Duplicate Path* Powiela ścieżkę, dając lokalny numer ścieżki dla wcześniej otwartej.

I$Create *Create New File* Tworzy nowy plik i otwiera ścieżkę do niego.

I$Open *Open Existing File* Tworzony deskryptor ścieżki ma pola PD_COUNT i PD_CNT ustawione na 1, PD_MOD - zgodnie z podanym trybem. PD_USER ustawia się na numer grupy i użytkownika procesu. Po wywołaniu I$Attach do PD_DEV wpisuje się adres miejsca w tablicy urządzeń.

I$MakDir *Make Directory File* Podobnie, jak w I$Create, przy atrybytach *directory*.

I$ChgDir *Change Default Directory* Tymczasowo otwiera ścieżkę, wywołuje funkcję modułu obsługi plików i zamyka ścieżkę. Adres urządzenia jest wpisywany do P$DIO - w deskryptorze procesu (jako bieżący katalog danych lub programów, w zależności od atrybutu *execute*).

I$Delete *Delete File* Tymczasowo otwiera ścieżkę, wywołuje moduł obsługi plików i zamyka ścieżkę.

I$Seek *Change Current Position* Przestawia bieżący wskaźnik w pliku (w otwartej ścieżce).

I$Read *Read Data* Czyta do bufora dane bez ich modyfikowania z otwartej ścieżki.

I$Write *Write Data* Pisze z bufora dane bez ich modyfikowania do otwartej ścieżki.

I$ReadLn *Read Line of ASCII Data* Działa jak I$Read, lecz kończy czytanie na końcu linii (w OS–9 zazwyczaj CR=$13).

I$WritLn *Write Line of ASCII Data* Działa jak I$Write, lecz zakłada, że moduł obsługi plików kończy na CR i dokonuje prostych konwersji (SCF - LF po CR, pauza po nowej stronie i rozwijanie tabulacji).

I$GetStt *Get Path Status* Pozwala na dostęp do takich własności urządzeń we/wy, które nie są osiągalne w inny sposób. Wymaga otwartej ścieżki. Przy wywołaniu podaje sie numer funkcji, który jest specyficzny dla sterownika urządzenia.

I$SetStt *Set Path Status* Jak I$GetStt, ale dla zadawania parametrów urządzeniom i realizowania specyficznych funkcji.

I$Close *Close Path* Zamyka otwartą ścieżkę. PD_COUNT i PD_CNT są zmniejszane, jeśli żaden inny proces z niej nie korzysta - jądro (*kernel*) wywołuje funkcję *close* modułu obsługi plików.

2.6 Deskryptor ścieżki

Deskryptor ścieżki jest strukturą w pamięci zajmującą 256 bajtów. Jest zerowany przy tworzeniu. Pierwsza połowa jest używana do przechowywania zmiennych potrzebnych przy obsłudze ścieżki. Jej początek jest wspólny dla wszystkich ścieżek:

```
adres   rozmiar     nazwa
-----   --------    --------------
$000    word (2)    PD_PD           Systemowy numer sciezki
$002    byte (1)    PD_MOD          Flagi trybu
$003    byte (1)    PD_CNT          Licznik uzyc (stary)
$004    long (4)    PD_DEV          Adres w tablicy urzadzen
$008    word (2)    PD_CPR          ID aktualnego procesu
$00a    long (4)    PD_RGS          Adres ramki stosu procesu
$00e    long (4)    PD_BUF          Adres bufora danych
$012    word (2)    PD_USER         Numer grupy ...
$014    word (2)                    ... i ID uzytkownika
$016    long (4)    PD_Paths        Nastepna sciezka na urzadzeniu
$01a    word (2)    PD_COUNT        Licznik uzyc sciezki (nowy)
$01c    word (2)    PD_LProc        ID ostatniego procesu
$01e    long (4)    PD_ErrNo        Numer ostatniego bledu
$022    long (4)    PD_SysGlob      Wskaznik pamieci lobalnej systemu
$026    word (2)
$000    word (2)
$02a    ...
-----   --------    --------------  OBSZAR OPCJI
$080    ...         PD_OPT
...
------------------------------------------------------------------------
```

Dalsza część zależy od modułu obsługi plików. Druga połowa ($80 - $FF) jest obszarem opcji, który jest wypełniany kopią obszaru opcji deskryptora urządzenia podczas otwierania ścieżki. Jego struktura zależy od modułu obsługi plików (*file manager*).

3 Procesy w OS–9

Proces składa się z programu, który został uruchomiony i dotąd się nie zakończył, oraz obszaru danych i struktury w pamięci używanej przez system do obsługi tego procesu. Struktura ta jest deskryptorem procesu (*process descriptor*). Ponieważ w OS–9 regułą jest używanie czystego kodu jeden program może mieć równocześnie wiele wcieleń - procesów używających tego samego programu, ale oddzielnych obszarów danych. Każdy z procesów jest obsługiwany przez kernel dzięki oddzielnemu deskryptorowi, który zawiera informacje o procesie.

```
---------------------------------------------------------
adres rozmiar   nazwa      opis pola
----- --------  ------     --------------
$000  word (2)  P$ID       ID procesu
$002  word (2)  P$PID      ID procesu rodzicielskiego
...
$008  long (4)  P$sp       systemowy wskaznik stosu
$00c  long (4)  P$usp      wskaznik stosu uzytkownika
$010  long (4)  P$MemSiz   wielkosc uzywanej pamieci
$014  word (4)  P$User     Numer grupy i uzytkownika
$018  word (2)  P$Prio     priorytet
$01c  word (2)  P$State    status procesu
$020  byte (1)  P$QueuID   ID kolejki procesow
...
$026  word (2)  P$Signal   kod sygnalu
...
$030  long (4)  P$QueuN    wskaznik nastepnego procesu
$034  long (4)  P$QueuP    wskaznik poprzedniego procesu
$038  long (4)  P$Module   adres modulu glownego
...
----- --------  ---------- OBSZAR STOSU
      ... (800) StackRoom
                P$Stack    szczyt stosu
---------------------------------------------------------
```

Proces (zadanie) jest tworzony przez funkcję systemową F$Fork, która zwraca numer zwany identyfikatorem procesu (ID) jednoznacznie określający dany proces. Jest on używany przez inne funkcje systemowe do komunikowania się z procesem i zmieniania jego zachowania. Numer jest zwalniany po zakończeniu procesu i może być przydzielony innemu zadaniu. Nie jest jednak możliwe, by dwa procesy miały ten sam numer równocześnie. Procesu o numerze 0 nie ma, a numer jeden posiada proces systemowy.
W typowych aplikacjach czasu rzeczywistego wiele procesów pracuje równocześnie. Muszą one wymieniać między sobą dane, informacje sterujące i synchronizować się wzajemnie.
W danej chwili proces może być w jednym z kilku stanów:

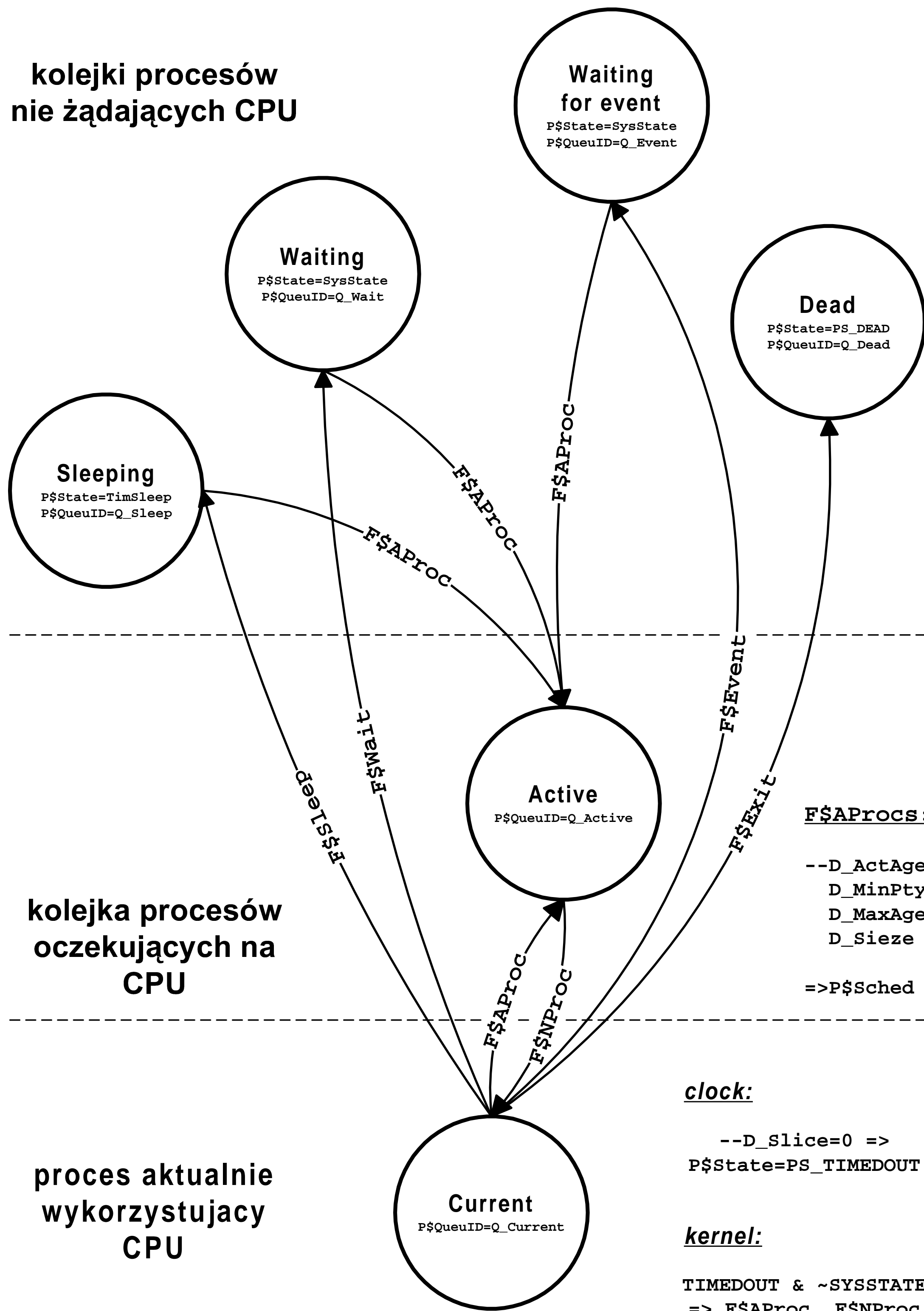
- **Active** - żądający czasu procesora
- **Waiting** - czekający na zakończenie procesu potomnego
- **Sleeping** - czekający przez określony czas, lub na zewnętrzne zdarzenie, lub na sygnał od innego procesu
- **Waiting for event** - czekający na zdarzenie wewnętrzne (semafor)
- **Debugged** - czekający na zezwolenie kontynuacji od procesu rodzicielskiego (debuggera)

- **Dead** - czekający na odebranie statusu wyjściowego przez proces rodzicielski (po zakończeniu)

Czas procesora jest dzielony pomiędzy procesy aktywne w jednostkach zwanych *time slices*. Osiąga się to dzięki cyklicznym przerwaniom zegarowym zwanym *ticks*. Zazwyczaj w OS–9 segment składa się z dwóch tików po 10 ms każdy.
Wykonywanie kolejno wszystkich procesów w krótkich segmentach czasu sprawia wrażenie równoległej pracy kilku programów. Długość segmentu dobiera się kompromisowo - dostatecznie małą, by stwarzać wrażenie równoległości, dostatecznie dużą, by nie tracić zbyt wiele czasu na przełączanie procesów. Dzięki przydzielaniu czasu tylko procesom aktywnym unika się marnowania go. System podziału czasu w OS–9 zapewnia równomierne przydzielanie czasu wszystkim procesom (z uwzględnieniem priorytetów), o ile programista nie zażąda inaczej. Specjalne cechy systemu podziału czasu pozwalają wpływać na sposób przydziału czasu, co jest szczególne ważne w systemach czasu rzeczywistego.
Zakończenie procesu następuje w wyniku wykonania przez program funkcji F$Exit, lub stwierdzenia błędu fatalnego (np. nieprzewidziany sygnał). Proces rodzicielski może przekazać wskaźnik obszaru parametrów, który będzie skopiowany do pamięci statycznej procesu potomnego.
Proces może również przekształcić się w inny, wykonujący inny program. Służy do tego funkcja F$Chain. Jest ona podobna do F$Fork, lecz proces rodzicielski zostaje zakończony, a jego deskryptor jest użyty do uruchomienia nowego procesu, który ma ten sam identyfikator.
Kernel zarządza procesami dzięki deskryptorom procesów. Są one zebrane w tablicy deskryptorów procesów, która zawiera adresy wszystkich deskryptorów. Identyfikator procesu (ID) jest indeksem w tej tablicy. Zerowe adresy w tablicy oznaczają brak procesu o tym ID. Tablica ta jest w miarę potrzeby powiększana (przepisywana do dwukrotnie większego obszaru).
Każdy proces ma początkowo rodzica - proces, który go uruchomił. Może on być osierocony przez zakończenie procesu rodzicielskiego przed zakończeniem potomka.
Proces, który kończy się nieosierocony, zwraca swój status wyjściowy procesowi rodzicielskiemu. Dzieje się to przez wykonanie w procesie rodzicielskim funkcji F$Wait. Zakończony (*dead*) proces może więc pozostawać w tablicy do czasu wykonania tej funkcji, lub zakończenia rodzica. Pozostawiany jest w tym przypadku tylko deskryptor, program i dane są zwalniane przez jądro OS–9. Jest to niezbędne dla zapewnienia poprawnej współpracy procesow w systemach wielozadaniowych, gdzie proces rodzicielski musi często znać status końcowy potomka.

3.1 Podział czasu w OS–9

Wielozadaniowść jest bardzo ważną cechą systemów czasu rzeczywistego i systemów wielodostępnych. Zwykle te dwa zastosowania wymagają drastycznie różnych systemów operacyjnych, ale obsługa wielozadaniowości w OS–9 pozwala je pogodzić bez rezygnowania z wymagań. Algorytm kolejkowania procesów firmy Microware jest prosty, elegancki, szybki i łatwy w obsłudze. Podstawową metodą jest karuzela (*round robin*), ale dostępne opcje pozwalają zmianę jego zachowania, w krańcowym przypadku do algorytmu czysto priorytetowego (jak w prostych systemach czasu rzeczywistego).
Funkcje obsługi procesów mieszczą się w jądrze (*kernel*). Ich celem jest udostępnianie wielu procesom czasu procesora. Jądro obsługuje listę procesów żądających czasu procesora (*active queue*). Każdy z procesów z tej kolejki otrzymuje czas procesora, o ile nie działa któryś z mechanizmów wywłaszczających. Proces bieżący (*current process*) nie znajduje się w kolejce. Zmienna globalna D_Proc wskazuje na jego deskryptor. W kolejce czekają te procesy, które żądają czasu procesora, ale go aktualnie nie dostały. Aby otrzymac czas procesora, proces musi zostać umieszczony w kolejce *active*. Usunięcie z tej kolejki może się dokonać wyłącznie przez wykonanie w procesie funkcji systemowej


kolejki procesów
nie żądających CPU
Waiting
for event
P$State=SysState
P$QueuID=Q_Event
Waiting
P$State=SysState
P$QueuID=Q_Wait
Dead
P$State=PS_DEAD
P$QueuID=Q_Dead
Sleeping
P$State=TimSleep
P$QueuID=Q_Sleep
F$AProc
F$AProc
F$AProc
F$Event
F$Sleep
F$Wait
F$Exit
Active
P$QueuID=Q_Active
F$AProcs:
--D_ActAge,
D_MinPty,
D_MaxAge,
D_Sieze
=>P$Sched
kolejka procesów
oczekujących na
CPU
F$AProc
F$NProc
clock:
--D_Slice=0 =>
P$State=PS_TIMEDOUT
proces aktualnie
wykorzystujacy
CPU
Current
P$QueuID=Q_Current
kernel:
TIMEDOUT & ~SYSSTATE
=> F$AProc, F$NProc

takiej jak czekanie na zdarzenie, uśpienie (*sleep*), lub zakończenie procesu przez kernel (*kill*). Przy uruchamianiu (*fork*) proces jest umieszczany w kolejce procesów aktywnych. przy pomocy funkcji F$AProc.
Kolejkowanie polega na umieszczaniu procesu w kolejce procesów aktywnych w zależności od priorytetu. Proces, który ma być uruchomiony jako następny znajduje się na czele kolejki. Dzięki temu przełączanie procesów jest szybkie. Usuwa się pierwszy proces z kolejki i czyni procesem bieżącym (funkcja F$NProc). Funkcja ta jest wywoływana tylko wtedy, gdy proces bieżący wywołał funkcję, która zawiesza jego pracę (*sleep*, *die*), lub przy powrocie do trybu użytkownika, gdy deskryptor procesu jest oznaczony jako przeterminowany (*timed out*). W tym ostatnim przypadku następuje umieszczenie tego procesu w kolejce *active*, jak przy uruchamianiu i dopiero potem - pobranie pierwszego w kolejce jako aktualnego. Możliwe jest więc wielokrotne kolejne przydzielenie odcinka czasu temu samemu procesowi. Jesli kolejka jest pusta, *time out* jest ignorowany dla oszczędzenia czasu przełączania procesów.
Procedura czasowa wywoływana przy każdym tiku zegara zmniejsza zmienną D_Slice, która oznacza ilosć tików do końca odcinka czasu (*slice*). Po jej wyzerowaniu, jądro ustawia flagę *timed out* w polu P$State deskryptora procesu bieżącego. Przełączenie odbywa się w chwili, gdy kernel ma wrócić do trybu użytkownika i zauważa *time out*. Wtedy wykonuje się fukcję F$AProc, by wstawić bieżący proces do kolejki, a następnie F$NProc, by uruchomić następny. Dzięki temu funkcje systemowe (wykonywane w trybie nadzorcy - *supervisor*) są niepodzielne (o ile nie wykonają *sleep*).
F$NProc usuwa proces z początku kolejki i czyni go bieżącym (D_Proc). Ustatwia D_Slice na wartość pobraną z D_TSlice (tiki na odcinek). Jeśli nie ma żadnych procesów do wykonania, procesor wykonuje instrukcję *stop*, co zatrzymuje jego pracę aż do przerwania. Zazwyczaj D_TSlice wynosi 2, by uniknąć przydzielania zbyt małych odcinków czasu (system nie rozróżnia jednostek czasu poniżej *tick*). Zmienne globalne podane są w Dodatku B.
Przydział czasu jest zarządzany przez kombinację czterech reguł:

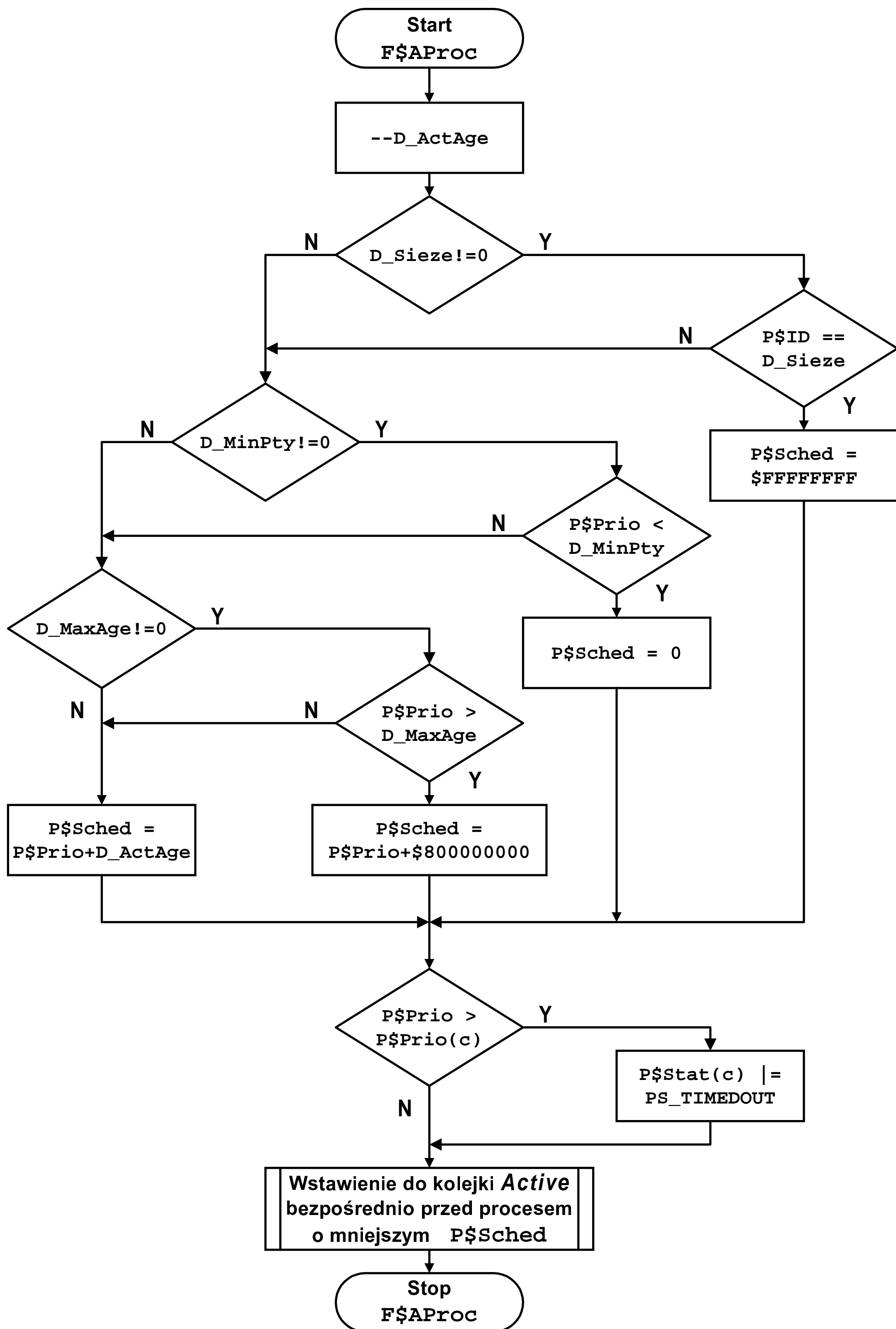
- karuzela (*round robin*). Czas procesora jest dzielony na odcinki, każdy z procesów aktywnych po kolei dostaje swój odcinek. Priorytety decydują o częstotliwości przydzielania odcinków.

- próg priorytetu (*minimum process priority*). Procesy o priorytecie niższym od progowego nie otrzymują odcinków czasu.

- wywłaszczanie priorytetowe. Proces o najwyższym priorytecie pozostaje bieżącym aż do własnej rezygnacji (*sleep*), lub uruchomienia procesu o wyższym priorytecie. Dotyczy to tylko procesów o priorytecie powyżej progu. Poniżej działa dalej karuzela (o ile chwilowo nie ma procesów aktywnych o priorytecie powyżej progu).

- pojedynczy proces. Programista przejmuje zarządzanie procesami dzięki mechanizmowi wywłaszczania. Jeden z procesów jest wyznaczany jako bieżący i pozostaje nim tak długo, dopóki nie zostanie zastąpiony innym, lub mechanizm ten nie zostanie wyłączony (D_Sieze).

Kernel obsługuje zmienną D_ActAge - wiek kolejki procesów aktywnych. Ma ona poczatkową wartość $7FFF0000 i jest zmniejszana o 1 przy każdym wywołaniu F$AProc. Funkcja ta wylicza stałą kolejkujcą (*scheduling constant*) dla procesu umieszczanego w kolejce. W normalnym przypadku jest ona sumą D_ActAge (po zmniejszeniu) i priorytetu procesu. Deskryptor procesu jest usuwany z kolejki, w której się dotąd znajdował i umieszczany w kolejce *active* bezpośrednio przed procesem o niższych wartosciach kolejkujących (P$Sched). Jesli dodatkowo proces ma wyższy priorytet od bieżącego, to bieżący jest przeterminowywany (*timed out*). Powoduje to uczynienie aktywnym procesu z czoła kolejki natychmiast po zakończeniu pracy w trybie nadzorcy. Jest to bardzo ważne w systemach czasu rzeczywistego.


Start
F$AProc
--D_ActAge
D_Sieze!=0
N
Y
P$ID ==
D_Sieze
N
Y
P$Sched =
$FFFFFFFF
D_MinPty!=0
N
Y
P$Prio <
D_MinPty
N
Y
P$Sched = 0
D_MaxAge!=0
Y
N
P$Prio >
D_MaxAge
N
Y
P$Sched =
P$Prio+D_ActAge
P$Sched =
P$Prio+$800000000
P$Prio >
P$Prio(c)
Y
N
P$Stat(c) |=
PS_TIMEDOUT
Wstawienie do kolejki Active
bezpośrednio przed procesem
o mniejszym P$Sched
Stop
F$AProc

Algorytm rotacyjny może być zmodyfikowany przez ustalenie progu priorytetu. Jeśli proces o priorytecie niższym niż wartość zmiennej globalnej D_MinPty jest umieszczany w kolejce *active* (F$AProc), to jego stała kolejkowania jest zerowana. W ten sposób trafia on na koniec kolejki. F$NProc sprawdza priorytet procesu, który ma być uruchomiony i oznacza go jako *timed out* w przypadku nieprzekraczania D_MinPty. Dodatkowo uruchamiana jest funkcja F$AProc, by umieścić proces w kolejce *active*, co jak poprzednio wyzeruje P$Sched i ulokuje proces na końcu kolejki. Po ustawieniu D_MinPty wszystkie procesy z kolejki dokończą swe wywołania systemowe i zostaną umieszczone na końcu.
Ustawienie D_MinPty powoduje zablokowanie wykonywania procesów o priorytecie poniżej progu. Przy obniżeniu progu funkcja F$SetSys służąca do zmiany zmiennych globalnych przegląda wszystkie procesy o zerowych polach P$Sched i wywołuje F$AProc umożliwiając uruchomienie procesów, których priorytety znalazły się powyżej nowego progu.
Zmienna D_MaxAge stanowi inny próg. Jesli nie jest zerem, to procesy o priorytetach poniżej progu są obsługiwane karuzelowo, a pozostałe otrzymują stałą kolejkujacą równą $80000000+P$Prio. W ten sposób zawsze poprzedzają w kolejce inne procesy, a pomiędzy sobą są uporządkowane według priorytetow. Dopóki w kolejce są procesy o priorytecie powyżej progu, procesy mniej ważne nie otrzymują czasu ($7FFF0000+P$Prio nie może przekroczyć $7FFFFFFF). F$SetSys przy zmianie D_MaxAge wywołuje F$AProc, by zaktualizować natychmiast kolejkę procesów.
Mechanizm wyłączania kolejkowania polega na ustawieniu w zmiennej D_Sieze numeru (ID) procesu, który ma być bieżącym. Gdy proces o tym ID jest uaktywniany (F$AProc), otrzymuje stałą $FFFFFFFF, co ustawia go na czele kolejki. Gdy ma dodatkowo wyższy priorytet, niż proces bieżący, to powoduje jego przeterminowanie i sam staje się bieżącym (jak przy innych trybach kolejkowania) dzięki funkcji F$NProc. Po zawieszeniu, lub zakończeniu tego procesu funkcja F$NProc nie uruchomi innych (jak w przypadku pustej kolejki). Programista może więc dowolnie sterować kolejnością wykonywania programów.
W pierwszej kolejności działa modyfikacja D_Sieze, następnie D_MinPty, a jako ostatnia - D_MaxAge.

3.2 Komunikacja między procesami w OS–9

3.2.1 Sygnały

Sygnały (*signals*) są przeznaczone do komunikacji pomiędzy niezależnymi, równoległymi procesami. Składają się one z identyfikatora procesu, dla którego są przeznaczone i z własnego numeru. Pewne sygnały są w systemie wstępnie zdefiniowane:

- 0 - kill - bezwzględne zakończenie procesu;
- 1 - wake-up - wznowienie procesu bedącego w stanie sleep;
- 2 - abort - przerwanie procesu ostatnio korzystającego z we/wy na terminal (Ctrl-E) - najczęściej exit(2);
- 3 - interrupt - (Ctrl-C) - najczęściej exit(3);
- 4 - hang-up - wysyłany przez SCF przy utracie łączności modemowej;
- 5-255 - zarezerwowane dla dalszych zastosowań systemowych;
- 256-65535 - dla użytkowników.

Proces, który odbierze sygnał inny niż wake-up zostanie zakonczony, o ile nie jest wyposażony w procedurę przechwytywania sygnałów (*kill* jest niemożliwy do przechwycenia, ale może być wysłany tylko przez właściciela procesu, lub super-usera (grupa 0). Następujące funkcje systemowe są przewidziane do obsługi sygnałów:

- F$Send - wysłanie sygnału;
- F$Icpt - zainstalowanie procedury przechwytywania sygnałów;
- F$Sleep - deaktywacja procesu do wyczerpania czasu, lub odebrania sygnału;
- F$SigMask - ustawianie maski sygnałów dla procesu.

Sygnał w OS–9 stanowi bardzo krótką wiadomość przesyłaną z jednego procesu do innego. Jest to mechanizm programowy, w odróżnieniu od przerwań, które są związane ze sprzętem. Podobieństwo polega jedynie na zdolności do asynchronicznego zmieniania przebiegu programu i możliwości maskowania.
Do wysyłania sygnałów służy funkcja F$Send. Podaje się jej numer (ID) procesu - adresata (są też sygnały wysyłane do wszystkich procesow - broadcasts). Proces wysyłający sygnał musi mieć taki sam numer grupy i użytkownika (ID), jak proces - adresat, lub być własnością administratora (grupa 0). Odebranie sygnału działa na proces dwojako:

- uaktywnia go (o ile dotąd nie był w kolejce *active*),
- przy najbliższym uruchomieniu w trybie użytkownika powoduje wykonanie procedury obsługi sygnałów (*signal handler function*).

Sygnał jest jedynym asynchronicznym sposobem komunikacji między procesami w OS–9. Ponadto, pozwala on oczekiwać na dostępność zasobów bez sprawdzania (*polling*), gdyż odebranie sygnału uaktywnia proces.
Sygnały otrzymywane przez proces są kolejkowane (może ich być wiele pomiędzy kolejnymi odcinkami czasu pracy procesu). Pole P$Signal w deskryptorze procesu zawiera kod ostatnio otrzymanego sygnału, lub zero (jeśli nie ma sygnałów).
Sygnał o kodzie 0 (S$Kill) nie jest kolejkowany. Powoduje natomiast ustawienie flagi *condemned* w polu P$State deskryptora procesu. Przy próbie uruchomienia w trybie użytkownika powoduje to bezwarunkowe zakończenie procesu.
Sygnał o kodzie 1 (S$Wake) służy do uaktywniania uśpionego sterownika urządzenia przez procedurę obsługi przerwania, nie powinien być używany w trybie użytkownika.
Funkcja F$SigMask służy do odraczania obsługi sygnałów. Zwiększa, zmniejsza, lub zeruje pole maski sygnałów (P$SigLvl) w deskryptorze procesu. Jeśli pole to nie jest zerowe, to sygnały są kolejkowane i powodują wprawdzie uaktywnienie procesu, ale nie uruchamiają procedury ich obsługi aż do wyzerowania tego pola. Parametr F$SigMask może być równy -1, 0, lub 1, co powoduje odpowiednio zmniejszenie (ale nie przy zerowej wartości), wyzerowanie, lub zwiększenie (maksymalnie do 255).

3.2.2 Łącza

Łącze (*pipe*) jest buforem typu FIFO (*first-in-first-out*). Jeden lub więcej procesów może pisać do tego bufora przy pomocy standardowych funkcji wyjścia, jeden lub więcej procesów może zeń czytać przy pomocy standardowych funkcji wejścia. Dane stanowią strumień bajtów, są czytane w kolejności, w jakiej nastąpił zapis. Każdy odczytany bajt jest usuwany z bufora (łącza), gdy wszystkie wpisane bajty są odczytane, bufor jest pusty. Dalszy odczyt jest możliwy dopiero po wpisaniu dalszych danych. Łącze ma ograniczoną pojemność. Gdy różnica pomiędzy ilościami wpisanych i odczytanych bajtów ją przekracza, dalszy wpis jest niemożliwy. Zazwyczaj mamy do czynienia z jednym procesem czytajacym z łącza, mimo wielu procesów piszacych doń.
Łącza w OS–9 są obsługiwane przez moduł obsługi plików **pipeman**. Jako bufory w pamięci nie są zależne od sprzętu i nie wymagają sterowników urządzeń. Mogą być one tworzone jako nazwane, lub nienazwane.
Łącze nienazwane jest tworzone przez:

```
path=create(``/pipe'',S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);
```

Łącze nazwane tworzy się podając drugi element nazwy ścieżki w funkcji `create()`:

```
path=create(''/pipe/blabla'',S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);
```

Łącza mogą służyć zarowno do przesyłania danych pomiędzy procesami, jak i do synchronizowania procesów. Proces czytający z łącza jest zawieszany po jego opróżnieniu (chyba że nie jest otwarta żadna ścieżka w trybie zapisu, kiedy to zwracany jest błąd ”koniec pliku”), aż do pojawienia się nowych danych w buforze. Proces piszący do łącza jest zawieszany po jego zapelnieniu (o ile są otwarte ścieżki w trybie odczytu, bowiem w przeciwnym przypadku zwracany jest błąd E_WRITE, by zapobiec blokadzie).
Łącza nazwane mogą istnieć mimo braku otwartych ścieżek, o ile zawierają dane. Proces piszący do zapełnionego łącza nazwanego jest zawieszany w każdym przypadku, w nadziei pojawienia się nowych otwartych ścieżek. Łącze nazwane jest automatycznie usuwane gdy jest puste i nie ma otwartych ścieżek.
Shell używa łącz nienazwanych do tworzenia potoków:

```
> dir -ud | grep -v "/$" | del -z
```

3.2.3 Zdarzenia (*events*)

Przy pomocy zdarzenia (*event*) można w OS–9 synchronizować procesy i przekazywać niewielkie ilości danych (wartość zdarzenia). W odróżnieniu od sygnałów, zdarzenia nie mogą zmieniać przebiegu programu (jak przerwania i sygnały), nie mogą uaktywniać procesów nie oczekujących na nie. Mają jednak liczne zalety wynikające z elastyczności (mogą przyjmować wiele wartości, są publiczne, są nieulotne).
Zdarzenie jest tworzone przez funkcję systemową, której dostarcza się nazwę i wartość początkową. Jądro umieszcza zdarzenie w tablicy po uprzednim sprawdzeniu, że nie ma innego o tej samej nazwie (do 12 znaków). Zdarzeniu jest przyporzadkowywany numer (ID) będący długim słowem, którego bardziej znaczącą połową jest numer kolejny z D_EvID (po zwiększeniu), a mniej znaczącą - indeks zdarzenia w tablicy. Tablica zdarzeń jest dynamicznie rozszerzalna, nie ma ograniczenia na ilość zdarzeń. Funkcja zwraca ID, lub błąd (gdy zdarzenie o podanej nazwie już istnieje).
Element tablicy zdarzen:

```
$00 word (2)  Event ID      - unikalny numer zdarzenia w systemie;
$02 byte (12) Event Name    - max. 11-znakowa nazwa zdarzenia;
$0e long (4)  Event Value   - 4-bajtowa wartosc zdarzenia;
$12 word (2)  Wait Incr.    - przyrost wartosci przy budzeniu;
$14 word (2)  Signal Incr.  - automatyczny przyrost wartosci;
$16 word (2)  Link Count    - ilosc uzyc zdarzenia;
$18 long (4)  Next Event    - wskaznik do nastepnego zdarzenia;
$1c long (4)  Prev. Event   - wskaznik do poprzedniego zdarzenia;
```

Po utworzeniu zdarzenia inne procesy mogą się doń przyłączyć (*link to event*) i uzyskać ID po podaniu nazwy (zwiększa to licznik użyć zdarzenia). Po zakończeniu korzystania ze zdarzenia proces może je zwolnić (*unlink from event*).
Proces znający ID zdarzenia może:

- odczytać jego wartość
- zmienić tę wartość
- czekać na wpadnięcie tej wartości do określonego przedziału

Przy każdej zmianie wartości zdarzenia jądro sprawdza, czy nowa wartość pasuje do przedziałów zadeklarowanych przez procesy oczekujące na to zdarzenie. W tym przypadku jądro budzi proces zwracając wartość zdarzenia, a następnie modyfikuje tę wartość o *wakeup increment*.
Wartość zdarzenia może zostać zmieniona przez:

- ustawienie bezwzględnej wartości (set absolute)
- dodanie (ze znakiem) wartości (set relative)
- dodanie przyrostu automatycznego (signal increment)
- chwilowe ustawienie bezwzględnej wartości (pulse event)

W tym ostatnim przypadku jądro nadaje nową wartość zdarzeniu tylko w celu obudzenia czekających procesow (o ile są), po czym przywraca pierwotną wartość.
Jądro systemu zawiera jedną funkcję F$Event pozwalającą operować zdarzeniami. Odpowiednia akcja jest powodowana dzięki kodom operacji:

```
Ev$Link     - uzycie zdarzenia (wedlug nazwy);
Ev$UnLnk    - zwolnienie zdarzenia;
Ev$Creat    - utworzenie zdarzenia;
Ev$Delet    - usuniecie istniejacego zdarzenia;
Ev$Wait     - oczekiwanie na zdarzenie;
Ev$WaitR    - oczekiwanie na wzgledna wartosc zdarzenia;
Ev$Read     - odczytanie wartosci licznika bez oczekiwania;
Ev$Info     - odczytanie parametrow zdarzenia;
Ev$Pulse    - sygnalizacja wystapienia zdarzenia;
Ev$Signl    - sygnalizacja wystapienia zdarzenia;
Ev$Set      - ustawienie licznika i sygnalizacja zdarzenia;
Ev$SetR     - wzgledne ustawienie licznika i sygnalizacja;
```

Najprostszą formą synchronizacji procesów moze byc *pulsing*. Jeden proces tworzy zdarzenie, drugi się przyłącza. Wartość początkowa zdarzenia - 0, przyrost przy budzeniu - 0. Proces oczekujacy ustawia w wywołaniu Ev$Wait przedział od 1 do 1, a proces synchronizujący wywołuje Ev$Pulse z wartością 1. Powoduje to zbudzenie procesu oczekującego na zdarzenie bez zmieniania wartości zdarzenia (pozostaje ona zerowa). Może się tu jednak zdarzyć, że proces zacznie oczekiwać na zdarzenie już po jego wystąpieniu, co spowoduje blokadę (zdarzenia nie są buforowane).
Bezpieczna forma synchronizacji procesów wymaga wzajemnych potwierdzeń (*handshake*). Zdarzenie jest tworzone jak poprzednio, proces oczekujący ustawia taki sam przedział, ale proces synchronizujący ustawia wartość zdarzenia na 1 (Ev$Set) i czeka na jego wyzerowanie (Ev$Wait). Proces synchronizowany (po wykonaniu odpowiednich operacji - na przykład przeczytaniu danych umieszczonych w pamięci przez proces synchronizujący) ustawia wartość zdarzenia na 0 i budzi w ten sposób proces synchronizujący.
Użycie zdarzenia jako semafora regulującego dostęp do niepodzielnego zasobu polega na zmuszeniu procesu zgłaszającego żadanie do poczekania na zwolnienie zasobu. Zdarzenie tworzy się z wartością 0, przyrostem przy budzeniu 1 i przyrostem sygnałowym (automatycznym) -1. Proces żądający zasobu czeka na zerową wartość zdarzenia. Przyrost przy budzeniu powoduje ustawienie wartości 1, blokując dostęp dla innych, podobnych procesów. Po zakończeniu korzystania z zasobu proces zmniejsza wartość zdarzenia (Ev$Signal), co powoduje obudzenie kolejnego z oczekujących procesów.

3.2.4 Alarmy

Alarmy pozwalają zsynchronizować proces z czasem rzeczywistym odmierzanym przez zegar systemowy (moduł **clock**) w przerwaniach. Alarm instaluje się przy pomocy funkcji F$Alarm. Powoduje ona wysłanie (w określonej chwili) sygnału o określonym kodzie do procesu wywołującego. Są dwa typy alarmów: jednorazowe i cykliczne (okresowe). Alarm jednorazowy powoduje wysłanie sygnału po upływie określonej ilości tików (alarm względny), lub przy określonym stanie zegara (data, czas) - alarm bezwzględny. Alarm

cykliczny powtarza wysyłanie sygnału z żądanym okresem aż do skasowania alarmu, lub zakonczenia procesu.
Można ustawić wiele alarmów równocześnie. F$Alarm zwraca unikalny numer alarmu (ID), który może służyć np. do jego usunięcia (funkcja F$Alarm). Usunięcie alarmu z ID=0 powoduje usunięcie wszystkich alarmów należących do danego procesu (np. przy zakończeniu procesu).
Alarmy nie są wykonywane bezpośrednio przez procedurę obsługi przerwania zegarowego. W celu wysłania alarmu budzony jest proces systemowy (o numerze 1, niewidoczny w `procs`). Ma on najwyższy z możliwych priorytetow (65535), więc zostanie uruchomiony przed wszystkimi procesami w trybie użytkownika. Takie rozwiązanie powoduje, że procedura obsługi przerwania zegarowego nie trwa zbyt długo i inne funkcje systemowe mogą być wykonywane, a równocześnie wysłany do procesu sygnał dociera (z punktu widzenia użytkownika) równie szybko.

4 Programy użytkowe OS–9

Programy użytkowe dostępne dla OS–9 pochodzą z różnych źródeł. Część z nich (komendy podstawowe, asembler, debuggery itp.) stanowią programy firmowe Microware. Istotnym uzupełnieniem jest oprogramowanie pochodzące od innych producentów i z *public domain*. Warto również podkreślić, że firmy dostarczające sprzęt wyposażają go w niezbędne składniki programowe (*drivery*, *file managery*, przykłady aplikacji).

4.1 Podstawowe komendy OS–9

Microware dostarcza wraz z systemem w każdej z wersji zestaw podstawowych programów użytkowych pozwalających zarządzać systemem:

- attr - odczyt i zmiana atrybutów pliku
- backup - duplikowanie dysku
- binex - zamiana pliku na postać szesnastkową (S-rekordy)
- build - tworzenie krótkich plików tekstowych
- cmp - porównywanie plików
- code - wyświetlanie szesnastkowych kodów klawiszy
- copy - kopiowanie plików
- count - zliczanie znaków, słów i linii w pliku
- date - wyświetlanie daty i czasu
- dcheck - sprawdzanie poprawności katalogu/dysku
- deiniz - odłączenie urządzenia
- del - kasowanie plików
- deldir - kasowanie katalogów
- devs - wyświetlanie tablicy zainicjowanych urządzeń we/wy
- dir - wyświetlanie zawartości kartoteki
- dsave - kopiowanie poddrzewa katalogów

- dump - szesnastkowe wyświetlanie zawartości pliku
- echo - wysyłanie tekstu na ekran
- edt - edytor liniowy
- exbin - zamiana S-rekordów na postać binarną
- fixmod - odtworzenie sum kontrolnych i CRC modułu
- format - formatowanie dysków
- free - wyświetlanie wolnego miejsca na dysku
- grep - przeszukiwanie plików według wzorca
- help - wyświetlanie informacji o komendach
- ident - wyświetlanie informacji o modułach
- iniz - inicjowanie urządzeń we/wy
- link - przyłączanie modułu w pamięci
- list - wyświetlanie zawartości pliku
- load - ładowanie modułów z pliku do pamięci
- login - włączanie się do systemu (wielodostęp)
- makdir - tworzenie katalogu
- mdir - wyświetlanie kartoteki modułów
- merge - łączenie plików na wyjście standardowe
- mfree - wyświetlanie wolnego miejsca w pamięci
- pd - wyświetlanie bieżącej ścieżki danych
- pr - wyświetlanie pliku z formatowaniem
- procs - wyświetlanie aktualnych procesów
- qsort - szybkie sortowanie pliku w pamięci
- rename - zmienianie nazwy pliku
- save - składowanie modułów pamięciowych do plików
- shell - powłoka - interfejs użytkownika, język komend
- sh - powłoka - interfejs użytkownika, język komend
- sleep - zatrzymanie procesu na zadany czas, lub do przerwania
- tee - kopiowanie wejścia na kilka ścieżek wyjściowych
- tmode - wyświetlanie i ustawianie parametrów terminala
- touch - aktualizacja daty dostępu do pliku
- tr - zamiana znaków w pliku na inne (filtr)
- umacs - edytor ekranowy MicroEMACS 3.6 (mały)

- emacs - edytor ekranowy MicroEMACS 3.10 (rozbudowany)
- unlink - odłączanie modułów pamieciowych
- xmode - zmiana parametrów urządzenia znakowego

Każda z wymienionych komend podaje swój skrócony opis (składnia, opcje) po wywołaniu z opcją `-?`. Czasem więcej informacji można uzyskać po wywołaniu:

```
help <komenda>
```

4.2 Środowisko programowania w OS–9

Tworzenie programu składa się z ciągu operacji powtarzanych aż do uzyskania zadawalającego wyniku:

- tworzenia (edycji) źródła,
- kompilacji/asemblacji do plików relokowalnych (*ROF - relocatable object file*),
- łączenia (*link*) ROF-ów w moduł programowy,
- testowania przy pomocy debuggera.

Narzędzia do tworzenia oprogramowania w OS–9 dostarczane przez Microware to:

- `cc` egzekutor kompilatora C
- `cpp` preprocesor C
- `c68` kompilator C dla 68000 i 68010
- `c68020` kompilator C dla 68020, 68030 i68040
- `o68` optymizator asemblera
- `r68` asembler dla 68000 i 68010
- `r68020` asembler dla 68020, 68030 i 68040
- `l68` linker
- `make` automatyczny zarządca kompilacji
- `debug` symboliczny debugger asemblera
- `srcdbg` symboliczny debugger C
- `sysdbg` debugger procesów w trybie systemowym

4.2.1 Kompilator

Kompilacja C składa się z trzech faz:

- wstępne przetwarzanie (`cpp`),
- kompilacja (`c68, c68020`),
- optymalizacja (`o68`).

Preprocesor tworzy roboczy plik źródłowy z rozwiniętymi makrami (`#define`), wstawkami (`#include`) i uwzględnieniem kompilacji warunkowej (tt #ifdef...).
Kompilator tłumaczy program w C na język asemblera (opcja -a w `cc` pozwala zakończyć na tym etapie).
Optymizator przegląda źródłowy program w języku asemblera w poszukiwaniu ciągów instrukcji, które można poprawić i zmienia je.
`Cc` i `make` używają pewnych konwencji w nazwach plików (rozszerzeń nazwy po ”.”). Część nazwy przed ostatnią kropką zwie się *root*. Tworzone pliki zachowują główną część nazwy (*”root”*) i otrzymują określone rozszerzenie:

```
".c"    zrodlo w C
".a"    zrodlo asemblerowe
".r"    plik relokowalny (ROF) - wynik asemblacji
      wynikowy modul programowy - wynik linkowania
".h"    pliki wlaczane do C
".d"    pliki wlaczane do asemblera
".l"    biblioteki ROF-ow
```

Biblioteki dołączane przez linker są pobierane z /dd/LIB i zależą od opcji cc:

```
      clibn.l, math.l, sys.l;
-x"    clib.l, sys.l;
-i"    cio.l, clibn.l, math.l, sys.l;
-ix"   cio.l, clib.l, sys.l;
```

Jako pierwszy moduł używany jest *cstart.r* (z /dd/LIB). Jest on odpowiedni tylko dla programów, dla modułów wynikowych innych typów (*trap, fman, ...*) trzeba go zastąpić kodem własnoręcznie napisanym w asemblerze.
Linker tworzy moduł wynikowy przez połączenie kilku modułów relokowalnych (ROF-ów). Moduły relokowalne mogą zawierać kod, definicje symboli, definicje pamięci statycznej. Mogą one być dostarczane jako pliki ”*.r”, lub biblioteki ”*.l”. Biblioteka zawiera kilka ROF-ów połączonych przez `merge`. Różnica polega na tym, że linker włącza do modułu wynikowego wszystkie ROF-y podane w linii wywołania (*.r), oraz tylko te ROF-y z bibliotek (*.l), które są niezbędne do zdefiniowania wszystkich nazw. Moduły ROF są włączane w kolejności wywołania w linii komendy l68. Moduły z bibliotek włącza się po wszystkich modułach ”*.r”. Podczas pracy linker buduje i aktualizuje tablicę nazw, które muszą być zdefiniowane. Przeglądanie bibliotek jest jednorazowe, moduły ROF nie definiujące potrzebnych nazw są pomijane. Jeśli ROF definiuje którąś z potrzebnych nazw, wszystkie jego nazwy publiczne (odwołania i definicje) są dołączane do tablicy.

4.2.2 Make

`Make` pozwala dokonywać automatycznie odpowiednich zmian w całej grupie plików według zadeklarowanych zależności pomiędzy nimi. Określa, czy któryś z plików wymaga uaktualnienia przez porównanie czasu jego ostatniej modyfikacji z takim samym czasem dla wszystkich plików, od których jest on uzależniony. Najczęściej jest wykorzystywany do kompilacji programów napisanych w językach wysokiego rzędu, ale moze też służyć do innych celów, gdy pliki wynikowe zależą od źródłowych, a procedury ich tworzenia są ustalone.
Opis tworzenia wyników dla `make` jest zawarty (domyślnie) w pliku o nazwie `makefile` w bieżącym katalogu danych. Można podać inną nazwę tego pliku w linii wywołania `make` (opcja -f). W *makefile* wystepują trzy typy zapisów:

- zależności - określające powiązania plików wynikowych ze źródłowymi:

```
<plik\_wynikowy>:[[<plik>],<plik>]
```

(rozpoczynające się w pierwszej kolumnie)

- komendy - podające sposób tworzenia pliku wynikowego (zaczynają się od znakow białych); dopuszczalna jest dowolna ilość dowolnych komend OS–9;
- komentarze - linie zaczynające się "*" lub "#" są ignorowane.

Linie dłuższe niż 256 znakow mogą być kontynuowane przy pomocy znaku "́ na końcu kontynuowanej linii.
Make czyta cały plik `makefile` i tworzy drzewo uzależnień na podstawie znalezionych zależności. W drugim przebiegu dodaje wbudowane zależności pomiędzy plikami relokowalnymi a zródłowymi, które nie zostały podane jawnie. W trzecim przebiegu sprawdzane i porównywane są daty plików według drzewa uzależnień. Jeśli data któregoś ze źródeł jest nowsza niż pliku wynikowego, to wykonywane są komendy podane po zależności (czas jest pamiętany w plikach z dokładnością do 1 minuty!).

4.2.3 Debuggery

`Debug` jest programem symbolicznego debuggera na poziomie języka asemblera. Pozwala uruchamiać i poprawiać programy pracujące w trybie użytkownika przy pomocy specjalnych funkcji systemowych. Proces poddawany uruchamianiu i testowaniu istnieje w systemie jak w czasie normalnej pracy, ale jest uruchamiany dopiero po odpowiednim wywołaniu systemowym wykonanym przez proces rodzicielski (*debug*), co pozwala np. ustawić pułapki (*breakpoints*). `Debug` próbuje załadować tablicę symboli (**.stb*) o tej samej nazwie głównej, co uruchamiany program. Poszukuje kolejno w katalogu *exec*, a potem według zmiennej środowiskowej PATH. Przed przeszukaniem każdego katalogu sprawdzany jest podkatalog STB (o ile istnieje). Jeśli plik *.stb nie zostanie znaleziony, to niemożliwe jest korzystanie z nazw symbolicznych, ale *debug* pracuje poprawnie.
Debug pozwala na :

- pułapki programowe,
- podgląd/modyfikację pamięci,
- podgląd/modyfikację rejestrów procesora,
- deasemblację kodu z pamięci,
- uruchamianie procesów (*fork*),
- przyłączanie się (*linking*) do modułów,
- sterowanie uruchamianiem programu.

`Srcdbg` działa podobnie, ale dodatkowo używa pliku **.dbg* tworzonego przez kompilator, który pozwala powiązać kod asemblerowy ze źródłem programu w C, a zatem wykonywać program krokowo według źródła w C, z wykorzystaniem nazw zmiennych i operacji na nich (obliczanie wyrażeń). Debugger źródłowy jest wyposażony w instrukcję użytkowania dostępną w czasie pracy dzięki komendzie `help`.

Bibliografia

[1] Paul S. Dayan, *The OS–9 Guru*, ISBN 0 9519228 0 7, Galactic Industrial Limited, Durham, 1992.

[2] Peter Dibble, *OS–9 Insights*, ISBN 0 918035 0105, Microware systems Corporation, Des Moines, 1988.

[3] *OS–9 Technical I/O Manual*, OIO68NA68MO, Microware Systems Corporation, Des Moines, 1990

[4] *OS–9 Technical Manual*, OST68NA68MO, Microware systems Corporation, Des Moines, 1989.

[5] *The OS–9 Catalog*, OS968NA68SL, Microware systems Corporation, Des Moines, 1990.

[6] *The OS–9 Hardware & Software Sourcebook*, SHD68NA68SL, Microware systems Corporation, Des Moines, 1989.

[7] *Using Professional OS–9*, OSU68NA68MO, Microware systems Corporation, Des Moines, 1989.

[8] Marek Wnuk, *OS–9 – modułowy, wielozadaniowy system czasu rzeczywistego*, Raport ICT SPR 31/94, Wydawnictwo Politechniki Wrocławskiej, Wrocław, 1994.