POLITECHNIKA KRAKOWSKA
im. Tadeusza Kościuszki

~$ Systemy operacyjne
~$ Przewodnik po laboratorium

~$ Anna Jasińska-Suwada
~$ Stanisława Plichta

Kraków 2010

POLITECHNIKA KRAKOWSKA
im. Tadeusza Kościuszki

~$ Systemy operacyjne

~$ Przewodnik po laboratorium

~$ Anna Jasińska-Suwada
~$ Stanisława Plichta

Kraków 2010

Spis treści

Wstęp

Niniejsza publikacja stanowi materiał do laboratorium z przedmiotu Systemy Operacyjne. Jest to kontynuacja skryptu Politechniki Krakowskiej *Przewodnik do ćwiczeń z przedmiotu: systemy operacyjne*, którego autorkami są Anna Jasińska-Suwada i Stanisława Plichta. Wszystkie zamieszczone w nim ćwiczenia i przykłady odnoszą się do systemu operacyjnego Linux. Zakres tematyczny skryptu obejmuje: uzupełniające przykłady i ćwiczenia dotyczące wyszukiwania plików oraz informacji zawartych w plikach; rozszerzone możliwości definiowania praw dostępu do plików – Acces Controll List; skrypty powłoki bash; narzędzie wspomagające pisanie skryptów – AWK; zastosowanie funkcji systemowych do zarządzania procesami, wątkami, wykonywania operacji na plikach z poziomu języka C; mechanizmy synchronizacji procesów – potoki, kolejki FIFO, semafory, kolejki komunikatów, pamięć dzielona; mechanizmy synchronizacji wątków.

Omówione w skrypcie mechanizmy synchronizacji procesów mają zastosowanie we współbieżnym wykonywaniu procesów w obrębie jednego systemu operacyjnego. Procesy te współdzielą zasoby pojedynczego komputera, współpracują ze sobą oraz współzawodniczą o niepodzielne zasoby.

Procesy wykonywane współbieżnie mogą być realizowane równolegle (jednocześnie), gdy system wyposażony jest w większą liczbę procesorów. Możemy tu wyróżnić maszyny ściśle powiązane, działające pod kontrolą jednego systemu operacyjnego, wyposażone we wspólną pamięć (nazywane **wieloprocesorami**, w których wirtualna przestrzeń adresowa dostępna jest dla wszystkich procesorów centralnych), a także maszyny luźno powiązane (multikomputery – zbiory procesorów wyposażonych w lokalne pamięci tworzące **systemy rozproszone**).

W wypadku równoległej architektury wieloprocesorów (SMP – *Symmetric Multiprocesor*), kiedy wszystkie procesory mają dostęp do całej pamięci operacyjnej (UMA – *Uniform Memory Access*), można stosować opisane w skrypcie mechanizmy synchronizacji, a kompilatory mogą być ponadto wyposażone w opcje zrównoleglenia kodu programu.

W systemach z pamięcią rozproszoną (NUMA – *Non Uniform Memory Access*) każda jednostka ma swoją lokalną pamięć.

Maszyny luźno powiązane mogą tworzyć **sieciowy system operacyjny** lub **prawdziwy rozproszony system operacyjny**. W sieciowym systemie operacyjnym użytkownicy mają dostęp do zdalnych zasobów za pośrednictwem sieci Internet, a każda maszyna jest niezależna i pracuje pod kontrolą własnego systemu operacyjnego. Współbieżne wykonywanie procesów w takim środowisku możliwe jest m.in. dzięki interfejsowi gniazd. System operacyjny Linux wyposażony jest

w interfejs (*socket interface*), który służy komunikacji pomiędzy procesami działającymi na różnych komputerach i umożliwia implementację rozproszonych systemów klient–serwer.

Rozproszone systemy operacyjne umożliwiają użytkownikom traktowanie luźno powiązanego sprzętu jako wirtualnego jednolitego systemu. Jeden system operacyjny kontroluje pracę wchodzących w jego skład nieautonomicznych komputerów. Zadaniem systemu operacyjnego jest m.in. kontrolowanie przemieszczania obliczeń i procesów. Rozproszone systemy operacyjne opierają się na mikrojądrach (*microkernels*), czyli jądrach świadczących minimalny zbiór usług, na bazie których można budować pozostałe potrzebne usługi.

Podstawą komunikacji międzyprocesowej w **systemach rozproszonych** jest mechanizm przekazywania komunikatów i oparty na nim mechanizm zdalnego wywołania procedury (RPC – *Remote Procedure Call*). Najpopularniejszym środowiskiem programowania równoległego jest protokół bazujący na przesyłaniu komunikatów (MPI – *Message Passing Interface*). Można z niego korzystać zarówno na komputerach wieloprocesorowych, jak i w systemach rozproszonych. Więcej informacji na temat systemów rozproszonych można znaleźć w pracach [2, 13].

Dla przejrzystości i lepszej czytelności tekstu w skrypcie zastosowano następujące czcionki:

Czcionka	Zastosowanie	Przykład
`Courier New`	polecenia systemowe	`grep`
Arial	nazwy plików	/home/janek/zrodla

W skrypcie zamieszczono rysunki, tabele, algorytmy, kody programów i przykłady ich uruchomień. Spisy wszystkich kategorii obiektów z uwzględnieniem numerów rozdziałów znajdują się na końcu książki.

1. Przeszukiwanie systemu plików[1]

Do przeszukiwania linuksowego systemu plików służą polecenia systemowe: `grep` i `find`. Jeśli poszukiwane są pliki według ich atrybutów zawartych w *i*-węźle pliku, a więc takich, jak np. rozmiar, nazwa, prawa dostępu, właściciel, należy zastosować polecenie `find`. W wypadku poszukiwania lub przeszukiwania plików ze względu na ich zawartość używamy polecenia `grep`. Niektóre polecenia złożone mogą wymagać obu komend.

1.1. Przeglądanie zawartości plików – polecenie `grep`

Polecenie `grep` jest uniwersalnym programem przeznaczonym do wyszukiwania w pliku wierszy zawierających określony wzorzec. Dane ze strumienia wejściowego lub z pliku wejściowego czytane są wierszami i wypisywane są te wiersze, które zawierają podany wzorzec.

```
grep opcje wzorzec plik
```

Uniwersalność programu `grep` bierze się z możliwości tworzenia złożonych wzorców za pomocą znaków uogólniających. Znaczenie wybranych metaznaków:

`.` – jeden dowolny znak,
`[a-z]` – litera z podanego zakresu,
`*` – zero lub więcej powtórzeń poprzedzającego elementu,
`+` – jedno lub więcej powtórzeń poprzedzającego elementu.

Można również określić położenie wzorca w wierszu:

`^wzorzec` – pasuje do linii rozpoczynających się danym wzorcem,
`wzorzec$` – pasuje do linii kończących się danym wzorcem.

Znak `^` po otwierającym nawiasie kwadratowym pełni rolę zaprzeczenia. Przykładowo `[^a-z]` oznacza znak nie będący małą literą.

Przykłady zastosowania wzorców polecenia `grep`:

`abc` – wypisuje wiersze zawierające łańcuch `abc`,
`^abc` – wypisuje wiersze zaczynające się łańcuchem `abc`,

[1] Na podstawie prac [3, 6, 9].

`abc$` – wypisuje wiersze kończące się łańcuchem `abc`,
`a..c` – wypisuje wiersze zawierające znaki `a` i `c`, rozdzielone dwoma dowolnymi znakami,
`a*c` – wypisuje wiersze zawierające znak `a` występujący dowolną liczbę razy i znak `c` lub tylko znak `c`,
`a+c` – wypisuje wiersze zawierające znak `a` występujący dowolną liczbę razy i znak `c`,
`^[Cc]` – wypisuje wiersze rozpoczynające się od znaku `C` lub `c`,
`^[^Cc]` – wypisuje wiersze nie rozpoczynające się od znaku `C` ani `c`.

Dostępne opcje:

`-i` – ignoruje wielkość liter przy porównywaniu tekstu ze wzorcem,
`-v` – wypisuje te linie, które nie zawierają podanego wzorca,
`-c` – podaje tylko liczbę wierszy odpowiadających wzorcowi,
`-n` – przed każdą linijką odpowiadającą wzorcowi podaje jej numer,
`-r` – umożliwia przeglądanie rekurencyjne katalogu.

Jeśli wzorzec zawiera odstępy lub znaki specjalne, należy ująć go w apostrofy lub cudzysłów. Jeżeli znak specjalny ma być użyty we wzorcu jako znak zwykły, wtedy należy poprzedzić go znakiem `\`. Gdy szukany wzorzec jest alternatywą, trzeba użyć operatora `|`, który maskowany jest znakiem `\`.

Ćwiczenia

Utwórz plik `tekst` o następującej treści:

```
System operacyjny LINUX
to system wielodostępny.
System ten jest również
wielozadaniowy.
```

Zadania 1–9 dotyczą przeszukiwania pliku `tekst`.

1. Wyświetl wszystkie linijki zawierające '**system**' z rozróżnieniem dużych i małych liter.

```
grep system tekst
```

2. Wyświetl wszystkie linijki zawierające '**system**', ignorując wielkość liter.

```
grep -i system tekst
```

3. Ile linijek zawiera wyraz '**system**' (ignorując wielkość liter)?

```
grep -ic system tekst
```

4. Wyświetl wszystkie linijki nie zawierające słowa '**system**', ignorując wielkość liter.

```
grep -vi system tekst
```

5. Wyświetl wszystkie linijki zaczynające się od '**Sy**'.

```
grep ^Sy tekst
```

6. Wyświetl wszystkie linijki kończące się na '**y**'.

```
grep y$ tekst
```

7. Wyświetl wszystkie linijki, w których '**oper**' poprzedza '**Sys**'.

```
grep "oper.\+Sys" tekst
```

8. Wyświetl te linie, które zawierają słowo '**to**' lub '**ten**'.

```
grep "to\|ten" tekst
```

9. Wyświetl te linie, które zawierają zarówno słowo '**to**', jak i '**ten**' (w dowolnej kolejności).

```
grep to tekst|grep ten
```

10. Odszukaj w katalogu /usr/include w plikach nagłówkowych (pliki z rozszerzeniem h) ciąg znaków '**pow**'.

```
grep pow /usr/include/*.h
```

11. Wyświetl linie zawierające tekst '**main**' ze wszystkich plików znajdujących się w poddrzewie katalogowym rozpoczynającym się w katalogu kat.

```
grep -r main kat
```

12. Przeanalizuj następujące polecenia:

```
grep ^[^d-] plik
grep "...x" plik
grep -v "^[cC]" plik.f > plikbk.f
ls -la ..|grep user1
grep "\.S" plik
grep ".S" plik
grep "int\|double" *.c
```

1.2. Wyszukiwanie plików

Do wyszukiwania plików w systemie plikowym według różnych kryteriów związanych z atrybutami pliku służą polecenia: `find`, `whereis`, `which`. Polecenie:

```
whereis plik
```

wyświetla ścieżki dostępu do plików o podanej nazwie, przeszukując standardowe katalogi linuksowe. Służy głównie do wyszukiwania programów w wersji źródłowej lub binarnej oraz dokumentacji do nich. Polecenie:

```
which plik
```

podaje ścieżkę dostępu do pliku, który jest wykonywany po wydaniu polecenia wskazanego przez parametr. Odwołuje się do zmiennej środowiskowej `$PATH`. Powyższe polecenia umożliwiają wyszukiwanie plików według nazwy. Bardziej uniwersalne jest polecenie `find`, które pozwala na wyszukiwanie plików według różnych kryteriów:

```
find katalog_startowy opcje kryterium
```

Najczęściej używane opcje i odpowiadające im kryteria poszukiwania:

`-name` – nazwa,

`-type` – typ; wymagany jest jednoznakowy argument określający typ:
- `d` – katalog,
- `f` – plik zwykły,
- `b` – plik specjalny blokowy,
- `c` – plik specjalny znakowy,
- `s` – semafor,
- `l` – link symboliczny,

`-size` – rozmiar pliku w: blokach – `b`, znakach – `c`, słowach – `w` lub kilobajtach – `k`, np.:
`-size +100c` – pliki o rozmiarze większym niż 100 znaków,
`-size -100w` – pliki o rozmiarze mniejszym niż 100 słów,

`-mtime` – czas modyfikacji – liczba dni, jakie minęły od ostatniej modyfikacji, np.:
`-mtime +3` – pliki modyfikowane więcej niż 3 dni temu,
`-mtime -3` – pliki modyfikowane mniej niż 3 dni temu,

`-atime` – czas dostępu – liczba dni, jaka minęła od ostatniego dostępu,

`-user` – użytkownik (właściciel),

`-perm` – prawa dostępu – podane symbolicznie bądź w formie 3 lub 4 cyfr z przedziału <0–7>. Polecenie `-perm 100` pozwala na odnalezienie plików, które mają ustawione prawo x dla właściciela,

`-exec` – powoduje wykonanie na odnalezionych plikach polecenia podanego po opcji exec; polecenie to musi być zakończone ciągiem znaków {} \, np.:

```
find / -name ".c" -exec cat {} \;
```

`-ok` – działa analogicznie do opcji exec, wymaga jednak dodatkowo potwierdzenia wykonania polecenia na każdym pliku,

`-newer` – czas modyfikacji pliku późniejszy niż wskazanego pliku.

Przy poszukiwaniu można korzystać również z operatorów logicznych OR, NOT, AND. Operator NOT polecenia `find` zapisuje się w postaci wykrzyknika `!`. Wykrzyknik umieszczony przed kryterium poszukiwania neguje to kryterium. Przykładowo, polecenie:

```
find . ! -name 'at'
```

pozwala odszukać wszystkie pliki mające nazwy różne od at. Operator logiczny OR zapisujemy jako **-o**, natomiast operator AND jako **-a**. Przykładowo, polecenie:

```
find . -name 'at' -o -type d
```

pozwala na wyszukanie plików o nazwie at lub typie d. Jeśli plik spełnia jedno lub drugie kryterium, jest uznawany za pasujący do wzorca. Kiedy kilka opcji zostanie podanych w wierszu poleceń, tworzą one operację AND. Polecenie:

```
find / -name 'at' >p1
```

rozpoczyna poszukiwania od głównego katalogu i szuka plików o nazwie at, a następnie zapisuje rezultat poszukiwań w pliku p1 (pełną nazwę każdego znalezionego pliku).

Ćwiczenia

1. Napisz polecenie, które odnajdzie w plikach źródłowych w C (o rozszerzeniu c) znajdujących się w katalogu /home/janek/zrodla linie zawierające tekst **tablica**, zapisze je w pliku tablica, w katalogu domowym, a na ekranie wyświetli liczbę znalezionych linii.

```
grep tablica /home/Janek/zrodla/*.c|tee tablica|wc -l
```

Należy zwrócić uwagę, że polecenie `tee` rozdzielające strumień wyjściowy umożliwia zapisanie wyniku pośredniego do pliku, a zarazem przekazanie go jako strumienia wejściowego do kolejnego polecenia w celu dalszego przetworzenia.

2. Znajdź w systemie plikowym pliki zwykłe modyfikowane ponad tydzień temu, do których są ustawione rozszerzone prawa dostępu na poziomie właściciela (`s` zamiast `x`) oraz katalogi z ustawionym bitem lepkości (`t` na poziomie reszty). Nazwy odnalezionych plików powinny pojawić się na ekranie i zostać zapisane do pliku wynik. Na ekranie nie mogą się pojawić komunikaty o błędach.

```
find  / -type f -mtime +7 -perm -4000 -o -type d \
-perm -1000 2>/dev/null
```

Znak – w opcji `perm`: `-4000` oznacza, że interesują nas pliki z ustawionym SUID, a pozostałe prawa dostępu do pliku nie są istotne (na poziomie prawa wykonania dla właściciela może wystąpić litera `s` lub `S`). Opcja `-perm -4100` pozwala znaleźć pliki, dla których ustawiony jest zarówno bit SUID (`S`), jak i prawo wykonywania dla właściciela (`x`), a więc pliki z prawem `s` na poziomie właściciela (`S + x ->s`).

3. Napisz polecenie zliczające w Twoim katalogu domowym (bez przeszukiwania w głąb) linie zawierające tekst `exec` w plikach zwykłych, których rozmiar jest mniejszy niż 200 słów.

```
find ~ -maxdepth 1 -type f -size -200w \
-exec grep exec {} \;|wc -l
```

Opcja `-exec` umożliwia wykonanie polecenia `grep` na wszystkich plikach znalezionych przez polecenie `find`.

4. Napisz polecenie zliczające pliki, których nazwy rozpoczynają się na literę `a` lub `A` z katalogu głównego oraz z Twojego katalogu domowego (suma).

```
ls -ld /[Aa]* ~/[Aa]*|wc -l
```

Najprościej zastosować polecenie `ls` z dwoma argumentami zawierającymi znaki uogólniające.

5. Ze swojego katalogu domowego wybierz pliki źródłowe w języku C, połącz je w jeden plik o nazwie `zrodla`, a do pliku `prog` zapisz te linie, które nie są komentarzem (nie rozpoczynają się znakami: `//`).

```
cat  ~/*.c |tee zrodla|grep -v ^//>prog
```

Polecenie `tee` skierowuje strumień wyjściowy polecenia `cat` do pliku `zrodla`, a jednocześnie umożliwia dalsze jego przetwarzanie za pomocą polecenia `grep`.

6. W plikach z Twojego katalogu domowego, których nazwy rozpoczynają się na literę a i zawierają dowolną cyfrę (nazwy!!!), znajdź linie rozpoczynające się dowolną dużą literą, a kończące się kropką. Znalezione linie zapisz do pliku linie.

```
grep ^[A-Z] ~/a*[0-9]*|grep \.$>linie
```

Wśród linii spełniających pierwszy warunek zadania – rozpoczynających się dowolną dużą literą – kolejne polecenie `grep` odnajduje linie kończące się kropką. Zwykle istnieje wiele sposobów zrealizowania zadanego przeszukiwania. Powyższe przykłady pokazują możliwe rozwiązania. Należy zwrócić uwagę na to, aby wybierać metody najprostsze, najmniej obciążające system, np. unikać stosowania polecenia `find`, gdy przeszukiwany jest jedynie wybrany katalog (zamiast poddrzewa katalogowego), a pliki wyszukiwane są według kryterium nazwy.

2. Lista kontroli dostępu – ACL[2]

Polecenie chmod systemu Linux pozwala definiować prawa dostępu do plików na trzech poziomach: dla właściciela pliku, grupy oraz pozostałych użytkowników systemu. W wielu wypadkach takie możliwości nie są wystarczające. Nie można za jego pomocą np. nadawać praw dostępu do plików dla wybranych użytkowników lub grup. Rozwiązaniem tego problemu jest stosowanie listy kontroli dostępu (ACL – *Access Control List*), która jest dostępna w większości dystrybucji Linuksa. Plikom i katalogom tworzonym w systemie Linux nadawane są prawa dostępu zależne od maski, którą ustawia się poleceniem umask. Jest to polecenie wewnętrzne (wbudowane) powłoki. Polecenie umask określa, które prawa mają być zamaskowane. Na przykład polecenie:

```
$umask 077
```

sprawi, że nowo utworzone pliki i katalogi nie będą miały żadnych praw na poziomie grupy oraz reszty użytkowników systemu:

```
$mkdir katalog
$touch plik
```

Poniższe polecenie pokazuje prawa do pliku plik i katalogu katalog – zostały one nadane zgodnie z maską 077:

```
$ls -ld katalog plik
drwx------ 2 anka anka 4096 2009-03-24 17:44 katalog
-rw------- 1 anka anka    0 2009-03-24 17:44 plik
```

Takim podstawowym uprawnieniom odpowiadają stosowne wpisy na liście ACL. Można je wyświetlić za pomocą polecenia getfacl. Polecenie getfacl wyświetla informacje o pliku: nazwę, właściciela, grupę oraz ACL, czyli listę wszystkich ustawionych praw dostępu do pliku. Każda linia opisuje prawa dostępu do jednej z trzech podstawowych klas użytkowników:

```
$getfacl katalog plik
# file: katalog
#owner: anka
#group: anka
user::rwx
group::---
```

[2] Na podstawie pracy [6].

```
other::---
#file: plik
#owner: anka
#group: anka
user::rw-
group::---
other::---
```

Do modyfikacji listy ACL służy polecenie `setfacl`. Umożliwia ono nadawanie praw dostępu do pliku dla dowolnego użytkownika oraz dowolnej grupy. Na liście ACL występuje pięć typów wpisów. Określają one:

- ACL_USER_OBJ – prawa użytkownika (odpowiada podstawowym prawom na poziomie właściciela pliku – `user`).
- ACL_USER – prawa dowolnego użytkownika.
- ACL_GROUP_OBJ – prawa grupy (odpowiada podstawowym prawom dla grupy – `group`).
- ACL_GROUP – prawa dowolnej grupy.
- ACL_MASK – maskę (określa maksymalne prawa dostępu do pliku dla wszystkich użytkowników z wyjątkiem właściciela (`user`) oraz innych (`other`)).
- ACL_OTHER – prawa innych (odpowiada podstawowym prawom dla reszty użytkowników systemu – `other`).

Aby nadać użytkownikowi `jas` prawo pisania do pliku plik, należy użyć opcji `m` (*modify*) polecenia `setfacl`:

```
$setfacl -m u:jas:w plik
```

Polecenie `ls -l` sygnalizuje istnienie listy ACL symbolem +, który pojawia się za prawami dostępu do pliku:

```
-rw--w----+ 1 anka anka     0 2009-03-24 17:44 plik
```

Pojawiło się prawo w dla grupy, które bez stosowania ACL oznaczałoby możliwość zapisu do pliku plik dla członków grupy pliku. Jednak w wypadku ACL efektywne prawa dostępu dla członków grupy tworzone są na podstawie wpisu dotyczącego praw grupy (tutaj: `---`).
Polecenie `getfacl --omit-header` wyświetla pełne informacje o prawach dostępu do pliku z pominięciem trzech linii nagłówkowych:

```
user::rw-
user:jas:-w-
group::---
mask::-w-
other::---
```

Poza wpisem określającym prawa dla użytkownika jas pojawił się wpis określający maskę. Dotyczy ona wszystkich użytkowników (poza właścicielem) i grup z listy ACL i oznacza, że prawo, które nie występuje w masce, nie będzie nadane. W naszym wypadku żaden użytkownik ani grupa nie uzyskają praw r i x. Maska -w- blokuje prawo odczytu pliku oraz wykonania dla wszystkich użytkowników z pominięciem właściciela pliku.

Do usuwania wpisów z listy ACL służy opcja x polecenia setfacl. Polecenie:

```
#setfacl -x u:jas plik
```

usuwa prawa dostępu do pliku plik dla użytkownika jas. Z listy ACL pliku plik usuwane są wpisy dotyczące użytkownika jas.
Jeśli po nadaniu uprawnień do pliku dla określonego użytkownika zmienimy maskę pliku, prawa dostępu do pliku mogą zostać ograniczone. Zbyt restrykcyjna maska blokuje możliwość dostępu do pliku dla wszystkich użytkowników. W poniższym przykładzie użytkownik jas uzyskuje prawa rwx do katalogu kat, a następnie w wyniku polecenia chmod maska pliku zostaje zmodyfikowana i w efekcie użytkownik jas traci prawa rw do pliku plik. Polecenie getfacl sygnalizuje tę sytuację, wyświetlając efektywne prawa użytkownika.

```
#umask 077
#mkdir kat
#ls -ld kat
drwx------ 1 anka anka    0 2009-03-26 17:44 kat
#setfacl -m u:jas:rwx kat
#ls -ld kat
drwxrwx---+ 1 anka anka    0 2009-03-26 17:44 kat
#getfacl -omit-header kat
user::rwx
user:jas:rwx
group::---
mask::rwx
other::---
#chmod g=x kat
#ls -ld kat
drwx--x---+ 1 anka anka    0 2009-03-26 17:44 kat
#getfacl --omit-header kat
user::rwx
user:jas:rwx        #effective:--x
group::---
mask::--x
other::---
```

Po nadaniu pełnych praw dla grupy użytkownik jas odzyskuje prawa rwx do katalogu kat:

```
#chmod g=rwx kat
#getfacl --omit-header kat
user::rwx
user:jas:rwx
group::---
mask::rwx
other::---
```

Przy każdej próbie dostępu do pliku wykonywany jest algorytm, który sprawdza poprawność odwołania, przeglądając wpisy w ACL. W zamieszczonym poniżej algorytmie 2.1 zmienna ACCES przyjmuje wartość TRUE, gdy dostęp jest dozwolony, a FALSE, gdy dostępu nie ma.

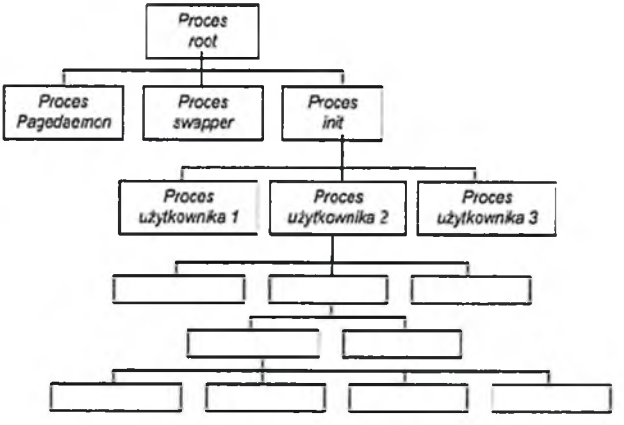
Przyjęte oznaczenia:
EUID – efektywny identyfikator użytkownika,
EGID – efektywny identyfikator grupy,
USER – właściciel pliku,
GROUP – grupa pliku.

```
if ( EUID==USER )
        if ( wpis ACL_USER_OBJ zawiera odpowiednie prawa )
                        ACCESS=TRUE
        else
                        ACCESS=FALSE

else if ( EUID odpowiada wpisowi ACL_USER )
        if( wpis ACL_USERzawiera odpowiednie prawa && wpis
            ACL_MASK zawiera odpowiednie prawa )
            ACCESS=TRUE
       else
            ACCESS=FALSE

   else if ( EGID lub ID jednej z grup procesu odpowiada GROUP lub
            wpisowi ACL_GROUP )
  if ( istnieje wpis  ACL_MASK )
                        if ( ACL_MASK zawiera odpowiednie prawa &&
                        któryś z powyższych      wpisów dla grupy zawiera
                        odpowiednie prawa )
                          ACCESS=TRUE
                else
                          ACCESS=FALSE
  else
                        if( któryś z powyższych  wpisów dla grupy zawiera
                        odpowiednie prawa )
                          ACCESS=TRUE
                else
                          ACCESS=FALSE
  else if ( wpis ACL_OTHER zawiera odpowiednie prawa )
  ACCESS=TRUE
     else
              ACCESS=FALSE
```

Alg. 2.1. Sprawdzanie poprawności odwołania do pliku z ACL

3. Procesy[3]

Proces w ramach Linuksa jest egzemplarzem wykonującego się programu, co odpowiada pojęciu zadania (*task*) w innych środowiskach. Zadania czy polecenia mogą się składać z wielu procesów wykonujących określoną czynność.
Proces jest więc elementarną, dynamiczną jednostką, która żyje w środowisku komputerowym i zmienia stany, aby doprowadzić do rozwiązania postawionego problemu. Za każdym razem, gdy powłoka uruchamia program, w odpowiedzi na polecenie tworzy ona nowy proces. Proste komendy wykonują się jako jeden proces, natomiast złożone polecenia, korzystające z potoków, powodują uruchomienie jednego procesu dla każdego segmentu potoku. Wiele procesów może jednocześnie wykonywać ten sam program – kilku użytkowników może równocześnie używać tego samego programu edytora, a każde wywołanie edytora jest liczone jako oddzielny proces. Procesy w systemie mogą być wykonywane współbieżnie, mogą też być dynamicznie tworzone i usuwane. System operacyjny musi więc zawierać mechanizmy tworzenia i kończenia procesu. Proces może tworzyć nowe procesy za pomocą wywołania systemowego. Proces tworzący nazywa się procesem macierzystym, a utworzone przez niego procesy jego potomkami. Każdy nowy proces może tworzyć kolejne procesy – powstaje wtedy drzewo procesów. Na wierzchołku drzewa znajduje się jeden proces sterujący wykonaniem niezwykle ważnego procesu init, który jest przodkiem wszystkich procesów systemowych oraz procesów użytkownika. Drzewo procesów w systemie Linux przedstawiono na rysunku 3.1.



Rys. 3.1. Drzewo procesów w systemie Unix

[3] Na podstawie prac [7–9].

3.1. Rozwidlanie procesów – identyfikatory procesu

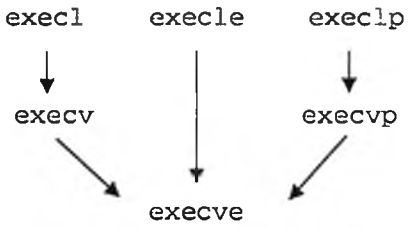
Linux dostarcza programistom kilku funkcji systemowych do tworzenia i manipulacji procesami. Najważniejsze z nich zamieszczono w tabeli 3.1.

Tabela 3.1

Polecenia systemowe związane z procesami

`fork()`	używane do tworzenia nowych procesów przez powielenie procesu wywołującego
`exec()`	rodzina procedur bibliotecznych i jedna funkcja systemowa odpowiadają za przekształcenie procesu wskutek podmiany jego przestrzeni adresowej przez nowy program
`wait()`	zapewnia elementarną synchronizację procesów – pozwala jednemu procesowi czekać, aż zakończy się inny proces z nim powiązany
`exit()`	używane do zakończenia procesów

Nowy proces tworzy się za pomocą funkcji systemowej zwanej rozwidleniem (`fork`), wykonanej przez proces macierzysty. Zawiera ono kopię przestrzeni adresowej procesu pierwotnego (ten sam program i te same zmienne, z tymi samymi wartościami). Proces „dziecko" jest więc dokładną kopią procesu macierzystego, lecz ma własny identyfikator PID. Oba procesy (macierzysty i potomny) kontynuują działanie od instrukcji występującej po `fork`, ale z jedną różnicą – funkcja `fork` przekazuje zero do nowego procesu (potomka), natomiast do procesu macierzystego przekazuje identyfikator procesu potomnego. Mechanizm rozwidlania procesów `fork` powoduje utworzenie tylko kopii procesu, co daje ograniczone możliwości tworzenia różnorodnych procesów. Aby było możliwe uruchamianie innych programów, konieczny jest jeszcze mechanizm podmiany kodu procesu. Jeśli uruchamiamy np. program `ls`, w pierwszej chwili powstaje kopia procesu naszej powłoki, której kod jest następnie podmieniany na kod programu `ls`. Każdy proces może w dowolnym momencie podmienić swój kod na inny, czyli zupełnie zmienić swoje własności i funkcje, nie zmieniając identyfikatorów procesu. W języku C dostępna jest rodzina funkcji służących do podmiany kodu. Drzewo wywołania `exec` pokazano na rysunku 3.2.



Rys. 3.2. Drzewo rodziny funkcji exec

Funkcje różnią się między sobą sposobem przekazywania parametrów, ostatecznie jednak wszystkie wywołują `execve()`, która jest rzeczywistą funkcją systemową. Rodzina funkcji `execel()` musi mieć podane argumenty jako listę zakończoną NULL, natomiast rodzina funkcji `execv()` wymaga tablicy argumentów.

```
int execl(const char *path, const char *arg0, ...,
const char *argn, char * /*NULL*/);
```

```
int execlp(const char *file, const char *arg0, ...,
const char *argn, char * /*NULL*/);
```

```
int execv(const char *path, char *const argv[]);
```

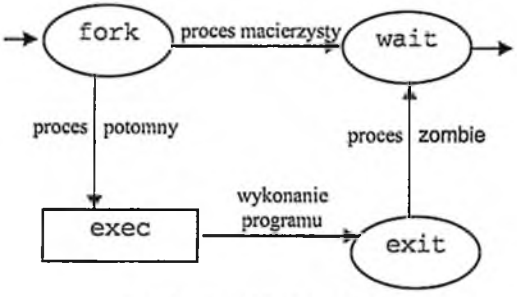
```
int execle(const char *path, const char *arg0, ...,
const char *argn, char * /*NULL*/, char *const
envp[]);
```

```
int execve(const char *path, char *const argv[], char
*const envp[]);
```

```
int execvp(const char *file, char *const argv[]);
```

Rodzina funkcji systemowej `exec()` umożliwia procesowi wykonanie kodu znajdującego się we wskazanym pliku. Po rozwidleniu, w wyniku wykonania funkcji systemowej `exec`, treść programu zawarta w przestrzeni adresowej procesu potomnego ulega wymianie na treść programu, który ma być wykonany w ramach

tego procesu. Treść nowego programu zastępuje tę odziedziczoną od rodzica, a bez zmian pozostają środowisko procesu (ustawienia zmiennych, strumień wejściowy, wejściowy i błędów) oraz priorytet procesu. W opisany sposób tworzone są wszystkie procesy systemowe. Rozwidlanie procesów przedstawiono na rysunku 3.3.



Rys. 3.3. Rozwidlanie procesów

Do wywołania powyższych funkcji niezbędne są następujące pliki nagłówkowe:

```
<sys/types.h>
<unistd.h>
```

Przykład typowego wywołania funkcji `fork()` i `exec()` podają kod 3.1 oraz kod 3.2.

```
switch (fork())
{case -1:
     perror(„fork error");
     exit(1);
 case 0:
      /* akcja dla procesu potomnego */
 default:
     /* akcja dla procesu macierzystego */
}
```

Kod 3.1. Typowe wywołanie funkcji `fork`

```
switch (fork())
{case -1:
      perror(„fork error");
      exit(1);
case 0: /* proces potomny */
     execl("./nowy_program.x","nowy_program.x",NULL);
     exit(2);
default:  /* proces macierzysty */}
```

Kod 3.2. Typowe wywołanie funkcji fork z exec

W momencie utworzenia proces otrzymuje swój unikalny identyfikator PID przydzielony przez system, który służy do identyfikacji procesów. Nie da się przewidzieć wartości PID (jedynie proces init ma zawsze taki sam numer PID równy 1). Każdy proces dziedziczy też od swojego rodzica jego identyfikator jako PPID (*Parent Process Identifier*). W ten sposób można jednoznacznie określić rodzica danego procesu. Z każdym procesem związany jest jeszcze identyfikator grupy procesów GPID (*Group Process Identifier*). Procesy należące do jednej grupy mają taki sam GPID. Do uzyskania wartości PID, PPID oraz GPID procesu służą następujące funkcje języka C, które są zadeklarowane w pliku nagłówkowym <unistd.h>:

```
pid_t getpid();
pid_t getppid();
pid_t getpgrp();
```

Program zamieszczony poniżej obrazuje działanie funkcji fork() oraz exec() z wypisaniem identyfikatorów procesu.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
void wypisz(void);
int main()
{
   int i;
   pid_t child;
   printf("Proces macierzysty:\n");
   wypisz();
   for (i = 1; i < 4; i++)
```

```
    {
       child = fork();
       switch (child)
       {
          case -1:
             perror("fork error !!!");
             exit(1);
          case 0:
             printf("\nProces potomny %d:\n", i);
             wypisz();
             execl("./zad1","zad1",NULL);
       }
    }
    sleep(3);
    exit(0);
}
void wypisz(void)
{
    printf("id uzytkownika: %d\n", (int)getuid());
    printf("id grupy uzytkownika: %d\n",
(int)getgid());
    printf("id procesu: %d\n", (int)getpid());
    printf("id pr macierzystego: %d\n",
(int)getppid());
    printf("id grupy procesow: %d\n", (int)getpgrp());
    fflush(NULL);
}
```

Kod 3.3. Utworzenie procesów potomnych i wypisanie ich identyfikatorów

3.2. Oczekiwanie na zakończenie procesów potomnych

Procesy powinny kończyć swoje działanie w kolejności odwrotnej do tej, w jakiej powstawały. Proces macierzysty nie powinien zakończyć się wcześniej niż jego procesy potomne. Jednakże taki wariant też jest możliwy. Jeśliby się tak stało, PPID w procesie potomnym utraci ważność. Istnieje niebezpieczeństwo, że zwolniony identyfikator procesu zostanie przydzielony przez system innemu procesowi. Dlatego też „osierocony" proces zostaje przejęty przez proces init, a jego PPID przyjmuje wartość 1. Sytuacja taka nie jest jednak czymś nienormalnym, a w niektórych systemach osierocone procesy nie mogą się poprawnie zakończyć i pozostają w systemie jako tzw. procesy zombie. Należy więc zadbać o to, aby

proces macierzysty poczekał na zakończenie swoich procesów potomnych i odebrał od nich kod zakończenia procesu. W tym celu proces macierzysty powinien wywołać funkcję wait tyle razy, ile utworzył procesów potomnych. Funkcja ta chwilowo zawiesza wykonanie procesu, podczas gdy działa proces potomny. Gdy potomek się zakończy, czekający proces rodzica zostanie wznowiony. Jeśli działa więcej niż jeden proces potomny i którykolwiek z nich się zakończy, wait powraca. Często jest wywoływana przez proces rodzica zaraz po wywołaniu fork i ma następującą składnię:

```
#include <sys/types.h>
#include <sys/wait.h>
int wait(int *stat_loc);
```

Funkcja zwraca identyfikator zakończonego procesu potomnego lub –1, jeżeli proces nie ma potomków. W zmiennej wskazywanej przez stat_loc zapisywana jest liczba szesnastkowa w postaci: XXYY, gdzie XX to kod zakończenia procesu potomnego, zaś YY jest numerem sygnału, który spowodował zakończenie procesu potomnego. Jedno wywołanie funkcji wait oczekuje tylko na zakończenie jednego procesu potomnego, dlatego proces rodzica powinien w pętli czekać na zakończenie każdego potomka. Funkcja systemowa wait jest zwykle używana do synchronizacji wykonania procesu macierzystego i procesu potomnego, zwłaszcza wtedy, gdy oba procesy mają dostęp do tych samych plików. Proces może zakończyć swoje działanie za pomocą funkcji systemowej exit, a jego proces macierzysty może oczekiwać na to zdarzenie przez wywołanie funkcji wait.

Jeżeli proces potomny zakończy się w czasie, gdy jego rodzic wywołuje funkcję wait, wtedy znika i jego status wyjścia jest przekazywany do rodzica za pomocą wywołania wait. Przykład podany poniżej jako kod 3.4 ilustruje połączenie fork() z wait().

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
int main(int argc, char **argv)
{
    int i;
    int status;  /* Stan wyjscia z wait() */
    pid_t wpid;  /* Identyfikator procesu z wait() */
    pid_t pid;  /* Identyfikator procesu potomnego */
    system("clear");
```

```
for(i=0; i<3; i++)
{
    puts("\n");
    pid = fork();

    switch(pid)
    {
    case -1:
 fprintf(stderr,"%s:Blad\n",strerror(errno));
 exit(1);
    case 0:
 printf("PID %ld: Proces potomny uruchomiony PPID:
%ld \n",   (long)getpid(), (long)getppid());
 execl("/bin/ps","ps",NULL);
 break;
    default:
 printf("PID %ld: Proces macierzysty, PID %ld. \n",\
     (long)getpid(),/* nasz PID */ \
     (long)pid);  /* PID proc. macierzystego */
 wpid = wait(&status); /* Stan wyjscia z procesu
potomnego */
 if(wpid == -1) {
     fprintf(stderr,"%s: wait()\n",strerror(errno));
     return(1);
     }
 else if (wpid != pid)
     abort();  /*To sie nigdy nie powinno zdarzyc w
tym programie */
 else {
     printf("Proces potomny o ident. %ld zakonczyl
dzialanie ze stanem 0x%04X\n",
   (long)pid,  /* PID proc. potomnego */
   status); /* Stan wyjscia */
      }
    }
}
return 0;
}
```

Kod 3.4. Połączenie `fork()` z `wait()`

3.3. Przedwczesne zakończenia – procesy zombie

Jeśli proces potomny zakończy się w czasie, gdy jego rodzic wywołuje funkcję wait, wtedy znika, a jego status wyjścia przekazywany jest do procesu rodzica. Mogą jednak wystąpić dwie inne sytuacje:

- Proces potomny kończy się w czasie, gdy jego proces rodzicielski nie wykonuje funkcji wait – kończący się proces jest umieszczany w stanie zawieszenia i staje się zombie.
- Proces rodzica kończy się, gdy jeden lub więcej procesów potomnych ciągle działa – procesy potomne są adaptowane przez proces init.

Proces zombie to taki proces, który zakończył działanie, ale nie został jeszcze wyczyszczony. Odpowiedzialność za wyczyszczenie swojego potomka spoczywa na procesie rodzica. Poniżej podany kod tworzy proces zombie.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t child_pid;
   printf("Proces macierzysty:\n");
   /*Tworzymy proces potomny  */
      child_pid = fork();
      if(child_pid >0) {
      {/* Proces rodzica - zasypiamy na minute  */
      sleep(60);
      }
      else
      {/* Proces potomny - natychmiast kończy
działanie  */
      exit(0)
      }
}
```

Kod 3.5. Utworzenie procesu zombie

Proces init o identyfikatorze równym 1 automatycznie czyści wszystkie procesy zombie, które są przez niego przejmowane. Aby upewnić się, że nie pozostawimy pożerających zasoby systemowe procesów zombie, należy okresowo wywoływać przez rodzica funkcję waitpid, która wyczyści potomków zombie.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Użycie opcji WNOHANG powoduje, że funkcja będzie działała w trybie nieblokującym, wyczyści proces potomny, jeśli taki znajdzie (zwraca ID zakończonego potomka), w przeciwnym wypadku zakończy działanie (zwraca 0).

```
#include <sys/wait.h>
#include <stdlib.h>
#include <sys/types.h>

int main (void) {
    pid_t pid;
    int status, exit_status;

    if ((pid = fork()) < 0)
       printf("Blad! Nie mozna utworzyc nowego
procesu\n");
    if (pid == 0) { //potomek
       printf("Potomek o pid %d jest uspiony...\n",
getpid());
       sleep(7); /* Uspienie na 7s */
       exit(5); /* Wyjscie  potomka z wartoscia
niezerowa */
    }
    /* Odtad dziala rodzic */
    while (waitpid(pid, &status, WNOHANG) == 0) {
       printf("Nadal czekam na zakonczenie
potomka...\n");
       sleep(1); /* Uspienie na 1s */
    }

    /* Uruchamiane gdy potomek sie zakonczy */
    exit_status = WEXITSTATUS(status);
    printf("Proces %d zakonczyl sie zwracajac wartosc
%d\n", pid, exit_status);
    exit (0);
}
```

Kod 3.6. Przykład wywołania funkcji `waitpid`

Bardziej eleganckim rozwiązaniem jest powiadamianie procesu rodzica przez wysłanie sygnału SIGCHLD i jego obsłużenie. Program zamieszczony poniżej jako kod 3.7 ilustruje, jak można wykorzystać procedurę obsługi SIGCHLD do wyczyszczenia procesów potomnych.

```
#include <sys/wait.h>
#include <signal.h>
#include <sys/types.h>
#include <string.h>
sig_atomic_t child_exit_status;

void clean_up_child_proces(int signal_number){
    /* czyszczenie procesu potomnego  */
    int status;
    wait(&status);
    /* zachowanie statusu wyjścia      */
    child_exit_status = status;}
int main ()
{
    /* obsługa SIGCHLD poprzez wywołanie
   clean_up_child_proces   */
   struct sigaction sigchld_action;
   memset (&sigchld_action,0,sizeof(sigchld_process);
   sigchld_action.sa_handler =
&clean_up_child_process;
   sigaction (SIGCHLD, &sigchld_action, NULL);
   /* Tutaj wykonywane są właściwe działania -
      tworzenie procesu potomnego */
     ..... }
```

Kod 3.7. Przykład wykorzystania procedury obsługi SIGCHLD

3.4. System plików /proc

W katalogu głównym systemu plikowego znajduje się katalog proc. Jest to punkt montowania wirtualnego systemu plików. Wykonując polecenie mount bez parametrów, otrzymujemy:

```
none on /proc type proc (rw)
```

Pierwsze pole none oznacza, że nie jest on związany z żadnym urządzeniem sprzętowym, takim jak dysk. System plików /proc stanowi okno, przez które można popatrzeć na działające jądro Linuksa. Pliki nie mają swoich odpowiedników na urządzeniach fizycznych, są tworzone przez jądro Linuksa w czasie ich czytania. Dzięki nim mamy dostęp do parametrów, struktur danych i statystyk jądra. System plików /proc dla każdego powołanego do życia procesu tworzy katalog, którego nazwą jest identyfikator procesu PID. Katalogi te pojawiają się oraz znikają dynamicznie, zgodnie z rozpoczęciem lub zakończeniem procesu. W każdym takim katalogu przechowywane są informacje o działającym procesie. Są to następujące pozycje:

cmdline – lista argumentów procesu,

cmd – dowiązanie symboliczne wskazujące aktualny katalog roboczy,

environ – środowisko procesu; zmienne środowiskowe oddzielone są znakami NULL,

exe – dowiązanie symboliczne wskazujące obraz wykonującego się procesu,

fd – podkatalog zawierający pozycje z otwartymi przez proces deskryptorami plików,

maps – wyświetlenie informacje o plikach odwzorowanych w procesie: zakres adresów w przestrzeni adresowej procesu, uprawnienia, nazwę pliku, wszystkie załadowane biblioteki współdzielone i inne informacje,

root – dowiązanie do głównego katalogu procesu,

status – zawiera różne informacje o procesie, m.in.: ID danego procesu i jego rodzica, prawdziwe i efektywne identyfikatory użytkownika i grupy,

statm – zawiera statystyki pamięci procesu: całkowity rozmiar procesu, rozmiar procesu w pamięci fizycznej, pamięć współdzieloną z innymi procesami, rozmiar załadowanego kodu wykonywalnego, rozmiar odwzorowanych przez proces bibliotek współdzielonych, pamięć używaną przez proces dla stosu oraz liczbę zmodyfikowanych przez program stron pamięci.

Większość pozycji katalogu /proc dostarcza czytelnej informacji, którą łatwo można przetworzyć. Są to między innymi pliki:

/proc/cpuinfo	– zawiera informacje o procesorze lub procesorach, na których działa system,
/proc/dma	– lista zarejestrowanych kanałów DMA,
/proc/devices	– podaje numery dostępnych w systemie urządzeń znakowych i blokowych,
/proc/pci	– podaje urządzenia podłączone do szyn PCI,
/proc/version	– zawiera numer wydania i kompilacji jądra, kiedy to się stało, na jakiej maszynie, jakiego użyto kompilatora, kto je skompilował,
/proc/stat	– niektóre statystyki jądra systemu,
/proc/kmsg	– komunikaty jądra (jeśli działa syslog, to nie powinno się ich czytać),
/proc/memoinfo	– zawiera informacje o wykorzystaniu pamięci systemowej (fizycznej oraz przestrzeni wymiany),
/proc/filesystem	– spis systemów plików obsługiwanych przez jądro,
/proc/mounts	– podaje podsumowanie zamontowanych systemów plików,
/proc/locks	– pokazuje wszystkie aktualnie wykorzystane w systemie blokady plików,
/proc/loadavg	– zawiera informacje o obciążeniu systemu, m.in.: liczbę aktywnych zadań w systemie i aktualnie działających procesów, ID ostatnio wykonywanego procesu,
/proc/uptime	– podaje czas od uruchomienia systemu oraz czas bezczynności systemu.

Katalog /proc/net zawiera informacje sieciowe, m.in.:

/proc/net/dev	– zawiera informacje o urządzeniach sieciowych,
/proc/net/tcp /proc/net/udo /proc/net/unix	– tabele gniazd.

Katalog /proc/sys zawiera pseudopliki reprezentujące zmienne jądra. Znajdują się w nim głównie podkatalogi z podziałem tych zmiennych na grupy. Najczęściej są używane:

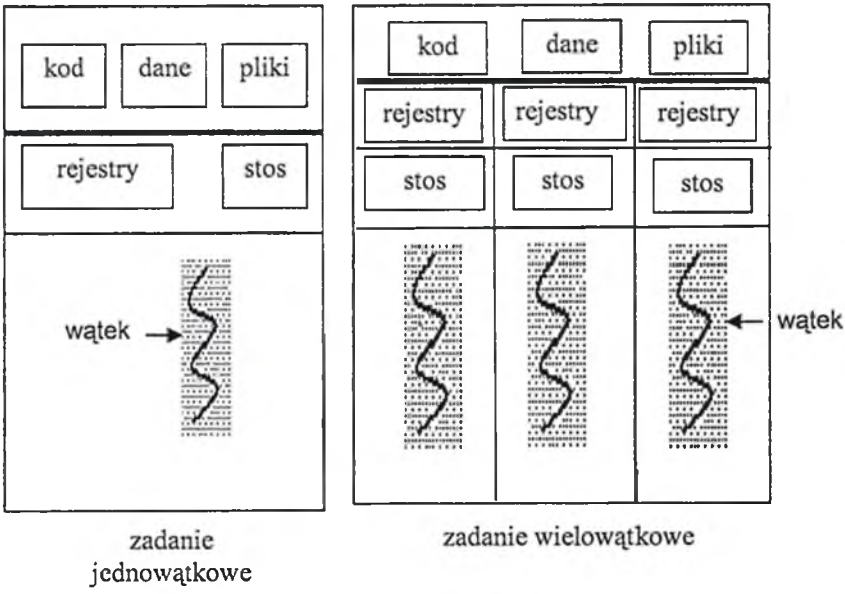
kernel/threads-max – podaje maksymalną liczbę procesów,

kernel/hostname – nazwa komputera.

Większość z tych plików ma postać tekstową i czasami mogą być w postaci niezbyt czytelnej. Istnieje wiele poleceń, które odczytują takie pliki i specjalnie je przetwarzają. Przykładem może być program `free`, który interpretuje plik /proc/meminfo.

4. Wątki[4]

Wątek, nazywany niekiedy procesem lekkim, jest podstawową jednostką wykorzystania procesora, w skład której wchodzą: licznik rozkazów, zbiór rejestrów i obszar stosu. Wątek współużytkuje wraz z innymi równorzędnymi wątkami sekcję kodu, sekcję danych oraz takie zasoby systemu operacyjnego, jak otwarte pliki i sygnały. Wątki istnieją wewnątrz procesów. Proces tradycyjny, czyli ciężki, jest równoważny zadaniu z jednym wątkiem. Zadania jedno- i wielowątkowe pokazano na rysunku 4.1.



Rys. 4.1. Zadania jedno- i wielowątkowe

Daleko posunięte dzielenie zasobów powoduje, że przełączanie procesora między równorzędnymi wątkami, jak również tworzenie wątków są tanie w porównaniu z przełączaniem kontekstu między procesami ciężkimi. Przełączanie kontekstu między wątkami wymaga przełączania zbioru rejestrów i nie trzeba wykonywać żadnych prac związanych z zarządzaniem pamięcią.

[4] Na podstawie prac [4, 8].

Linux implementuje standardowe API wątków POSIX znane jako pthreads. Wszystkie funkcje obsługi wątków i typy danych są zadeklarowane w pliku nagłówkowym pthread.h.

4.1. Tworzenie wątków

Każdy wątek w procesie jest identyfikowany przez ID wątku. Po utworzeniu wątek wykonuje funkcję wątku, czyli zwykłą funkcję zawierającą kod, który wątek ma wykonać. Po zakończeniu funkcji kończy się również wątek, który ją wykonywał. Funkcje wątków pobierają jeden parametr typu `void*` i zwracają wartość typu `void*`. Linux przekazuje tę wartość do wątku, nie wykonując na niej żadnej operacji. Program może korzystać z tego parametru do przekazywania danych do nowego wątku. Wartość zwracaną możemy użyć do przekazania danych z kończącego się wątku do programu tworzącego wątek. Do tworzenia wątku służy funkcja `pthread_create()` o następującej postaci:

```
int pthread_create(pthread_t *tid, const pthread_attr_t
*attr, void *(*func)(void *), void *arg)
```

Należy podać:

- Wskaźnik zmiennej typu `pthread_t` – przechowuje ID utworzonego wątku.
- Wskaźnik obiektu atrybutu wątku – steruje szczegółami interakcji wątku z resztą programu. Jeśli będzie to wartośc typu NULL, wątek zostanie utworzony z domyślnymi wartościami..
- Wskaźnik funkcji wątku – zwykły wskaźnik funkcji typu `void*`.
- Wartość argumentu wątku typu `void*` – cokolwiek podamy, będzie przekazane jako argument do funkcji wątku przy rozpoczęciu jego wykonania.

Aby zrozumieć działanie wątków, porównajmy sterowanie wieloma wątkami ze sterowaniem wieloma procesami. Każdy proces działa niezależnie od innych procesów, tzn. ma własny licznik rozkazów, rejestr stosu i przestrzeń adresową. Taka organizacja jest wygodna wówczas, gdy zadania wykonywane przez procesy nie są ze sobą powiązane. Wiele procesów może też realizować to samo zadanie. Do obsługi tego samego przedsięwzięcia wydajniej jest zastosować jeden proces z wieloma wątkami. Jeden proces wielowątkowy zużywa mniej zasobów niż zwielokrotnione procesy. Wykonanie wątku w procesie przebiega sekwencyjnie, a każdy wątek ma własny stos i licznik rozkazów. Poniżej przedstawiono przykładowy program, w którym wykorzystane zostały wątki do obliczania sumy elementów kolejnych wierszy macierzy. Policzone sumy z wątków przekazywane są do wątku głównego, gdzie następuje ich zsumowanie.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define REENTRANT
int sum1, sum2;
int tab[2][10];
void *rob1()
{
  int i;
  for (i=0; i<10; i++) sum1 += tab[0][i];
    printf("Suma pierwszego wiersza  %d\n", sum1 );
    pthread_exit(0);
}
void *rob2()
{
  int i;
  for (i=0; i<10; i++) sum2 += tab[1][i];
    printf("Suma drugiego wiersza %d\n", sum2 );
    pthread_exit(0);
}
int main()
{
int i,j;
pthread_t id1, id2;
srand(0);
pthread_create(&id1, NULL, rob1, NULL);
pthread_create(&id2, NULL, rob2, NULL);
for(i=0; i<2; i++)
  for(j=0; j<10; j++)
    tab[i][j] = rand() %20;
pthread_join( id1, NULL );
pthread_join( id2, NULL);
printf("suma wszystkich elementow %d\n", sum1 + sum2 );
pthread_detach( id1 );
pthread_detach( id2 );
return 0;}
```

Kod 4.1. Wykorzystanie wielowątkowości

W bibliotece standardowej C nie ma funkcji obsługi wątków, znajdują się one w bibliotece `libpthread`, dlatego przy kompilacji programu należy dodać opcję `-lpthread`.

```
gcc -o watek watek.c -lpthread
```

Tak jak `pthread_create()` rozdziela dany program na dwa wątki, funkcja `pthread_join()` scala dwa wątki w jeden.

```
int pthread_join(pthread_t  tid, void **status)
```

Pobiera ona dwa argumenty:

- ID wątku, na który chcemy poczekać.
- Wskaźnik zmiennej `void*`, w której zachowana zostanie wartość zwrócona przez wątek – jeśli nie jest nam potrzebna, wpisujemy NULL.

Czasem przydaje się możliwość sprawdzenia, który wątek jest wykonywany. Funkcja `pthread_self()` zwraca ID wątku, w którym zostanie wywołana. ID wątku można porównywać z innym za pomocą funkcji `pthread_equal()`. Funkcję tę można wykorzystać do sprawdzenia, czy dany wątek odpowiada aktualnemu wątkowi, ponieważ błędem jest wywołanie przez wątek funkcji `pthread_join()` w odniesieniu do tego samego wątku.

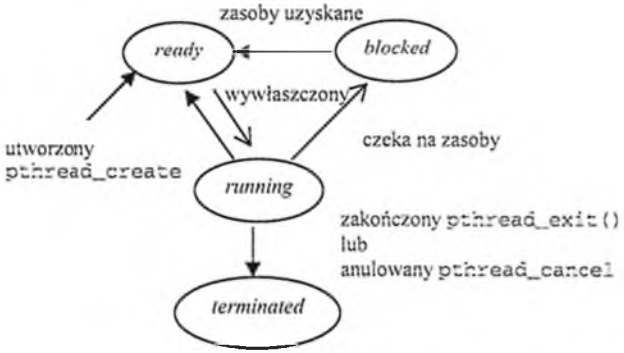
```
if (!pthread_equal (pthread_self (), other_thread))
pthread_join (other_thread, NULL)
```

4.2. Atrybuty wątków

Atrybuty wątków umożliwiają sterowanie zachowaniem wątku. Tworząc nowy wątek, do jego skonfigurowania nie musimy korzystać z wartości domyślnych, ale możemy utworzyć i ustawić własny obiekt atrybutów wątku. W tym celu należy:

- Utworzyć obiekt `pthread_attr_t` poprzez deklarację zmiennej automatycznej tego typu.
- Wywołać funkcję `pthread_attr_init`, przekazując jej wskaźnik do utworzonego obiektu. Atrybuty zostaną zainicjowane wartościami domyślnymi.
- Zmodyfikować obiekt atrybutów nowymi wartościami atrybutów.
- Przekazać przy tworzeniu wątku wskaźnik do obiektu atrybutu.
- Wywołać funkcję `pthread_attr_destroy`, aby zwolnić obiekt atrybutów.

Pojedynczego obiektu atrybutów można używać do zainicjowania wielu wątków. W przeciwieństwie do procesów wszystkie wątki w jednym programie współdzielą przestrzeń adresową, co umożliwia wątkom korzystanie z tych samych danych. Każdy wątek ma jednak swój własny stos wywołań, co pozwala na wykonywanie innego kodu i wywoływanie w normalny sposób funkcji (lokalne zmienne przechowywane na stosie tego wątku). Wątki mogą znajdować się w różnych stanach, co pokazano na rysunku 4.2.



Rys. 4.2. Stany wątku

Programowanie z użyciem wątków jest skomplikowane, ponieważ większość programów wielowątkowych w systemie wieloprocesorowym może wykonywać się w tym samym czasie. Dość skomplikowane jest debugowanie wielowątkowego programu, gdyż trudno jest powtórzyć zachowanie programu (raz będzie działał dobrze, innym razem się zawiesi – nie ma sposobu na to, aby zmusić system do uszeregowania wątków tak samo). Problemy z wielowątkowością wynikają z jednoczesnego korzystania z tych samych danych.

4.3. Anulowanie wątków

Normalnie wątek kończy się, gdy jego funkcja zakończy działanie albo zostanie wywołana funkcja `pthread_exit ()` o następującej postaci:

```
int pthread_exit(pthread_t id_watku)
```

Aby jeden wątek mógł przerwać inny, należy użyć funkcji `pthread_cancel()` i przekazać jej ID wątku, który ma być usunięty. Anulowany wątek zwraca specjalną wartość `PTHREAD_CANCELED`. Wątek pod względem anulowania może znajdować się w jednym z trzech stanów:

- Może być asynchronicznie anulowalny – w każdej chwili wykonania można go anulować.
- Może być synchronicznie anulowalny – żądania anulowania są kolejkowane, a wątek jest anulowany, gdy osiągnie określone miejsce wykonania.
- Może nie być anulowalny – żądanie anulowania jest ignorowane.

Aby zmienić sposób anulowania wątku na asynchronicznie anulowalny, należy użyć funkcji `pthread_setcanceltype()`, podając jako pierwszy argument `PTHREAD_CANCEL_ASYNCHRONOUS`. Drugi argument, który można pozostawić

pusty, jest wskaźnikiem zmiennej, w której zostanie zachowany poprzedni typ anulowania wątku:

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL)
```

Aby można było powrócić do stanu synchronicznego anulowania, pierwszy parametr musi być PTHRAD_CANCEL_DEFERRED

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL)
```

Punkty anulowania wątków tworzone są funkcją `pthread_cancel()`, która służy do przetwarzania czekających żądań anulowania w synchronicznie anulowalnym wątku. Trzeba ją wywoływać okresowo podczas długich obliczeń w funkcji wątku, w miejscach, gdzie można wątek przerwać bez wycieku zasobów lub innych niepożądanych efektów.

5. Skrypty powłoki bash[5]

Powłoka bash jest najpopularniejszą, a zarazem domyślną powłoką systemu Linux. W porównaniu do podstawowej wersji powłoki Bourne'a, **bash** (czyli *Bourne-Again Shell*) zawiera wiele zaawansowanych możliwości zaczerpniętych od innych, chętnie stosowanych przez użytkowników systemu Unix powłok: **csh** oraz powłoki Korne'a (**ksh**). Poza funkcjami interpretera poleceń oraz zapewnienia interfejsu pomiędzy użytkownikiem a jądrem systemu powłoka jest również bogatym narzędziem programistycznym. Powłoki, zwłaszcza bash, mają możliwości podobne do języków programowania. Można w nich definiować zmienne oraz przypisywać im wartości. Definicje zmiennych, polecenia Linuksa i struktury sterujące można umieszczać w pliku tekstowym (skrypcie), który następnie można uruchomić. Jest on interpretowany przez powłokę.

5.1. Sposoby uruchamiania skryptów powłoki bash

Istnieje kilka sposobów uruchamiania skryptu:

- `. skrypt`
- `source skrypt`
- `skrypt`
- `bash skrypt`

Polecenia `.` oraz `source` umożliwiają interpretację skryptu przez bieżącą powłokę i nie wymagają ustawienia prawa `x` do pliku skryptu, a jedynie `r`. Aby uruchomić skrypt poprzez jego nazwę, plik musi mieć ustawione prawo `x` oraz `r`. Tak uruchomiony skrypt interpretowany jest przez nową powłokę. Polecenie `bash skrypt` to jawne wywołanie nowej powłoki, a plik `skrypt` jest argumentem – plikiem zawierającym dane do wykonania. Aby prześledzić cztery wymienione sposoby uruchamiania skryptów, zdefiniuj dwie zmienne powłoki: lokalną oraz środowiskową, wykonując w powłoce bash polecenia:

```
$ zmlok=cos
$ zmsrod=cossrod
$ export zmsrod
```

[5] Na podstawie prac [3, 6, 7, 9].

Następnie wyedytuj za pomocą edytora plik skrypt o następującej zawartości:

```
echo "wartosc zmiennej lokalnej=Szmlok"
echo "wartosc zmiennej srodowiskowej=Szmsrod"
echo "PID procesu=SS"
```

Zadaniem skryptu jest wyświetlenie wartości zdefiniowanych zmiennych oraz identyfikatora procesu. Zmienna specjalna `SS` jest ustawiana przez powłokę i oznacza identyfikator bieżącego procesu (PID).
Po uruchomieniu skryptu za pomocą polecenia `source` lub polecenia . na ekranie wyświetlana jest zarówno wartość zmiennej lokalnej, jak i wartość zmiennej przekazywanej do nowych powłok poleceniem `export`, a PID procesu odpowiada PID-owi bieżącej powłoki. Dwa pozostałe sposoby – wywołanie skryptu poprzez nazwę oraz jawne wywołanie nowej powłoki ze skryptem jako argument – tworzą nowy proces, czyli nową powłokę, która interpretuje skrypt, dlatego zmienna lokalna nie jest dostępna, a PID procesu różni się od PID-u macierzystego basha.

5.2. Wyrażenia arytmetyczne

Często wykonywaną operacją stosowaną w skryptach powłoki bash jest obliczenie wartości wyrażeń. Można to zrobić za pomocą mechanizmów interpretacji wyrażeń basha, polecenia `let` albo korzystając z zewnętrznego programu `expr`. Istnieją dwa rodzaje mechanizmów basha:

1. `$((wyrażenie))`
2. `$[wyrażenie]`

Przykład

```
a=$((Sa+2))
a=$[Sa+2]
```

Używając polecenia `let`, zmienną a można zmodyfikować w następujący sposób:

```
let a=Sa+2
```

Należy zwrócić uwagę na brak spacji w wyrażeniu. Jeśli zastosuje się spacje, należy użyć znaków ":

```
let a="Sa + 2"
```

Programem, który służy do obliczania wartości wyrażenia jest `expr`. Pobiera on jako argumenty wyrażenie do obliczenia. Każda liczba i każdy znak muszą być oddzielone znakami spacji. Symbole odwróconego cudzysłowu wymuszają wykonanie polecenia, którego wynik stanie się wartością zmiennej.

```
a=`expr $a + 2`
```

W wypadku mnożenia może być konieczne zamaskowanie znaku * za pomocą symbolu \:

```
a=`expr $a \* 2`
```

5.3. Parametry powłoki

W skryptach powłoki bash można odwoływać się do zmiennych określających parametry wywołania skryptu. Ich znaczenie jest następujące:

`$0`	nazwa skryptu
`$1,$2,..$9`	parametry pozycyjne (przekazane do skryptu) lub ustalone ostatnio wykonanym poleceniem `set`. Odwołanie do parametrów o numerach większych niż 9 ma postać: `${nn}`
`$*`	lista parametrów `$1`, `$2`, ... rozdzielonych separatorem (`$IFS`)
`$@`	lista parametrów bez separatorów
`$?`	status zakończenia ostatnio wykonanego polecenia
`$$`	identyfikator (PID) procesu interpretującego skryptu
`$#`	liczba parametrów skryptu

5.4. Zmienne tablicowe

W skryptach powłoki bash można definiować tablice. Tablica jest listą wartości oddzielonych spacjami. Oto przykład definicji tablicy o nazwie `tab`:

```
tab=(kambr ordowik sylur dewon karbon)
```

Kolejne elementy tablicy indeksowane są liczbami całkowitymi, zaczynając od 0, a do *i*-tego elementu tablicy odwołuje się poprzez `${tab[i]}`. `${tab[@]}` i `${tab[*]}` oznaczają wszystkie elementy tablicy. Kolejny element tablicy można dodać poleceniem `tab[i]=`*`nowa_wartosc`*, a do usunięcia elementu

służy polecenie unset. ${#tab[i]}oznacza długość *i*-tego elementu tablicy. Poniższy skrypt ilustruje definiowanie i sposoby odwołania się do elementów tablicy:

```
#!/bin/bash
tab=(kambr ordowik sylur dewon karbon)
echo ${tab[@]}
echo ${tab[*]}
for i in 0 1 2 3 4
do
  echo tab[$i]=${tab[i]}
  echo długosc tab[$i] = ${#tab[i]}
done
tab[5]=perm
echo ${tab[@]}
unset tab[0]
echo ${tab[@]}
```

Kod 5.1. Definiowanie tablic

```
$ bash tablice
kambr ordowik sylur dewon karbon
kambr ordowik sylur dewon karbon
tab[0]=kambr
długosc tab[0] = 5
tab[1]=ordowik
długosc tab[1] = 7
tab[2]=sylur
długosc tab[2] = 5
tab[3]=dewon
długosc tab[3] = 5
tab[4]=karbon
długosc tab[4] = 6
kambr ordowik sylur dewon karbon perm
ordowik sylur dewon karbon perm
$
```

Przykład 5.1. Efekt wykonania kodu 5.1

5.5. Wczytywanie danych

Do wczytywania wartości zmiennych z klawiatury służy polecenie `read`. Jeśli nie podamy nazwy zmiennej, wówczas dane są wczytywane do zmiennej o domyślnej nazwie REPLY. Polecenie:

```
read a
```

wczytuje linie ze standardowego wejścia do zmiennej `a`.
Jednym poleceniem można wczytać wartości większej liczby zmiennych. Przykładowo, polecenie:

```
read a1 a2 a3
```

wczytuje ze standardowego wejścia wartości trzech zmiennych: `a1`, `a2`, `a3`.

5.6. Instrukcje warunkowe

Powłoka bash umożliwia warunkowe wykonywanie poleceń. Dostępne są warunkowe struktury sterujące `if` oraz `case`. W przeciwieństwie do instrukcji sterujących w językach programowania, `if` sprawdza kod wykonania polecenia, a nie wyrażenie. Oto składnia polecenia `if-then-else`:

```
if  polecenie
      then
 polecenia1
      else
 polecenia2
fi
```

Jeśli słowo kluczowe **`then`** występuje w tej samej linii co **`if`**, musi zostać poprzedzone średnikiem:

```
if  polecenie; then
 polecenia
fi
```

Możliwa jest konstrukcja wielopoziomowa instrukcji warunkowej `if-elif-else-fi`

```
if polecenie1
  then
    polecenia1
  elif polecenie2
    then
      polecenia2
  elif polecenie3; then
      polecenia3
    else
      polecenia4
fi
```

Poleceniem często występującym bezpośrednio po słowie kluczowym `if` jest `test` (badające pewien warunek). Jeśli warunek jest spełniony, polecenie `test` zwraca wartość 0, czyli logiczną prawdę, w przeciwnym wypadku wartość niezerową, czyli fałsz. Zamiast słowa kluczowego `test` można również używać nawiasów kwadratowych.

Polecenia:

```
test $a -eq 1
[ $a -eq 1 ]
```

są równoważne i sprawdzają, czy wartość zmiennej a wynosi 1. Należy pamiętać o znakach spacji po znaku [i przed znakiem]. Do budowania warunków można stosować wiele operatorów. Najczęściej używane to:

- **operatory logiczne:**

  `! wyr` – negacja,
  wyr1 `-a` *wyr2* – logiczne AND,
  wyr1 `-o` *wyr2* – logiczne OR (lub),

- **operatory do porównania łańcuchów znakowych:**

  ciag1`=`*ciag2* – ciągi identyczne,
  ciag1`!=`*ciag2* – ciągi różne,
  `-z` *ciag* – ciąg pusty,
  `-n` *ciag* – ciąg nie jest pusty,
  `ciag` – ciąg nie jest pusty,

- testowanie plików:

  `-f plik` – plik zwykły,
  `-d plik` – katalog,
  `-r plik` – plik do odczytu,
  `-w plik` – plik do zapisu,
  `-s plik` – plik niepusty,
  `-x plik` – plik wykonywalny,

- porównywanie liczb:

```
wyr1 -eq wyr2,
wyr1 -ne wyr2,
wyr1 -lt wyr2,
wyr1 -le wyr2,
wyr1 -gt wyr2,
wyr1 -ge wyr2.
```

Należy zwrócić uwagę, że do porównywania ciągów znaków używa się operatora =, natomiast do porównywania liczb operatora `-eq`.

W powłoce bash można korzystać z konstrukcji `case`, która umożliwia porównanie zmiennej z wieloma wzorcami i w zależności od wyniku dopasowania wykonanie odpowiedniej akcji. Składnia instrukcji warunkowej `case` jest następująca:

```
case napis in
     wart1)
polecenia1
      ;;
     wart2)
polecenia2
      ;;

     .....

       *)
    polecenia3
      ;;
  esac
```

gdzie napis może być odwołaniem do zmiennej – np. $a. Wartość napis porównywana jest kolejno ze wszystkimi wzorcami wart1, wart2 i wykonywane jest polecenie przypisane do danej wartości. Jeśli żadne dopasowanie nie zakończy się sukcesem, wykonywane są polecenia domyślne występujące po symbolu *. Należy zwrócić uwagę na podwójny znak średnika kończący każdy potok instrukcji. Jeśliby go zabrakło, powłoka wykonałby kolejne instrukcje.

5.7. Pętle

W powłoce bash dostępne są trzy typy struktur sterujących, tworzących pętle: for, while oraz until.
Pętla for powoduje wykonanie ciągu instrukcji dla każdego elementu z listy. Składnia jest następująca:

```
for zmienna in lista_wartosci
      do
polecenia
      done
```

Powtarzany ciąg poleceń jest zawarty między słowami kluczowymi do a done. Lista wartości może być podana jawnie lub za pomocą metaznaków, tak jak w przykładzie:

```
for   plik  in   *.c
do echo plik $plik
cat $plik|more
done
```

Kod 5.2. Skrypt wykorzystujący pętlę for z listą wartości

Uruchomienie powyższego skryptu powoduje wyświetlenie nazw i zawartości wszystkich plików źródłowych w języku C z katalogu bieżącego. Jeśli nie występuje *lista_wartosci* ani nie pojawia się słowo kluczowe in, polecenia pętli wykonywane są dla wszystkich parametrów skryptu. Tak więc pętla for bez listy (for *zmienna*) jest równoważna pętli:

```
for zmienna in $@
```

Przykładem pętli `for` bez listy jest poniższy skrypt obliczający sumę wszystkich parametrów wywołania:

```
s=0
for skl
do
s=$[$s+$skl]
done
echo suma=$s
```

Kod 5.3. Skrypt wykorzystujący pętlę `for` bez listy wartości

W powłoce bash dostępna jest również pętla `for` składniowo zbliżona do pętli `for` z języka C. Ilustruje to poniższy przykład:

```
for((i=0;i<5;i++));do echo "---$i---";done
```

Należy zwrócić uwagę na fakt, że wyrażenia sterujące pętlą umieszczone są w podwójnych nawiasach. Podobnie jak w wypadku struktury sterującej `if`, w składni pętli obowiązuje zasada, że jeśli słowo kluczowe `do` znajduje się w tej samej linii, co słowo kluczowe rozpoczynające pętlę (`for`, `while`, `until`), musi być poprzedzone przecinkiem. Podobnie jest ze słowem kluczowym `done` kończącym pętlę.

Pętle `while` oraz `until` powodują powtarzanie ciągu poleceń w zależności od wartości logicznej wskazanego polecenia (zwykle jest to polecenie `test`).

```
while polecenie
    do
     polecenia
    done
```

```
until polecenie
  do
 polecenia
  done
```

Obie pętle wykonują polecenie testujące przed wejściem do pętli. Różnica polega na tym, że pętla `while` jest wykonywana, gdy *`polecenie`* zakończy się sukcesem (zwraca kod równy 0), natomiast `until` powtarza ciąg poleceń przy niezerowym kodzie zakończenia polecenia *`polecenie`*.

Poniższe skrypty ilustrują, jak stosując pętlę `while` oraz pętlę `until`, obliczyć sumę wszystkich parametrów wywołania:

```
s=0
until [ ! $1 ]
do
s=$[$s+$1]
shift
done
echo suma=$s
```

Kod 5.4. Skrypt wykorzystujący pętlę until

```
s=0
while [ $1 ]
do
s=$[$s+$1]
shift
done
echo suma=$s
```

Kod 5.5. Skrypt wykorzystujący pętlę while

Zastosowano w nich polecenie `shift`, które przesuwa zmienne parametryczne o jedną pozycję. Pierwsze wywołanie polecenia `shift` w skrypcie powoduje, że wartość `$0` pozostaje niezmieniona, natomiast `$1` przyjmuje wartość drugiego parametru, `$2` będzie odpowiadało trzeciemu parametrowi itd. Dzięki zastosowaniu polecenia `shift` w pętli do wszystkich parametrów skryptu, bez względu na ich liczbę, można się odwoływać przez `$1`.
Innym sposobem na tworzenie pętli jest użycie struktury sterującej `select..do`. Jej składnia jest następująca:

```
select zmienna in lista
do
  polecenie
done
```

Na podstawie zadanej listy tworzy ona na ekranie ponumerowane menu i wyświetla znak zachęty, określony przez zmienną powłoki `PS3`. Użytkownik dokonuje wyboru poprzez podanie liczby odpowiadającej wybranej pozycji i wykonywane są wtedy instrukcje pętli (zwykle jest to `case`). Działanie pętli `select..do` ilustruje poniższy skrypt.

```
#!/bin/bash
select a in a b c koniec
do
case $a in
"a") echo "a"
;;
"b") echo "b"
;;
"c") echo "c"
;;
"koniec") exit
;;
*) echo "zly wybor"
;;
esac
done
```

Kod 5.6. Skrypt wykorzystujący polecenie `select`

A oto sposób wywołania i efekt działania skryptu:

```
$ PS3=-----
$ export PS3
$ bash skselect
1) a
2) b
3) c
4) koniec
-----1
a
-----2
b
-----3
c
-----4
$
```

Przykład 5.2. Efekt wykonania skryptu kod 5.6

Wywołanie skryptu poprzedza zdefiniowanie zmiennej PS3. Wybory użytkownika (1, 2, 3, 4) poprzedzone są znakami zachęty "-----". Wybranie opcji 1–3 powoduje wyświetlenie na ekranie aktualnego wyboru, natomiast 4 kończy pętlę (wykonanie polecenia exit).

5.8. Potoki i listy poleceń

Do skrócenia zapisu w skryptach powłoki bash stosuje się listy poleceń. Lista poleceń jest sekwencją poleceń lub potoków rozdzielonych jednym z następujących symboli: ;, &, &&, || i opcjonalnie zakończonych znakiem ; lub &. Symbole ; i & mają taki sam priorytet, niższy od priorytetów symboli && i || (równorzędnych).

- ; oznacza sekwencyjne wykonywanie poleceń.
- Potok poprzedzony znakiem & jest wykonywany w tle, a kolejne polecenie jest wykonywane natychmiast.
- Symbol && oznacza koniunkcję – polecenie występujące po nim zostanie wykonane tylko wtedy, gdy polecenie poprzedzające go zakończy się powodzeniem (kodem 0).
- Symbol || oznacza alternatywę – polecenie występujące po nim zostanie wykonane tylko wtedy, gdy polecenie poprzedzające go zakończy się niepowodzeniem (kodem błędu).

Oto przykład dwóch równoważnych skryptów, których zadaniem jest sprawdzenie, czy istnieje plik .profil.

```
if [ -f .profil ]; then
exit 0
fi
exit 1
```

```
[ -f .profil ] && exit 0 || exit 1
```

Skrypt z kolejnego przykładu sprawdza, czy pierwszy parametr wywołania jest katalogiem z prawem wykonania.

```
if [ -d $1 ] && [ -x $1 ]
then
    echo "katalog z prawem x"
else

    echo "nie katalog lub brak praw x"
fi
```

W powyższym skrypcie zastosowano operator koniunkcji && dotyczący dwóch poleceń testujących wartość wyrażenia. Skrypt ten można napisać również stosując jeden operator `test: [ ]` ze złożonym warunkiem zbudowanym za pomocą operatora koniunkcji `-a`, tak jak w przykładzie poniżej:

```
if [ -d $1 -a -x $1 ]
then
    echo "katalog z prawem x"
else
    echo "nie katalog lub brak praw x"
fi
```

Za pomocą listy poleceń ten sam skrypt można zapisać znacznie krócej, ale za to mniej czytelnie:

```
[ -d $1 ] && [ -x $1 ] echo "kat. z x"|| echo "nie"
```

Do pełnego diagnozowana przypadków elementarnych konieczne jest zastosowanie złożonej struktury `if`:

```
if [ -d $1 ]
then  if [ -x $1 ]
      then
        echo "katalog z prawem x"
      else
        echo "katalog bez prawa x"
      fi
else
 echo „nie katalog"
fi
```

5.9. Definiowanie funkcji

W bashu także można definiować funkcje – zbiory instrukcji, które często powtarzają się w programie. Funkcję definiuje się w następujący sposób:

```
[function] nazwa_funkcji()
{
 instrukcje
}
```

Słowo kluczowe `function` może zostać pominięte. Funkcję w skrypcie wywołuje się, podając jej nazwę i parametry: *`nazwa_funkcji parametry`*. Wewnątrz funkcji parametry są widoczne jako zmienne `$1`, `$2`, `$*` itd., przysłaniając parametry wywołania skryptu. Wyjście z funkcji odbywa się po wykonaniu polecenia `return` lub ostatniego polecenia w funkcji. Zwracany jest kod wyspecyfikowany w poleceniu `return` lub kod zakończenia ostatniego polecenia funkcji. Po zakończeniu funkcji parametry pozycyjne przyjmują poprzednie wartości.

Oto przykład prostego skryptu bash wykorzystującego funkcję, w której wyświetlana jest zachęta do podania wartości z klawiatury:

```
#!/bin/bash
function naglowek
{
echo "podaj skladnik"
}

for ((i=0;i<5;i++))
do
    naglowek
    read i
    s=$(($s + $i))
done
echo suma=$s
```

Kod 5.7. Skrypt pobierający dane z klawiatury

Wygodnie jest umieścić funkcje w innym pliku (np. `funkcja`). W skrypcie odwołanie do funkcji musi być poprzedzone dołączeniem pliku zawierającego definicje funkcji za pomocą polecenia `. funkcja`.

Oto przykład skryptu odwołującego się do funkcji nagłówek zdefiniowanej w pliku funkcja oraz przykładowy plik o nazwie funkcja zawierający definicję funkcji:

plik funkcja:

```
#!/bin/bash
function naglowek
{
echo podaj składnik
}
```

skrypt

```
#!/bin/bash
. funkcja
for ((i=0;i<5;i++))
do
  naglowek
  read ii
  s=$(($s + $ii))
done
echo suma=$s
```

Kod 5.8. Skrypt odwołujący się do funkcji zapisanej w innym pliku

Poniżej przedstawiony jest skrypt, który rysuje na ekranie trójkąt prostokątny o zadanej wysokości, wypełniony zadanym znakiem. Wykorzystana została funkcja linia, która rysuje pojedynczą linię. Jej parametrami są: liczba początkowych spacji, liczba kolejnych znaków oraz znak.

```
#!/bin/bash
linia()
{
for((i=0;i<$1;i++))
do
echo -n " "
done
for((i=0;i<$2;i++))
do
echo -n $3
done
}
echo -n "wciecie="
read sp
echo -n "znak="
read zn
echo -n "wysokosc="
read r
for((k=1;k<=$r;k++))
```

```
do
linia sp k $zn
echo
sp=$[$sp-1]
done
```

Kod 5.9. Skrypt rysujący trójkąt o zadanej wysokości

A oto przykładowe wywołanie skryptu:

```
$ bash trojkat
wciecie=20
znak=$
rozmiar=10
                    $
                   $$
                  $$$
                 $$$$
                $$$$$
               $$$$$$
              $$$$$$$
             $$$$$$$$
            $$$$$$$$$
$
```

Przykład 5.3. Efekt wykonania skryptu kod 5.9

5.10. Przechwytywanie sygnałów

W skryptach powłoki bash istnieje możliwość przechwytywania sygnałów przychodzących do procesu. Służy do tego polecenie `trap`. Pozwala ono na zastąpienie klasycznej reakcji na sygnał przychodzący do procesu wskazanym poleceniem lub ignorowanie go. Dwa typy sygnałów: SIGKILL (9) i SIGSTOP (19) nie mogą być przechwycone. Składnia polecenia `trap` jest następująca:

`trap` *`polecenie sygnał`*

Listę wszystkich sygnałów można uzyskać poleceniem `kill -l`. Przy wywoływaniu polecenia `trap` można używać zarówno numeru sygnału, jak i jego symbolicznej nazwy.

W poniższym przykładzie obsługę sygnału INT zastąpiono poleceniem wyświetlającym tekst: `sygnal`. Sygnał INT o kodzie 2 można wysłać do procesu za pomocą kombinacji klawiszy Ctrl+c lub poleceniem:

`kill -2 PID_procesu` wydanym z innego terminala

```
trap 'echo sygnal' INT
for ((i=1;i>0;i++))
do
sleep 2
done
```

Skrypt powoduje wykonanie nieskończonej pętli. Próby zakończenia go poprzez wysłanie sygnału SIGINT powodują jedynie wyświetlenie na ekranie tekstu `sygnal`. Można go natomiast zakończyć, wysyłając sygnał SIGKILL (9) lub zatrzymując go za pomocą sygnału SIGSTOP (19), któremu odpowiada kombinacja klawiszy Ctrl+z, a następnie polecenia `kill %`.

5.11. Przykłady skryptów

1. Skrypt porównuje rozmiary dwóch plików, których nazwy są parametrami skryptu. Na początku sprawdza liczbę argumentów i ich poprawność.

```
if [ $# -ne 2 ]
then
echo blad: zla liczba parametrow
exit
fi
echo liczba argumentow=$#
if [ ! -f $1 ]
then
  echo blad: nie ma pliku $1
  exit
fi
if [ ! -f $2 ]
then
  echo blad: nie ma pliku $2
  exit
fi
d1=`cat $1|wc -c`
```

```
d2=`cat $2|wc -c`
echo $d1
echo $d2
if [ $d1 -gt $d2 ]
then
  echo „wiekszy jest plik $1"
else
  echo „wiekszy jest plik $2"
Fi
```

Kod 5.10. Porównanie rozmiaru dwóch plików

Na uwagę zasługuje konstrukcja `d1=`cat $1|wc -c`, która do zmiennej d1 podstawia liczbę bajtów pliku będącego pierwszym parametrem. Zastosowanie potoku ma na celu wyeliminowanie ze strumienia wyjściowego nazwy pliku. Polecenie wc przetwarza strumień wynikowy polecenia cat i wynikiem jest jedynie jego rozmiar, bez nazwy pliku (polecenie wc -c plik zwraca dwie wartości: nazwę i rozmiar pliku).
Alternatywnym rozwiązaniem jest zastosowanie polecenia set, które ustawia zmienne parametryczne, ale przysłaniając parametry wywołania. W poniższym skrypcie zmienna parametryczna $1 będzie miała wartość równą liczbie bajtów pliku będącego parametrem.

```
set $(wc -c $1)
echo pierwszy --- $1
echo drugi ------ $2
```

Kod 5.11. Wykorzystanie polecenia set

```
$ bash set2 set2
pierwszy --- 70
drugi ------ set2
$
```

Przykład 5.4. Efekt wykonania kodu 5.11

2. Skrypt zamieniający nazwy dwóch plików będących parametrami skryptu.

```
if [ $# -ne 2 ]
then
echo blad: zla liczba parametrow
exit
fi
echo liczba parametrow=$#
if [ ! -f $1 ]; then
 echo blad: nie ma pliku $1
fi
if [ ! -f $2 ]; then
 echo blad: nie ma pliku $2
fi
mv $1 plik$$
mv $2 $1
mv plik$$ $2
```

Kod 5.12. Skrypt zamieniający nazwy plików

W skrypcie zastosowano bezpieczny sposób tworzenia pliku tymczasowego: `plik$$` z wykorzystaniem zmiennej `$$` określającej unikalny identyfikator procesu.

3. Skrypt wczytuje nazwy plików i sprawdza, czy istnieje taki plik i czy ma ustawione prawa do odczytu. Sprawdzanie powtarza się dla wszystkich wprowadzanych wartości z klawiatury, do podania pustej nazwy. Gdy nic podamy ani jednej wartości, skrypt kończy działanie z kodem błędu 4.

```
#!/bin/bash
read plik
[ ! $plik ] && exit 4
while [ $plik ]
do
if [ -f $plik ]
then
  if [ -r $plik ]
  then
    echo moge czytac $plik
  else
    echo nie do odczytu $plik
  fi
```

```
else
  echo nie ma takiego pliku $plik
fi
read plik
done
```

Kod 5.13. Skrypt sprawdzający istnienie pliku

4. Skrypt oblicza sumę danych będących parametrami wywołania skryptu (od drugiego parametru) do napotkania wartości równej pierwszemu parametrowi.

```
#suma bez konca
s=0
kon=$1
shift
until [ $1 -eq $kon ]
do
s=$[$s+$1]
shift
done
echo suma=$s
```

```
#suma bez konca
s=0
kon=$1
shift
while [ $1 -ne $kon ]
do
s=$[$s+$1]
shift
done
echo suma=$s
```

Kod 5.14. Skrypt z parametrami

5. Skrypt wyświetla menu zdefiniowane w funkcji i realizuje polecenia w zależności od wyboru. Zastosowano polecenie `break`, kończące wykonywanie bieżącej pętli i przejście do pierwszego polecenia po tej pętli, oraz `continue`, przerywające działanie bieżącej iteracji pętli i przejście do kolejnej.

```
#!/bin/bash
naglowek()
{
echo  k - koniec
echo  l - ls -l
echo  w - who
echo  d - date
}
while [ 0 ]
do
  naglowek
  read wybor
```

```
   case $wybor in
   l)
      ls -l;;
   w)
      who;;
   d)
      date;;
   k)
      echo "czy na pewno koniec T N"
      read wybor
      case $wybor in
      T|t)
          break;;
      N|n)
          continue;;
      esac;;
   *)
      echo zla opcja
continue
esac
done
```

Kod 5.15. Skrypt z definicją menu wyboru

6. Język AWK[6]

AWK jest narzędziem służącym do przetwarzania tekstu i generowania raportów według zdefiniowanych reguł. Wiele poleceń systemu Unix przetwarza strumień wejściowy lub pliki z danymi, tworząc strumień wynikowy. Nazwa AWK pochodzi od pierwszych liter nazwisk jego twórców: Aho, Weinberga, Kernighana. AWK jest bardzo wygodnym narzędziem dającym duże możliwości: wykonywanie operacji na ciągach znaków i operacji arytmetycznych, korzystanie z konstrukcji programowania strukturalnego. Jest interpreterem, dzięki czemu programy łatwo można uruchamiać pod innym systemem operacyjnym.
AWK można uruchomić w następujący sposób:

```
awk -f program plik1 plik2 plik3...
```

W tym wypadku AWK przetwarza kolejno linie z plików występujących w linii poleceń: `plik1`, `plik2` i kolejne, wykonując na nich program zapisany w pliku program.
Jeśli kod programu nie jest długi, można umieścić go bezpośrednio w poleceniu pomiędzy znakami pojedynczego cudzysłowu:

```
awk 'kod programu' plik1
```

Program języka AWK składa się z par:

wzorzec {akcja}

Wzorzec jest wyrażeniem logicznym, które jest dopasowywane kolejno do każdej linii każdego z plików wejściowych. Jeśli wyrażenie jest prawdziwe, wykonywana jest akcja. Jeśli wzorzec nie występuje, akcja jest wykonywana dla każdej linii. Najprostszy wzorzec to wyrażenie regularne występujące między znakami /. Polecenie:

```
awk '/Jan/ {print $0}' /etc/passwd
```

wyświetla linie z pliku /etc/passwd, w których występuje słowo `Jan`. Akcja `print $0` oznacza wyświetlenie całej linii wejściowego pliku. Jeśli akcja jest pominięta, wykonywana jest akcja domyślna, którą jest właśnie `print $0`. Zatem polecenie:

```
awk '/Jan/' /etc/passwd
```

[6] Na podstawie pracy [6].

jest równoważne poprzedniemu. Jeśli w programie AWK występuje więcej par **wzorzec**, **akcja**, są one oddzielone średnikami lub znakiem końca linii. Poniższy przykład ilustruje, jak można z pliku /etc/passwd wybrać wpisy opisujące użytkowników o imieniu `Jan` oraz `Maria` i wyświetlić w sposób sformatowany (w równych kolumnach) jedynie wybrane dane: imię i nazwisko (5 kolumna) oraz nazwę logowania (1 kolumna). Spacja występująca we wzorcach po imionach umożliwia precyzyjne dopasowanie.

```
awk  -F: '/Jan / {printf "%-20s %s\n",$5,$1}
          /Maria / {printf "%-20s %s\n",$5,$1}'
/etc/passwd
```

Opcja `-F` pozwala zdefiniować separator pól – w powyższym przykładzie będzie to znak dwukropka. Funkcja `printf` pozwala sformatować wyjście podobnie jak w języku C, natomiast `$1`, `$2`, ... oznaczają kolejne pola linii tekstu wejściowego przy zadanym separatorze.

Akcja może się również składać z wielu poleceń, wówczas oddzielone są one średnikami lub znakiem nowej linii. W pisaniu programów awk bardzo przydatne jest stosowanie bloku `BEGIN`, w którym definiuje się akcje wykonywane przed przetwarzaniem tekstu wejściowego, oraz bloku `END`, który jest wykonywany na końcu, po przetworzeniu wszystkich wierszy. Ilustruje to poniższy przykład, w którym zliczani są użytkownicy o imionach `Jan` i `Maria`:

```
awk -F: 'BEGIN {printf "\n\n";i=0;j=0}
          /Jan / {printf "%-20s %s\n",$5,$1; i++}
          /Maria / {printf "%-20s %s\n",$5,$1; j++}
          END {printf "linii-%s,Janow-%s, Marii-%s\n",
          NR,i,j}'
/etc/passwd
```

Kod 6.1. Przykład wykorzystania AWK

W programie awk zmienne nie wymagają deklaracji i inicjowane są wartością 0, więc zwykle nie jest konieczne inicjowanie ich w bloku `BEGIN`. Wbudowana zmienna `NR`, wykorzystana w tym programie, wskazuje numer aktualnie przetwarzanego wiersza – jej wartość na końcu programu jest równa liczbie przetworzonych wierszy.

Program awk może być wywołany w skrypcie powłoki. Jeśli w programie awk chcemy się odwołać do zmiennych skryptu, w tym również parametrów skryptu, możemy je przekazać za pomocą opcji `-v`. Ilustruje to poniższy skrypt basha, który podobnie jak poprzedni program, wyświetla informacje o wybranych użytkownikach systemu, lecz tym razem imiona są parametrami skryptu.

Skrypt skawk:

```
awk -v N1="$1 " -v n2=$2 -F:
'BEGIN {i=0;j=0;N2=n2 " ";printf "\n %s %s\n",N1,N2}
{if (index($0,N1) !=0) printf "%-20s %s\n",$5,$1;i++}
{if (index($0,N2) !=0) printf "%-20s %s\n",$5,$1;j++}
END {printf "\n\n linii=%s %s - %s %s -
%s\n",NR,N1,i,N2,j}' /etc/passwd
```

Kod 6.2. Przykład wykorzystania parametrów skryptu w programie awk

Przykładowe uruchomienie skryptu:

```
bash skawk Jan Maria
```

Opcja `-v` powoduje wprowadzenie zmiennych skryptu powłoki bash do programu awk. W tym przykładzie do programu awk przekazywane są parametry skryptu: `$1` i `$2`. Zmienna `N1` odpowiada parametrowi pierwszemu z dodaną spacją na końcu, a zmienna `n2` – parametrowi drugiemu. Dodanie spacji na końcu zmiennej można wykonać również jak w przykładzie: `N2=n2 " "`. W programie wykorzystane zostały elementy zaczerpnięte z języka C: instrukcje arytmetyczne, instrukcja sterująca `if` oraz funkcja `index`, która wyszukuje w bieżącej linii (pierwszy parametr funkcji `index` ma wartość `$0`) ciąg znaków odpowiadający zmiennej `N1` (drugi parametr) i zwraca jego położenie. Kolejne wywołanie funkcji `index` dotyczy zmiennej `N2`.

Innym sposobem odwołania się do parametrów skryptu bash w programie awk jest ich ujęcie w pojedyncze cudzysłowy: `'$1'`. Pokazuje to poniższy przykład – skrypt realizujący to samo zadanie, co poprzedni:

```
awk  -F: 'BEGIN {printf "\n\n";i=0;j=0}
          /'$1' / {printf "%-20s %s\n",$5,$1; i++}
          /'$2' / {printf "%-20s %s\n",$5,$1; j++}
          END {printf "linii-%s,Janow-%s, Marii-%s\n",
          NR,I,j}'
/etc/passwd
```

Kod 6.3. Przykład odwołania do parametrów skryptu bash w programie awk

Wzorzec występujący w programie AWK może mieć postać wyrażenia regularnego, analogicznego do wyrażeń występujących w poleceniu `grep`.

W poniższym przykładzie program AWK przetwarza wyjście polecenia `ls -l`, wybiera linie opisujące pliki zwykłe, w odniesieniu do których członkowie grupy mają prawo czytania, i sumuje rozmiary takich plików. Na standardowym strumieniu wynikowym wyświetla policzoną sumę.

```
ls -l|awk '/^-...r/ {x=x+$5}; END {print  x}'
```

Przykłady skryptów powłoki bash wykorzystujących program AWK:

1. Skrypt badający zajętość file-systemów, dopisujący aktualne wyniki wraz z aktualną datą i czasem do pliku plik

```
DAT=`date +"%Y-%m-%d %T"`
echo $DAT
df|awk -v DT="$DAT" '{if (NR != 1) printf "%s %-20s
%-20s\n", DT,$6,$5}' >> plik
```

2. Skrypt wyświetlający informacje o wykonywalnym pliku będącym parametrem skryptu.

```
zm=`whereis $1|awk '{print $2}'`
echo $zm
if  [ $zm ]
then
ls -ld $zm|awk '{print $9,$3,$5}'
else
echo brak
fi
```

3. Trzy różne skrypty zapisujące do pliku plik1 linie z pliku będącego pierwszym parametrem skryptu z zakresu określonego przez dwa pozostałe parametry.

```
awk -v z1=$2 -v z2=$3 'NR>z1 && NR<=z2' $1 >plik1
```

```
awk -v z1=$2 -v z2=$3 'NR>z1 && NR<=z2 {print $0>
"plik1"}' $1
```

```
cat $1|awk 'NR>'$2' && NR <'$3' {print $0 > "plik1"}
```

7. Funkcje systemowe w przykładach[7]

7.1. Operacje na plikach

Programy użytkownika systemu Linux odwołują się do jądra systemu poprzez funkcje systemowe. Tylko za ich pośrednictwem możliwe jest np. korzystanie z systemu plików czy też mechanizmów komunikacji między procesami. Wywołanie funkcji niskopoziomowych, które udostępniają plik (`open`), pozwala na wykonanie operacji odczytu (`read`) i zapisu (`write`) na pliku oraz zamknięcie pliku (`close`), odwołując się bezpośrednio do sterowników urządzeń. Oznacza to przełączenie do pracy w trybie jądra przy każdym takim wywołaniu i wiąże się często z małą wydajnością. Dlatego w systemie Linux dostępne są biblioteki, które optymalizują wykonywane operacje, jak np. standardowa biblioteka wejścia/wyjścia z plikiem nagłówkowym stdio.h, wykorzystująca bufory o rozmiarach dostosowanych do sprzętu.

Niskopoziomowe funkcje dostępu do plików korzystają z deskryptorów plików, natomiast funkcje wysokopoziomowe używają odpowiadających im strumieni plikowych. Każdy uruchomiony program ma dostępne trzy strumienie plikowe, zadeklarowane w pliku stdio.h, którym odpowiadają deskryptory zamieszczone w tabeli 7.1.

Tabela 7.1

Standardowe strumienie plikowe

	Deskryptor	Strumień plikowy
Standardowe wejście	0	stdin
Standardowe wyjście	1	stdout
Błędy	2	stderr

Każdy otwarty plik w programie otrzymuje deskryptor będący kolejną liczbą całkowitą. Można się do niego odwołać również poprzez link symboliczny z podkatalogu katalogu proc opisującego dany proces. Ilustruje to poniższy przykład 7.1 wraz z zapisem fragmentu sesji:

[7] Na podstawie prac [6, 7, 10].

```
$ ps
  PID  TTY          TIME CMD
19685 pts/1    00:00:00 bash
19868 pts/1    00:00:00 sem
19912 pts/1    00:00:00 ps
$ ls -l /proc/19868/fd
razem 0
lrwx------ 1 anka anka 64 04-27 11:09 0 -> /dev/pts/1
lrwx------ 1 anka anka 64 04-27 11:09 1 -> /dev/pts/1
lrwx------ 1 anka anka 64 04-27 11:09 2 -> /dev/pts/1
lrwx------  1  anka  anka  64  04-27  11:09  3  ->
/home/anka/dane
$
```

Przykład 7.1. Pliki użytkowane przez proces, którego PID = 19868

W programie sem otwarty został plik użytkownika anka o nazwie dane. Identyfikator (PID) uruchomionego procesu to 19868. W katalogu proc jest podkatalog opisujący ten proces, a w nim katalog fd zawierający deskryptory plików procesu. Znajdujące się tu wpisy odpowiadają deskryptorom plików procesu i są linkami symbolicznymi do używanych w procesie plików, a deskryptor 3 odpowiada plikowi dane.

Zadanie
Napisz program w języku C, który testuje prawo SUID do pliku. Do pliku dane w swoim katalogu domowym zablokuj prawa dostępu dla członków grupy. W programie program otwórz plik dane do odczytu i zapisu i zapisz do niego jakieś dane, po czym odczytaj te dane. Ustaw prawa dostępu, które umożliwiają członkom grupy uruchomienie programu. Przetestuj program i poproś użytkownika należącego do tej samej grupy o przetestowanie jego działania. Ustaw prawo SUID do pliku i ponownie wykonaj testy.

7.1.1. Semantyka spójności

Semantyka spójności systemu plików przewiduje, że efekty modyfikacji pliku są natychmiast dostępne w innych procesach używających tego samego pliku. Można się o tym przekonać w najprostszy sposób, otwierając dwa okna z powłoką pod systemem. Jeśli w jednym z nich edytujemy plik dane poleceniem: cat >dane, to w drugim oknie, wykonując co pewien czas polecenie cat dane, widzimy aktualną wersję pliku.

Natychmiastowe uaktualnianie danych w plikach edytowanych przez kilka procesów mogłoby prowadzić do niezamierzonych efektów. Rozważmy następujący

przykład. Jest to program w języku C, który odwołuje się do pliku na dysku, zapisując do niego pewne dane, a następnie odczytuje dane z tego pliku. W czasie działania procesu inny proces modyfikuje plik z danymi. Kod programu przedstawiony jest poniżej.

```
#include <studio.h>
main()
{
int i,ii;
FILE *plik;
plik=fopen("dane","r+");
for (i=0;i<3;i++)
  fprintf(plik,"%d\n",i);
scanf("%d",&i);
fseek(plik,0,SEEK_SET);
for (i=0;i<5;i++)
  {
   fscanf(plik,"%d",&ii);
   printf("\t%d\n",ii);
  }
}
```

Kod 7.1. Program ilustrujący semantykę spójności

Do pliku dane zapisywane są w programie liczby 0, 1, 2. Następnie funkcja `fseek` ustawia wskaźnik odczytu pliku na początek pliku i odczytywanych jest 5 pierwszych liczb z pliku dane. Jeśli pomiędzy zapisem a odczytaniem danych z pliku (przed wywołaniem funkcji `fseek`, gdy nasz proces czeka na dane dla funkcji `scanf`) inny proces próbowałby nadpisać plik dane, np. wprowadzając do niego pięć liczb 9, ostateczna zawartość pliku byłaby następująca: 0 1 2 9 9. Wynika to z faktu, że zapis do pliku następuje dopiero po minięciu się bieżących wskaźników do pliku we wszystkich procesach, w których plik jest otwarty. Wskaźnik do pliku dane w naszym procesie był ustawiony po liczbie 2, zatem kolejny proces dopisał jedynie dwie liczby 9. Jeśli modyfikacja pliku przez inny proces nastąpiłaby po ustawieniu wskaźnika na początku pliku (co można uzyskać, przestawiając kolejność wywołań funkcji `scanf` i `fseek`), to do pliku zostałyby zapisane same liczby 9. Zatem jeśli proces zapisał do pliku jakieś dane, można oczekiwać, że nie będą one zmodyfikowane, dopóki wskaźnik do bieżącego rekordu w pliku nie zostanie zmodyfikowany w taki sposób, że zostanie przesunięty ponad te dane.

7.1.2. Nisko- i wysokopoziomowe operacje wejścia/wyjścia

Kolejne przykłady ilustrują różnice między nisko- i wysokopoziomowymi operacjami wejścia/wyjścia. Poniżej (Kod 7.2 i Kod 7.3) przedstawiono dwa programy, za pomocą których można łatwo porównać sposoby wyprowadzania informacji z programu.

```
#include <stdio.h>
//niskopoziomowo
// bez buforowania

main()
{
int i=0;
while (i<20)
{
  write(1,"1",1);
  sleep(1);
  i++;
};
i=0;
while (i<10)
{
  write(2,"2",1);
  sleep(1);
  i++;
}
}
```

Kod 7.2. Niskopoziomowe operacje wejścia/wyjścia

```
#include <stdio.h>
//wysokopoziomowo

main()
{
int i=0;
while (i<10)
{
  printf("1");
  sleep(1);
//fflush(stdout);
  i++;
};
i=0;
while (i<10)
{
  fprintf(stderr,"2");
  sleep(1);
  i++;
}
}
```

Kod 7.3. Wysokopoziomowe operacje wejścia/wyjścia

Uruchamiając pierwszy program, przekonamy się, że gdy korzystamy z funkcji niskopoziomowych zarówno strumień wynikowy, jak i strumień błędów natychmiast są wyświetlane na monitorze – w naszym wypadku `"1"`, a następnie `"2"` będą pojawiać się kolejno co sekundę. Po uruchomieniu drugiego programu, korzystającego ze standardowej biblioteki wejścia/wyjścia, po chwili czekania na ekranie pojawią się kolejno cyfry `"2"`, a następnie dziesięć `"1"` na raz. Strumień wynikowy przy korzystaniu z funkcji `printf` podlega buforowaniu, natomiast strumień błędów wyświetlany jest na ekranie natychmiast, bez opóźnienia. Zastosowanie zakomentarzowanej w programie funkcji `fflush` spowoduje tzw. spłu-

kanie danych, czyli wyświetlenie kolejnych "1", bez wypełniania bufora, co w efekcie przyniesie podobny efekt, jak w sytuacji stosowania funkcji niskopoziomowych.

Do programowego kopiowania plików można używać funkcji i niskopoziomowych, i wysokopoziomowych. Ich efektywność może być różna i zależy od przyjętych parametrów. Celem zadania jest porównanie czasów kopiowania dużego pliku z zastosowaniem różnych funkcji oraz różnej wielkości buforów. Do badania czasu trwania procesu wykorzystamy polecenie systemowe `time`, które wyświetla czas działania procesu będącego parametrem, z wyszczególnieniem czasu w trybie systemowym, w trybie użytkownika oraz czasu rzeczywistego.

Zadanie

Porównaj czasy kopiowania pliku o rozmiarze 10MB, używając funkcji niskopoziomowych oraz funkcji ze standardowej biblioteki we/wy. W każdym wypadku kopiowanie powinno się odbywać po znaku oraz za pomocą buforów o rozmiarach 1024B, 2048B oraz 4096B.

Czasy kopiowania (tryb systemowy) zbierz w tabelce i przedstaw wnioski. Programy powinny mieć jak najprostszą formę, tak aby czas ich wykonania odpowiadał czasowi kopiowania. Przykładowe programy przedstawiono poniżej.

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
char c;
int we, wy;
we=open("we", O_RDONLY);
wy=open("wy", O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
while(read(we,&c,1)==1)
 write(wy,&c,1);
}
```

Kod 7.4. Program kopiujący znaki wykorzystujący funkcje niskopoziomowe

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
char blok[1024];
int we, wy;
int liczyt;
we=open("we", O_RDONLY);
wy=open("wy", O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
while((liczyt=read(we,blok,sizeof(blok)))>0)
write(wy,blok,liczyt);
}
```

Kod 7.5. Program kopiujący bloki wykorzystujący funkcje niskopoziomowe

```
#include <stdio.h>
int main(){
FILE *we,*wy;
int c;
we=fopen("we","r");
wy=fopen("wy","w");
if((we!=NULL)&&(wy!=NULL))
   while((c=fgetc(we))!=EOF)
      fputc(c,wy);
else
printf("blad otwarcia\n");
}
```

Kod 7.6. Program kopiujący znaki wykorzystujący funkcje wysokopoziomowe

```
#include <stdio.h>
int main(){
char blok[1024];
FILE *we,*wy;
int c;
we=fopen("we","r");
wy=fopen("wy","w");
if((we!=NULL)&&(wy!=NULL))
while(fgets(blok,1024,we))
 fputs(blok,wy);
else
printf("blad otwarcia\n");
}
```

Kod 7.7. Program kopiujący bloki wykorzystujący funkcje wysokopoziomowe

7.1.3. Operacje na katalogach

Grupa funkcji systemowych dedykowanych do programowego przetwarzania katalogów odwołuje się do strumienia katalogowego, czyli wskaźnika do struktury DIR. Wpisy katalogowe używają struktury dirent. Program przedstawiony poniżej jako rys. 7.9 wykorzystuje funkcję `opendir` otwierającą katalog, funkcję `readdir` odczytującą wpis z katalogu oraz funkcję `closedir` zamykającą katalog.

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main(int argc, char * argv[])
{
DIR *dirp;
struct stat status;
struct dirent *direntp;
dirp=opendir(argv[1]);
while((direntp=readdir(dirp))!=NULL)
  {
  printf("\t%s\n",direntp->d_name);
  lstat(direntp->d_name,&status);
  printf("czas dost=%d\n
rozmiar=%d\n",status.st_atime,status.st_size);
  if(S_ISDIR (status.st_mode))
 printf("katalog\n");
}
closedir(dirp);
}
```

Kod 7.8. Program ilustrujący operacje na katalogach

Program otwiera katalog, którego nazwa jest parametrem programu, i wyświetla informacje o wszystkich znajdujących się w nim plikach. Informacje o kolejnych plikach wprowadzane są do struktury typu dirent, a następnie funkcja `lstat` na podstawie nazwy pliku wypełnia strukturę stat atrybutami pliku (m.in. prawa dostępu, numer węzła, identyfikator użytkownika, czas ostatniego dostępu). Ze znacznikiem `st_mode` określającym prawa dostępu i typ pliku związane są również makra sprawdzające typ pliku. W programie użyto makra `S_ISDIR` sprawdzającego, czy dany plik jest katalogiem.

7.2. Funkcje systemowe związane ze środowiskiem i czasem

Pisząc programy w języku C, można korzystać z wielu funkcji systemowych. Za pomocą funkcji getenv można odczytywać wartość zmiennej środowiskowej, a funkcja putenv pozwala na zdefiniowanie nowej zmiennej środowiskowej procesu:

```
#include <stdlib.h>
char *getenv(const char *nazwa);
int putenv(const char *ciag);
```

Oto przykład programu, który wywołany z jednym parametrem odczytuje wartość zmiennej środowiskowej wskazanej tym parametrem, natomiast wywołany z dwoma parametrami definiuje nową zmienną i nadaje jej wartość według drugiego parametru:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int liczarg, char *tablarg[])
{
char *zmienna, *wart;
zmienna=tablarg[1];
wart=getenv(zmienna);
if(wart)
  printf("zmienna %s ma wartosc %s\n",zmienna,wart);
else
  printf("zmienna %s nie ma wartosci\n",zmienna);
char *string;
wart=tablarg[2];
string=malloc(strlen(zmienna)+strlen(wart)+2);
if(!string)
     {fprintf(stderr,"brak pamieci\n"); exit(1);}
strcpy(string, zmienna);
strcat(string,"=");
strcat(string,wart);
if(putenv(string)!=0)
   {
     fprintf(stderr,"blad\n");
     free(string);
     exit(1);
    }
```

```
wart=getenv(zmienna);
if(wart)
   printf("nowa wartosc %s jest %s\n", zmienna,wart);
else
   printf("nowa wart %s jest null\n", zmienna);
 }
exit(0);
}
```

Kod 7.9. Odczytywanie i modyfikacja zmiennej środowiskowej powłoki

```
$ ./srod KO hkflhk
zmienna KO nie ma wartosci
nowa wart KO jest hkflhk

$ KO=2 ./srod KO hkflhk
zmienna KO ma wartosc 2
nowa wart KO jest hkflh
```

Przykład 7.2. Efekt uruchomienia kodu 7.9

Funkcja `time` zwraca aktualny czas w postaci liczby sekund, jakie upłynęły od 1.01.1970 roku. Wynik trafia również w miejsce wskazane przez argument:

```
#include <time.h>
time_t time(time_t *tloc);
```

Oto przykłady wywołania funkcji `time`.

```
#include<time.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
time_t czas;
time_t *ti;
printf("czas=%ld\n",time(ti));
sleep(5);
czas=time(ti);
printf("czas=%ld\n",*ti);
```

```
sleep(2);
czas=time(ti);
printf("czas=%ld\n",czas);
exit(0);
}
```

Kod 7.10. Przykłady wywołania funkcji `time`

Funkcje `gmtime` i `localtime` zamieniają czas niskopoziomowy (podany za pomocą typu `time_t`) na strukturę `tm`, która zawiera następujące pola:

`int tm_sec` – sekundy <0; 60>,
`int tm_min` – minuty <0; 60>,
`int tm_hour` – godziny <0; 23>,
`int tm_mday` – dzień miesiąca <1; 31>,
`int tm_mon` – miesiąc <0; 11>; 0 – oznacza styczeń,
`int tm_year` – rok-1900; rok 2009 to 109,
`int tm_wday` – dzień tygodnia <0; 6>; 0 – to niedziela,
`int tm_yday` – dzień w roku <0; 365>,
`int tm isdst` – strefa czasowa.

```
#include <time.h>
struct tm *gmtime(const time_t *czas);
struct tm *localtime(const time_t *czas);
```

Funkcja `localtime` zwraca czas lokalny, uwzględniając strefę czasową.
Do przekształcenia czasu przedstawionego w formacie struktury `tm` na typ `time_t` służy funkcja `mktime`:

```
time_t mktime(struct tm *wskczas);
```

Istnieją funkcje `asctime` i `ctime` przekształcające czas na czas w ściśle określonym formacie.

```
#include <time.h>
char *asctime(const struct tm *wskczas);
char *ctime(const time_t *wartczas);
```

Jeśli czas jest typu `time_t`, wywołanie funkcji `time(czas)` jest równoważne wywołaniu `asctime(localtime(czas))`.

Fragment kodu:

```
czas=time(ti);
printf("ctime: %s\n",ctime(ti));
```

powoduje wyświetlenie informacji:

```
ctime: Fri Jun 5 23:50:31 2009
```

Do wyspecyfikowanego przez użytkownika formatowania czasu służą funkcje `strftime` i `strptime`. Funkcja `strftime` przekształca czas wskazany przez `wskczas` podany w formacie struktury `tm` według formatu wskazanego przez `format`, a wynik zapisuje w stringu `s`.

```
#include <time.h>
size_t strftime(char *s, size_t maxrozmiar, const char
*format, struct tm *wskczas);
```

Oto wybrane specyfikatory formatu:

`%a, %A` – skrócona, pełna nazwa dnia tygodnia,
`%b, %B` – skrócona, pełna nazwa miesiąca,
`%c` – data i godzina,
`%H` – godzina,
`%Y` – rok-1900,
`%p` – a.m. lub p.m.

Funkcja `strptime` wczytuje datę w postaci ciągu znaków (`bufor`) i wypełnia strukturę `tm` (`wskczas`) według zadanego formatu (`format`).

```
#include <time.h>
char *strptime(const char *bufor, const char
*format, struct tm *wskczas);
```

Poniżej przedstawiono program testujący funkcje związane z czasem. Program sprawdza, jaką datą jest czas = 0, zamienia datę: `1970-01-01:01:00:00` z powrotem na typ `time_t`, a następnie podaje dzień tygodnia odpowiadający wprowadzonej w zadanym formacie dacie.

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
int main()
{
struct tm *wczas;
time_t czas;
char bufor[1024];
czas=0;
wczas=gmtime(&czas);
printf("czas=%ld\n",czas);
strftime(bufor, 1024, "%Y-%m-%d:%T", wczas);
printf("czas 0  to: %s\n", bufor );
char znak;
strcpy(bufor,"1970-01-01:01:00:00");
printf("bufor=%s\n",bufor);
znak = strptime(bufor, "%Y-%m-%d:%T", wczas);
czas = mktime(wczas);
printf("zamiana na time_t   --- czas= %d\n", czas);
printf("Podaj date <YYYY-MM-DD>\n");
fgets(bufor, 12,stdin);
strptime(bufor, "%Y-%m-%d", wczas);
strftime(bufor, 1024, "%A", wczas);
printf("dzien tygodnia= %s\n", bufor);
}
```

Kod 7.11. Testowanie funkcji związanych z czasem

Oto efekt uruchomienia programu:

```
czas=0
czas 0 to 1970-01-01:00:00:00
bufor=1970-01-01:01:00:00
zamiana na time_t   --- czas= 0
Podaj date <YYYY-MM-DD>
1989-06-05
dzien tygodnia= Monday
```

Przykład 7.3. Przykład uruchomienia kodu 7.11

8. Proste sposoby komunikacji procesów[8]

Procesy mogą wykorzystywać różne sposoby komunikacji, takie jak:
- Sygnały (*signals*).
- Pamięć odwzorowywana – podobna do współdzielonej, związana jest z plikiem w systemie plików.
- Potok (*pipes*) – umożliwia sekwencyjną komunikację między powiązanymi procesami.
- Kolejka FIFO (*named pipes*) – podobna do potoku, umożliwia komunikację między niepowiązanymi procesami.
- Mechanizmy IPC.

Wymienione rodzaje komunikacji różnią się między sobą. Jedne ograniczają komunikację do procesów powiązanych, inne do procesów współdzielących ten sam system plików. Niektóre umożliwiają procesowi tylko zapis lub odczyt informacji, różnią się również liczbą procesów biorących udział w komunikacji.

8.1. Sygnały

Sygnały są metodą zawiadamiania procesu o zajściu jakiegoś zdarzenia. Pojawienie się sygnałów powoduje przerwanie pracy procesu i wymusza natychmiastową ich obsługę. W pewnym sensie przypominają one przerwania. Różnica polega na sposobie generowania. Przerwania tworzone są przez sprzęt, natomiast sygnały mogą być wysłane:
- z procesu do procesu,
- z systemu operacyjnego do procesu.

Z tego względu nazywamy je czasami przerwaniami programowalnymi.

Każdy sygnał ma swój unikalny numer i pojawia się w określonej sytuacji. Ponieważ różne wersje systemu Linux nie były konsekwentne w tej numeracji, wprowadzono również nazwy symboliczne sygnałów. W Linuksie sygnały są zdefiniowane w pliku `/usr/include/bits/signum.h`.

Sygnały są z natury asynchroniczne – gdy proces otrzyma sygnał, natychmiast go przetwarza. Jedyną metodą obsługi sygnałów jest wcześniejsze zadeklarowanie, co należy z nimi zrobić. Istnieją trzy możliwości:
- Sygnał możemy zignorować – po przyjściu sygnału do procesu praktycznie nic się nie stanie. Istnieją jednak dwa sygnały, z którymi nie możemy postąpić

[8] Na podstawie prac [4, 6–8].

w ten sposób: SIGKILL oraz SIGSTOP. Gwarantuje nam to możliwość zakończenia dowolnego programu (który np. się zawiesił). Nie należy również ignorować sygnałów, które zostały wywołane w wyniku błędu sprzętowego (np. SIGSEGV).

- Sygnał możemy przychwycić przez podanie funkcji, która zastanie wywołana po pojawieniu się sygnału. W jej wnętrzu możemy umieścić właściwie dowolny kod.
- Ostatnim sposobem radzenia sobie z sygnałami jest po prostu nicrobienie niczego – pozwolenie na wywołanie domyślnej funkcji obsługi sygnału.

Istnieje pięć sposobów generowania sygnałów:

1. Skróty klawiszowe (najczęściej stosowane) – pewne kombinacje klawiszy powodują generowanie sygnałów.

   CTRL+C – wciśnięcie tych klawiszy powoduje wysłanie przez system operacyjny sygnału SIGINT do bieżącego procesu. Domyślnie sygnał ten powoduje natychmiastowe zakończenie procesu.

   CTRL+\ – powoduje zakończenie bieżącego procesu i generowanie obrazu pamięci tego procesu; generowany jest sygnał SIGQUIT.

   CTRL+Z – powoduje zawieszenie procesu przez wysłanie sygnału SIGTSTP.

2. Funkcja systemowa `kill` zezwala, aby proces wysłał sygnał do drugiego procesu albo do samego siebie. Składnia wygląda następująco:

```
include <sys/types.h>
#include <signal.h>

int kill(int idproc, int sig);
int raise(int sig);
```

   Aby można było jej użyć, proces wysyłający musi mieć taki sam identyfikator użytkownika, jak proces, który odbiera sygnał, albo wysyłający musi być nadzorcą systemu.

3. Polecenie `kill` służy również do wysyłania sygnałów. To polecenie powoduje wykonanie programu, który pobiera argumenty z wiersza polecenia i wywołuje funkcję systemową `kill`. Format wywołania to:

```
kill -<signal> <PID>
```

4. Pewne wykrywane sprzętowo sytuacje są przyczyną generowania sygnałów. Przykładowo, odwołanie do niewłaściwego adresu pamięci generuje sygnał SIGSEGV. Błąd przy wykonywaniu operacji na liczbach zmiennoprzecinkowych jest wskazywany sygnałem SIGFPE.

5. Pewne sytuacje wykrywane przez oprogramowanie systemowe, o których jest powiadamiane jądro, powodują generowanie sygnałów. Przykładem jest sygnał SIGURG, który pojawia się, gdy na połączeniu sieciowym znajdą się dane wysokopriorytetowe, lub sygnał SIGALRM, generowany przy przekroczeniu terminu licznika zegarowego ustawionego w procesie.

Lista sygnałów jest długa i zależy od wersji systemu – można ją wyświetlić poleceniem:

```
kill -l
```

Najbardziej znane sygnały to:

- **SIGTERM** – jest sygnałem zakończenia wysyłanym domyślnie przez polecenie `kill`.
- **SIGKILL** – jest jednym z dwóch sygnałów, które nie mogą być ani przechwytywane, ani ignorowane. Daje administratorowi systemu niezawodną metodę usunięcia dowolnego procesu.
- **SIGUSR1** – jest sygnałem zdefiniowanym przez użytkownika, stosowanym w programach aplikacyjnych.
- **SIGUSR2** – jest sygnałem zdefiniowanym przez użytkownika, stosowanym w programach aplikacyjnych.
- **SIGHUP** – jest zwykle wykorzystywany do zbudzenia oczekującego programu lub spowodowania ponownego odczytania plików konfiguracyjnych.
- **SIGSEGV** – wskazuje, że program wykonał odwołanie do niewłaściwego adresu pamięci.
- **SIGBUS** – wskazuje błąd sprzętowy zdefiniowany w konkretnej implementacji.
- **SIGFPE** – oznacza pojawienie się wyjątku matematycznego, np. dzielenie przez zero, przepełnienie liczby zmiennoprzecinkowej itp.

Do ustawiania akcji sygnału możemy użyć funkcji `sigation()`. Możemy za jej pomocą sprawdzać lub modyfikować akcję związaną z konkretnym sygnałem. Jej pierwszym parametrem jest numer sygnału. Następne dwa to wskaźniki do struktur `sigaction`, w których najważniejszym polem jest `sa_handler`. Może ono przyjmować jedną z trzech wartości:

- SIG_IGN – powoduje ignorowanie sygnału.
- SIG_DFL – powoduje wywołanie akcji domyślnej.
- Wskaźnik funkcji obsługującej sygnał.

Przykład użycia funkcji `sigaction()` przedstawia poniższy kod:

```
#include <signal.h>
struct sigaction
{
     void (*sa_handler)(int);
     sigset_t sa_mask;
     int sa_flags;
};
int sigaction(int signo, const struct sigaction *act,
struct sigaction *act);
```

Kod 8.1. Przykład użycia funkcji `sigaction()`

Argument `signo` jest numerem sygnału, dla którego chcemy sprawdzić lub zmodyfikować dyspozycje. Wskaźnik `act` zawiera nowe dyspozycje:

`sa_handler` – wskaźnik do funkcji obsługi sygnału,
`sa_mask` – zbiór dodatkowych sygnałów do zablokowania,
`sa_flags` – zbiór dodatkowych opcji.

Procedura obsługi sygnału może zostać przerwana przez dostarczenie innego sygnału – trzeba zatem zwracać uwagę na to, co robi program w procedurze obsługi sygnału.

8.2. Odwzorowanie w pamięci

Odwzorowywanie w pamięci umożliwia komunikację różnych procesów za pomocą współdzielonego pliku. Tworzy ono połączenie pomiędzy plikiem a pamięcią procesu. Plik dzielony jest na fragmenty wielkości strony i kopiowany do stron pamięci wirtualnej, przez co staje się dostępny w przestrzeni adresowej procesu. Takie rozwiązanie umożliwia szybki dostęp do pliku. Aby odwzorować zwykły plik w pamięci procesu, korzystamy z `mmap()`:

```
void * mmap(void *start, size_t length, int prot, int
flags, int fd, off_t offset);
```

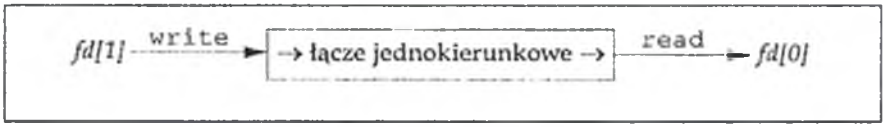
Pierwszym argumentem jest adres, pod którym umieszczony będzie odwzorowany plik (NULL pozwala wybrać dostępny adres początkowy). Drugim argumentem jest rozmiar odwzorowania w bajtach, trzeci określa ochronę adresów (są to połączone operacją alternatywy flagi PROT_READ, PROT_WRITE oraz PROT_EXEC). Czwarty argument to flagi zawierające dodatkowe opcje (są to flagi: MAP_FIXED, MAP_PRIVATE oraz MAP_SHARED), piąty jest przesunięciem względem początku pliku.

8.3. Potoki

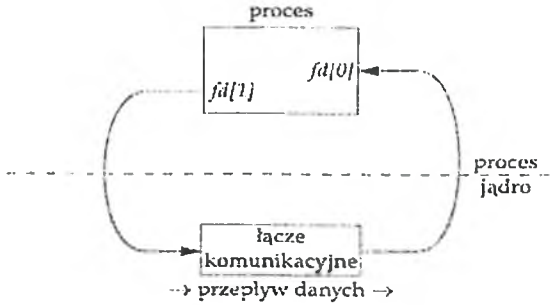
Potok jest jednokierunkowym urządzeniem komunikacyjnym. Dane zapisane na jednym końcu potoku są odczytywane na jego drugim końcu. Są to urządzenia szeregowe – dane odczytujemy w tej samej kolejności, w jakiej zostały zapisane. Są one pierwotną formą komunikacji międzyprocesowej w systemie Unix. W wielu sytuacjach są bardzo użyteczne, mają bowiem bardzo istotne ograniczenie w postaci braku nazwy, zatem mogą być używane tylko przez procesy spokrewnione (macierzysty, potomny). Pojemność potoku jest ograniczona. Jeśli proces piszący pisze szybciej niż proces czytający pobiera plik, proces piszący jest blokowany do czasu, aż dostępne będzie więcej miejsca. Jeśli w potoku nie będzie danych, a proces czytający będzie próbował je czytać, zostanie zablokowany do czasu ich nadejścia.

Aby utworzyć potok, korzystamy z polecenia `pipe`, do którego trzeba przekazać tablicę dwóch liczb całkowitych. W zerowym elemencie tablicy przechowywany jest deskryptor do odczytu, natomiast w pierwszym – deskryptor do zapisu.

```
int pipe_fd[2];
int read_fd;
int write_fd;

pipe (pipe_fd);
read_fd =pipe_fd[0];
write_id=pipe_fd[1];
```



Rys. 8.1. Łącze jednokierunkowe



Rys. 8.2. Komunikacja między procesem macierzystym i potomnym

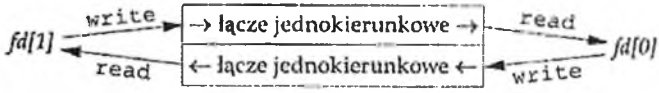
Poniżej zamieszczono przykładowy program ilustrujący komunikację między procesami spokrewnionymi:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    int fd[2];
    pid_t pid;
    int size = 256;
    char buf[size];
/* utworzenie łącza    */
    if (pipe(fd) < 0)
        exit(1);}
    /* utworzenie procesu potomnego */
    if ((pid = fork()) < 0) {
        exit(1);
    }
    else if (pid == 0) {
/* zamknięcie końca do odczytu procesu potomnego */
        close(fd[0]);
/* zapisanie informacji do łącza    */
        write(fd[1], "komunikat\n", 10);}
    else {
        close(fd[1]);
/* zamknięcie końca do zapisu procesu macierzystego
*/
        read(fd[0], buf, size);// odczytanie
informacji
```

```
        printf("%s", buf);
    }
    return 0;
}
```

Kod 8.2. Komunikacja między procesami

Łącza jednokierunkowe zapewniają komunikację w jednym kierunku. Kiedy potrzebujemy przepływu w obu kierunkach, musimy utworzyć dwa łącza i korzystać z jednego dla każdego kierunku (rys. 8.3).



Rys. 8.3. Łącze dwukierunkowe

Często chcemy utworzyć proces potomny i ustawić jeden z końców potoku jako standardowe wejście lub wyjście. Funkcje dup i dup2 służą do powielania istniejących deskryptorów pliku.

```
#include <unisted.h>
int dup(int deskryptor1);
int dup(int deskryptor1, deskryptor2);
```

- Funkcja dup tworzy (i zwraca) nowy deskryptor pliku, gwarantując, że będzie on miał najmniejszą wartość spośród wszystkich wolnych wartości numerów dla deskryptorów. Argumentem funkcji jest istniejący już deskryptor, który chcemy powielić.
- Funkcja dup2 tworzy nowy deskryptor, którego wartość jest równa deskryptorowi 2. Jeżeli deskryptor 2 jest już otwarty, zostanie zamknięty przed wykonaniem kopiowania. Gdy deskryptory 1 i 2 są sobie równe, wtedy zostaje zwrócona wartość deskryptora 2 bez uprzedniego zamknięcia (dlatego zawsze trzeba sprawdzić, jakie są deskryptory).

Najczęściej tworzymy łącze komunikacyjne powiązane z innym procesem, aby czytać dane wyjściowe albo przesyłać dane wejściowe. Ułatwiają to funkcje popen oraz pclose, które eliminują konieczność wywoływania funkcji pipe, fork, dup2 oraz exec.

```
#include <stdio.h>
FILE* popen(const char*, cmdstring, const char* type);
int pclose(FILE* fp);
```

Funkcja popen zwraca wskaźnik (uchwyt) pliku (FILE*, odpowiadający otwartemu plikowi), gdy wszystko jest w porządku, w razie błędu zwraca NULL. Wywołuje funkcję fork, a następnie exec, która wykonuje przez powłokę polecenie cmdstring. Argument type może przyjmować dwie wartości: "w" (uchwyt związany ze standardowym wejściem) oraz "r" (uchwyt związany ze standardowym wyjściem).

Funkcja pclose przekazuje stan zakończenia polecenia cmdstring, gdy wszystko jest w porządku, w razie błędu zwraca wartość –1. Zamyka standardowy strumień I/O.

Zamieszczony poniżej program tworzy dwa potoki do komunikacji z procesami potomnymi: fp_in to koniec potoku do odczytu danych wygenerowanych przez proces KTO; fp_out to koniec potoku do zapisu danych przez proces grep. Dane wynikowe procesu KTO trafiają (za pośrednictwem procesu macierzystego) do procesu grep i są wypisywane na standardowe wyjście.

```
#include <stdio.h>
int main(){
   FILE *fp_in, *fp_out;
   int size=256;
   char buf[size];

   fp_in=popen("KTO","r"); // zwraca koniec do odczytu
   fp_out=popen("grep systemy", "w"); // zwraca koniec
do zapisu
   while (fgets(buf,size,fo_in);
          fputs(buf,fp_out);
   pclose(fp_in);
   pclose(fp_out);
   return0;}
```

Kod 8.3. Komunikacja między procesami potomnymi

8.4. Kolejki FIFO

Nazwa pochodzi od angielskiego określenia *first in, first out*, czyli „pierwszy na wejściu, pierwszy na wyjściu". Kolejki FIFO są podobne do łączy i zapewniają jednokierunkowy przepływ danych. Łączą w sobie cechy pliku i łącza. Podobnie jak plik, kolejka FIFO ma swoją nazwę, co umożliwia komunikację procesom ze sobą niepowiązanym. Kolejka FIFO jest tworzona za pomocą funkcji mkfifo.

Potoki nazwane mają zastosowanie w przekazywaniu danych przez polecenia powłoki do innych poleceń bez tworzenia plików na dysku (zwłaszcza dla poleceń nieliniowych) oraz w aplikacjach do przesyłania danych klient–serwer.

```
#include <sys/types.h
#include <sys/types.h
int mkfifo(const char* parthname, mode_t mode);
int mknod(const char* parthname,mode_t mode,
dev_t dev);
```

Można użyć jednej z powyższych funkcji:

- Funkcja `mkfifo` – za pierwszy argument przyjmuje nazwę kolejki (ewentualnie ścieżkę, gdzie zostanie utworzona), za drugi argument przyjmuje uprawnienia właściciela potoku, grupy i innych użytkowników (analogicznie jak dla funkcji `open`, np. `0600`). Utworzony potok może służyć do zapisu i odczytu (przez odpowiedni proces), dlatego trzeba to uwzględnić przy nadawaniu uprawnień.
- Funkcja `mknod` – ma większą funkcjonalność niż funkcja `mkfifo` i służy do tworzenia specjalnych plików. Tworząc potok, trzeba ustawić `mode` jako `S_ISFIFO` (ewentualnie z podaniem maski, np. `S_ISFIFO|0666`) oraz `0` jako `dev`.

Kolejki FIFO używa się tak jak zwykłego pliku. Aby możliwa była komunikacja za pomocą kolejki, jeden program musi otworzyć ją do zapisu, a inny do odczytu. Można korzystać z niskopoziomowych funkcji I/O (`open`, `write`, `read`, `close`) oraz z funkcji I/O biblioteki C (`fopen`, `fprintf`, `fscanf`, `fclose`).

Przy zapisie do łącza lub kolejki FIFO bufora danych za pomocą niskopoziomowych funkcji I/O można posłużyć się następującym kodem:

```
int fd=open (fifo_path, O_WRONLY);
write (fd,data, data_lenght);
close(fd);
```

Aby odczytać tekst z kolejki FIFO za pomocą funkcji I/O biblioteki C, można użyć następującego kodu:

```
FILE* fifo=fopen (fifo_path, "r");
fscanf(fifo, "%s", bufor);
close(fifo);
```

Wiele procesów może pisać do kolejki FIFO lub z niej czytać. Bajty od każdego procesu są niepodzielnie zapisywane do maksymalnej wielkości PIPE_BUF = 4KB w systemie Linux. Typowe przykłady otwierania kolejki to:

`open(fifoName, O_RDONLY|O_NONBLOCK)` – funkcja powraca natychmiast

`open(fifoName, O_RDONLY)` – funkcja zablokuje się, dopóki inny proces nie otworzy kolejki do zapisu

`open(fifoName, O_WRONLY)` – funkcja zablokuje się, dopóki inny proces nie otworzy tej kolejki do odczytu

Istotna zaleta kolejek FIFO objawia się, gdy serwer jest procesem o długim czasie działania, niespokrewnionym z klientem. Klient tworzy kolejkę FIFO o powszechnie znanej nazwie ścieżki, otwiera ją do odczytu, a klient, który później startuje, otwiera ją do zapisu i wysyła przez nią do serwera zadania. Oto przykład wykorzystania kolejki FIFO do realizacji zadania producent–konsument:

Program producenta:

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include<limits.h>
#include<sys/types.h>
#include<sys/stat.h>
#define NAZWA_FIFO "/tmp/moje_fifo"
#define ROZMIAR 10
FILE *fd;
int main(int argc,char *argv[]){
 int potok,res,i=0,n,l,w;
 char bufor[ROZMIAR+1];
 if(access(NAZWA_FIFO,F_OK)==-1)
  {
   res=mkfifo(NAZWA_FIFO,0777);
   if(res!=0)
   {
    fprintf(stderr,"Blad mkfifo-%s\n",NAZWA_FIFO);
    exit(2);
```

```
    } }
  potok=open(NAZWA_FIFO,O_WRONLY);
  fd=fopen("zrodlo","r");
  if(potok!=-1)
   {
    while(!(w=feof(fd)))
     {
      sleep(1);
      fscanf(fd,"%d",&l);
      sprintf(bufor,"%d",1);
      printf("Liczba wysłana-%d\n",1);
      res=write(potok,bufor,ROZMIAR);
      if(res==-1)
       {
        fprintf(stderr,"Blad zapisu do potoku\n");
        exit(3);   } }
    (void)close(potok);
   }
 else
    exit(4);
printf("Koniec  porducenta-%d\n\t",getpid());
exit(0); }
```

Kod 8.4. Przykład wykorzystania kolejki FIFO – kod producenta

Program konsumenta:

```
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include<limits.h>
#include<sys/types.h>
#include <sys/stat.h>
#define NAZWA_FIFO "/tmp/moje_fifo"
#define ROZMIAR 10

int main()
 {
  int potok,res=1;
  char bufor[ROZMIAR+1];
  memset(bufor,'\0',sizeof(bufor));
  potok=open(NAZWA_FIFO,O_RDONLY);
```

```
  if(potok!=-1)
   {
    while(res>0)
     {
      res=read(potok,bufor,ROZMIAR);
      if(res!=0)
      printf("Liczba odczytana-%s\n",bufor);
     }
   (void)close(potok);
   }
   else
   {
   exit(EXIT_FAILURE);
   }

   printf("Koniec konsumenta %d\n\t",getpid());
   exit(EXIT_SUCCESS);
}
```

Kod 8.5. Przykład wykorzystania kolejki FIFO – kod konsumenta

9. Mechanizmy IPC[9]

Podobnie jak łącza, mechanizmy IPC (*Inter Process Communication*) to grupa mechanizmów komunikacji i synchronizacji procesów działających w ramach tego samego systemu operacyjnego. Mechanizmy IPC obejmują:

- Kolejki komunikatów – umożliwiają przekazywanie określonych porcji danych.
- Pamięć współdzieloną – umożliwia współdzielenie kilku procesom tego samego fragmentu wirtualnej przestrzeni adresowej.
- Semafory – umożliwiają synchronizację procesów w dostępie do współdzielonych zasobów, np. do pamięci współdzielonej.

Wszystkie urządzenia IPC mają podobny interfejs. Najważniejszą wspólną cechą jest klucz urządzenia IPC. Klucze są liczbami używanymi do identyfikacji obiektów IPC, co umożliwia wspólne użycie zasobów IPC przez kilka niespokrewnionych procesów. Klucze różnych obiektów nie mogą się powtarzać. Do znajdowania unikatowego klucza zależnego od ścieżki pliku służy funkcja `ftok()`.

```
key_t ftok(const char *path,int id)
```

Zwraca ona numer klucza na podstawie `path` pliku. Parametr `id` daje dodatkowy poziom niepowtarzalności – ta sama `path` dla różnych `id` daje różne klucze. Do tworzenia obiektów IPC i manipulowania ich danymi służy zbiór funkcji przedstawiony w tabeli 9.1.

Tabela 9.1

Zbiór funkcji do tworzenia obiektów IPC i manipulacji ich danymi

Działanie funkcji	Kolejka komunikatów	Pamięć współdzielona	Semafory
Rezerwowanie obiektu IPC oraz uzyskiwanie do niego dostępu	`msgget`	`shmget`	`semget`
Sterowanie obiektem IPC, uzyskiwanie informacji o stanie modyfikowanych obiektów IPC, usuwanie obiektów IPC	`msgctl`	`shmctl`	`semctl`
Operacje na obiektach IPC: wysyłanie i odbieranie komunikatów, operacje na semaforach, rezerwowanie i zwalnianie segmentów pamięci wspólnej	`msgsnd`, `msgrcv`	`shmat`, `shmdt`	`semop`

[9] Na podstawie prac [4, 6–8].

Wywołania systemowe get (msgget, shmget, semget) są stosowane do tworzenia nowych obiektów IPC lub do uzyskania dostępu do obiektów, które już istnieją. Drugim wywołaniem systemowym wspólnym dla mechanizmów IPC jest ctl (msgctl, shmctl, semctl), używane w celu przeprowadzenia operacji kontrolnych na obiektach IPC. Funkcje get zwracają wartości całkowitoliczbowe, nazywane identyfikatorami IPC, które identyfikują obiekty IPC. Od strony systemu operacyjnego identyfikator IPC jest indeksem w systemowej tablicy zawierającej struktury z danymi dotyczącymi uprawnień do obiektów IPC. Struktura IPC jest zdefiniowana w pliku <sys/ipc.h>.

Każda z funkcji get wymaga określenia argumentu typu key_t, nazywanego kluczem, który umożliwia generowanie identyfikatorów IPC. Procesy poprzez podanie tej samej wartości klucza uzyskują dostęp do konkretnego mechanizmu IPC. Wartość klucza można określić, podając samodzielnie konkretną wartość. Polu temu można także przypisać stałą IPC_PRIVATE, która spowoduje utworzenie obiektu IPC o niepowtarzalnej wartości identyfikatora. Łącze identyfikowane przez klucz wygenerowany na podstawie stałej IPC_PRIVATE pozwala na komunikację jedynie pomiędzy procesami spokrewnionymi, ponieważ procesy potomne dziedziczą wartość klucza od swoich przodków.

Drugim parametrem wspólnym dla wszystkich wywołań z rodziny get jest znacznik komunikatu, który określa prawa dostępu do tworzonego obiektu IPC. Prawa te mogą być połączone operacją logiczną OR z flagami IPC_CREAT lub IPC_EXCL. Flaga IPC_CREAT nakazuje funkcjom get utworzenie nowego obiektu IPC, jeśli on jeszcze nie istnieje. Jeśli natomiast obiekt IPC już istnieje i jego klucz nie został wygenerowany z użyciem stałej IPC_PRIVATE, to funkcje get zwrócą identyfikator tego obiektu. Natomiast użycie flag IPC_CREAT | IPC_EXCL spowoduje, że gdy obiekt IPC dla danej wartości klucza już istnieje, wywołanie funkcji get zakończy się błędem. Dzięki połączeniu tych dwóch flag użytkownik ma gwarancję, że jest on twórcą danego obiektu IPC.

Wywołania funkcji ctl mają dwa argumenty wspólne: identyfikator obiektu IPC otrzymany w wyniku wywołania odpowiedniej funkcji get oraz następujące stałe: IPC_STAT, IPC_SET i IPC_RMID, zdefiniowane w pliku <sys/ipc.h>:

- IPC_STAT – zwraca informację o stanie danego obiektu IPC,
- IPC_SET – zmienia właściciela, grupę i tryb obiektu IPC,
- IPC_RMID – usuwa obiekt IPC z systemu.

9.1. Obsługa mechanizmów IPC z konsoli systemu

Na poziomie systemu operacyjnego dane znajdujące się w obiekcie IPC pobiera się za pomocą polecenia `ipcs`. Informacje na temat konkretnych obiektów: kolejek komunikatów, pamięci współdzielonej i semaforów, otrzymamy, stosując odpowiednio przełączniki: `-q`, `-m`, `-s`.
Informacja na temat kolejki komunikatów o identyfikatorze `msgid`:

```
ipcs -q msgid
```

Informacja na temat segmentu pamięci współdzielonej o identyfikatorze `shmid`:

```
ipcs -m shmid
```

Informacja na temat zestawu semaforów o identyfikatorze `semid`:

```
ipcs -s semid
```

Dodatkowo przełącznik `-b` pozwala uzyskać informację na temat maksymalnego rozmiaru obiektów IPC, czyli liczby bajtów w kolejkach, rozmiarów segmentów pamięci współdzielonej i liczby semaforów w zestawach.

Usunięcie obiektu IPC można natomiast przeprowadzić, wykonując polecenie systemowe `ipcrm`.

Usunięcie kolejki komunikatów o identyfikatorze `msgid`:

```
ipcrm -q msgid
```

Usunięcie segmentu pamięci współdzielonej o identyfikatorze `shmid`:

```
ipcrm -m shmid
```

Usunięcie zestawu semaforów o identyfikatorze `semid`:

```
ipcrm -s semid
```

9.2. Kolejki komunikatów

Kolejki komunikatów umożliwiają przesyłanie pakietów danych, nazywanych komunikatami, pomiędzy różnymi procesami. Sam komunikat jest zbudowany jako struktura `msgbuf`, której definicja znajduje się w pliku <sys/msg.h>.

```
struct msgbuf{
long mtype; //typ komunikatu (>0)
char mtext[1]; //treść komunikatu
}
```

Każdy komunikat ma określony typ i długość. Typ komunikatu pozwalający określić rodzaj komunikatu nadaje proces inicjujący komunikat. Komunikaty są umieszczane w kolejce w kolejności ich wysyłania. Nadawca może wysyłać komunikaty nawet wówczas, gdy żaden z potencjalnych odbiorców nie jest gotowy do ich odbioru. Komunikaty są w takich wypadkach buforowane w kolejce i oczekują na odebranie. Przy odbiorze komunikatu odbiorca może oczekiwać na pierwszy przybyły komunikat lub na pierwszy komunikat określonego typu. Komunikaty w kolejce są przechowywane nawet po zakończeniu procesu nadawcy tak długo, aż nie zostaną odebrane lub kolejka nie zostanie zlikwidowana. Podczas tworzenia kolejki komunikatów tworzona jest systemowa struktura danych o nazwie `msgid_ds`. Definicję tej obsługiwanej przez system struktury można znaleźć w pliku nagłówkowym <sys/msg.h>. Funkcje umożliwiające komunikację za pomocą kolejek komunikatów zdefiniowane są w plikach nagłówkowych: <sys/types.h>, <sys/ipc.h>, <sys/msg.h>.

9.2.1. Utworzenie kolejki komunikatów

Do utworzenia kolejki komunikatów służy funkcja `msgget` o następującej postaci:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
```

Funkcja zwraca identyfikator kolejki komunikatów, natomiast w razie błędnego zakończenia funkcji możliwe kody błędów (`errno`) to:

- EACCES – kolejka o danym kluczu istnieje, ale proces wywołujący funkcję nie ma wystarczających praw dostępu do tej kolejki.
- EEXIST – kolejka o danym klucz istnieje, a msgflg zawiera jednocześnie oba znaczniki: IPC_CREAT i IPC_EXCL.
- EIDRM – kolejka została przeznaczona do usunięcia.
- ENOENT – kolejka o danym kluczu nie istnieje oraz msgflg nie zawiera flagi IPC_CREAT.
- ENOMEM – kolejka komunikatów powinna zostać utworzona, ale w systemie brak jest pamięci na utworzenie nowej struktury danych.
- ENOSPC – kolejka komunikatów powinna zostać utworzona, ale przekroczone zostałoby systemowe ograniczenie (MSGMNI) na liczbę istniejących kolejek komunikatów.

Funkcja msgget nie wykona się prawidłowo, jeśli kolejka komunikatów o danym kluczu już istnieje, a msgflg będzie zawierać flagi IPC_CREAT oraz IPC_EXCL.

9.2.2. Dodawanie komunikatu do kolejki

Funkcja msgsnd wysyła komunikat do kolejki o identyfikatorze msgid:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msgid, struct msgbuf *msgp, int msgs,
int msgflg)
```

Funkcja ma następujące argumenty:

- msgid – identyfikator kolejki komunikatów,
- msgp – wskaźnik na komunikat do wysłania,
- msgs – rozmiar właściwej treści komunikatu w bajtach,
- msgflg – flagi specyfikujące zachowanie się funkcji w warunkach nietypowych. Wartość ta może być ustawiona na 0 lub IPC_NOWAIT.

Funkcja poprawnie wykonana zwraca zera. W razie błędnego zakończenia funkcji możliwe są następujące kody błędów:

- EACCES – kolejka o podanym kluczu istnieje, ale proces wywołujący funkcję nie ma wystarczających praw dostępu do kolejki.
- EAGAIN – kolejka jest pełna, a flaga IPC_NOWAIT była ustawiona.
- EFAULT – niepoprawny wskaźnik msgq.

- EIDRM – kolejka została przeznaczona do usunięcia.
- EINTR – otrzymano sygnał podczas oczekiwania na operację zapisu.
- EINVAL – niepoprawny identyfikator kolejki lub ujemny typ wiadomości, lub nieprawidłowy rozmiar wiadomości.

Jeśli w momencie wysyłania komunikatu system osiągnął już limit długości kolejki, to w zależności od wartości flagi msgflg funkcja nie wyśle komunikatu i powróci z funkcji msgsnd (dla msgflg =IPC_NOWAIT) lub zablokuje proces wywołujący aż do chwili, gdy w kolejce będzie wystarczająco dużo wolnego miejsca, by żądany komunikat mógł być wysłany (przy msgflg=0).

Treść wysyłanego komunikatu w rzeczywistości może mieć dowolną strukturę. Przykładowa struktura msgbuf:

```
struct msgbuf {
long mtype; // typ komunikatu (int >0)
char mtext[1]; // tresc komunukatu
}
```

Pole mtype określa typ komunikatu, dzięki czemu możliwe jest przy odbiorze wybieranie z kolejki komunikatów określonego rodzaju. Typ komunikatu musi być wartością większą od 0.

9.2.3. Pobranie komunikatu z kolejki

Do pobrania komunikatu z kolejki służy funkcja msgrcv o następującej postaci:

```
int msgrcv (int msgid, struct msgbuf *msgp, int msgs,
long msgtyp, int msgflg)
```

Funkcja ma następujące argumenty:

- msgid – identyfikator kolejki komunikatów,
- msgp – wskaźnik do obszaru pamięci, w którym ma zostać umieszczony pobrany komunikat,
- msgs – rozmiar właściwej treści komunikatu,
- msgtyp – typ komunikatu, który ma być odebrany z kolejki. Możliwe są następujące wartości zmiennej msgtyp:
  - msgtyp > 0 – pobierany jest pierwszy komunikat typu msgtyp,
  - msgtyp < 0 – pobierany jest pierwszy komunikat, którego wartość typu jest mniejsza lub równa wartości |msgtyp|,

- msgtyp = 0 – typ komunikatu nie jest brany po uwagę – funkcja pobiera pierwszy komunikat dowolnego typu,
- msgflg – flaga specyfikująca zachowanie się funkcji w warunkach nietypowych. Wartość ta może być ustawiona na 0, IPC_NOWAIT lub MSG_NOERROR.

Poprawnie wykonana funkcja zwraca ilość odebranych bajtów, w wypadku błędnego zakończenia zwracana jest wartość –1. Możliwe kody błędów są analogiczne jak dla funkcji msgsnd.

Odebranie komunikatu oznacza pobranie go z kolejki. Raz odebrany komunikat nie może zostać odebrany ponownie. Argument msgflg określa czynność, która jest wykonywana, gdy żądanego komunikatu nie ma w kolejce lub miejsce przygotowane do odebrania komunikatu jest niewystarczające.

Gdy wartością msgflg jest IPC_NOWAIT, funkcja przy żądaniu odbioru komunikatu, którego nie ma w kolejce, nie będzie blokowała wywołującego ją procesu, natomiast flaga MSG_NOERROR spowoduje odpowiednie obcinanie rozmiaru komunikatu za dużego, by go odebrać. W sytuacji, gdy flaga MSG_NOERROR nie jest ustawiona i otrzymany komunikat jest za długi, funkcja zakończy się błędem. Jeśli nie ma znaczenia fakt, czy komunikaty mają być obcinane, czy nie, flagę msgflg należy ustawić na 0.

9.2.4. Zarządzanie kolejką

Do zarządzania kolejką służy funkcja msgctl o postaci:

```
int msgctl(int msgid, int cmd, struct msgid_ds. *buf)
```

Funkcja ma następujące argumenty:

- msgid – identyfikator kolejki,
- cmd – stała specyfikująca rodzaj operacji,
  - cmd = IPC_STAT – pozwala uzyskać informację o stanie kolejki,
  - cmd = IPC_SET – pozwala zmienić związane z kolejką ograniczenia,
  - cmd = IPC_RMID – pozwala usunąć kolejkę z systemu,
- buf – wskaźnik na zmienną strukturalną, przez którą przekazywane są parametry operacji.

Poprawne wykonanie funkcji zwraca 0, w przypadku błędu –1. Możliwe kody błędów są następujące:

- EACCES – nie ma praw do odczytu oraz cmd jest ustawiona na IPC_STAT,
- EFAULT – adres wskazywany przez buf jest nieprawidłowy,
- EIDRM – kolejka została usunięta,
- EINVAL – msgqid nieprawidłowe lub msgsz mniejsze od 0,
- EPERM – komendy IPC_SET lub IPC_RMID zostały wydane, podczas gdy proces nie ma praw dostępu do zapisu.

Przykładowy program obsługujący kolejki komunikatów – proces klienta – wysyła do procesu serwera ciąg znaków. Serwer odbiera ten ciąg znaków i przetwarza go, zamieniając w nim wszystkie litery na duże, a następnie wysyła tak przetworzony ciąg znaków z powrotem do klienta. Klient odbiera przetworzony ciąg znaków i wypisuje go na ekranie.

Proces klienta:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
//maksymalny rozmiar wiadomosci
#define MAX 80
#define SERWER 1
//struktura komunikatu
struct komunikat {
        long mtype;
        char mtext[MAX];
};
int main(int argc, char *argv[])
{
    int nr_klienta,tem,i;
    char tmptxt[10],wiadomosc[80];
    key_t key;   //unikalny klucz kolejki komunikatow
    int IDkolejki;  //identyfikator kolejki
    struct komunikat kom;  //przesylany komunikat
    nr_klienta=getpid();
    //tworzenie unikalnego klucza urzadzenia IPC dla
kolejki komunikatow
    key = ftok(".", 123); //tworzymy kolejke komunikatow
    //tworzenie kolejki
    if( (IDkolejki = msgget(key, IPC_CREAT | 0660)) == -1)
    {
        perror("msgget() calling...");
        exit(1);
```

```
    }
while(1)
{
    //wysylanie wiadomosci
    kom.mtype = SERWER; //zapisujemy typ komunikatu -
klienci wszyscy wysylaja do serwera - typ 1
    sprintf(tmptxt,"%d~",getpid());
    strcpy(kom.mtext, tmptxt);
    printf("K[%d]: Podaj tekst do wyslania:\n",getpid());
    i=0;
    while(1)
    {
     wiadomosc[i]=getchar();
     if ((wiadomosc[i]=='\n') || (i>=80))
      {
 wiadomosc[i]='\0';
 break;
      }
     i++;
    }
    strcat(kom.mtext, wiadomosc);   //laczymy wprowadzona
przez uzytkownika wiadomosc do wyslania z zawartoscia
kom.text
    printf("K[%d]: Wysylanie... \"%s\" -> SERWER\n",
nr_klienta, &kom.mtext[strlen(kom.mtext)-
strlen(wiadomosc)]);
    msgsnd(IDkolejki, (struct msgbuf *)&kom,
strlen(kom.mtext)+1, 0); //wyslanie zawrtosci kom
    kom.mtype = getpid(); //odczytuje ze swojej kolejki o
typie rownym pidowi procesu
    msgrcv(IDkolejki, (struct msgbuf *)&kom, MAX,
kom.mtype, 0);
    printf("K[%d]: Odebrano: \"%s\" zaadresowane do %ld\n",
nr_klienta, kom.mtext, kom.mtype);
} //koniec while(1)
}
```

Kod 9.1. Przykład programu obsługującego kolejki komunikatów – kod klienta

Proces serwera:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
#include <signal.h>
#include <ctype.h>

//maksymalny rozmiar wiadomosci
#define MAX 80
#define SERWER 1
char temp[15];
int wezpid(char[]);
void sig_hand(int);
//struktura komunikatu
struct komunikat {
        long mtype;
        char mtext[MAX];
};
int main(int argc, char *argv[])
{
    char *text;
    int msize;
    int i,pid;
    key_t key;  //unikalny klucz kolejki komunikatow
    int IDkolejki;  //identyfikatow kolejki
    struct komunikat kom;  //przesylany komunikat
    //tworzenie unikalnego klucza urzadzenia IPC dla
kolejki komunikatow
    key = ftok(".", 123);
    if( (IDkolejki = msgget(key, IPC_CREAT | 0660)) == -1)
//tworzenie kolejki
    {
        perror("msgget() calling...");
        exit(1);
    }
    signal(SIGCLD,SIG_IGN);
    signal(SIGINT,sig_hand); //po nacisnieciu przez
uzytkownika CTRL+C wywoluje sie funkcja sig_hand()
    printf("^C konczy prace serwera\a\n");
    sleep(1);
    printf("\a");
    while(1)
    {
    printf("S: Czekam na komunikat...\n");
    kom.mtype = SERWER;//odczytuje z kolejki serwera-typ 1
    msgrcv(IDkolejki, (struct msgbuf *)&kom, MAX,
kom.mtype, 0);
    printf("S: Odebrano od: %s\n", kom.mtext);
    //przetwarzamy wiadomosc
    msize = strlen(kom.mtext);
```

```
    //text = malloc(sizeof(char) * msize);
    for(i=0; i<msize; i++)
    {
      //kom.mtext[i] -= 32; //zamiana liter na duze
      kom.mtext[i]=toupper(kom.mtext[i]);
    }
    pid=wezpid(kom.mtext);
    //wysylanie wiadomosci
    kom.mtype = pid;
    printf("S: Wysylanie... %s -> %ld\n", kom.mtext,
kom.mtype);
    msgsnd(IDkolejki, (struct msgbuf *)&kom,
strlen(kom.mtext)+1, 0);
    }
}
int wezpid(char text[MAX])
{
 int i,pid,len,oldi;
 len=strlen(text);
 for(i=0;i<12;i++)
 {
  temp[i]=text[i];
  if(temp[i]=='~')
  {
   temp[i+1]='\n';
   break;
  }
 }
 oldi=i;
 for(i=0;i<len-oldi;i++)
 {
  text[i]=text[i+1+oldi];
 }
 pid=atoi(temp);
 return pid;
}
void sig_hand(int sig_n)
{
 key_t key;
 int IDkolejki;
 if((sig_n==SIGTERM) || (sig_n==SIGINT))
 {
   printf("SIGTERM\n");
   key = ftok(".", 123);
   //tworzenie kolejki
   IDkolejki = msgget(key, IPC_CREAT | 0660);
   //usuwanie
```

```
    msgctl(IDkolejki, IPC_RMID, 0);
    exit(0);
  }
}
```

Kod 9.2. Przykład programu obsługującego kolejki komunikatów – kod serwera

9.3. Pamięć współdzielona

Pamięć współdzielona jest specjalnie utworzonym segmentem wirtualnej przestrzeni adresowej, do którego dostęp może mieć wiele procesów. Jest to najszybszy sposób komunikacji pomiędzy procesami. Szybkość dostępu jest taka sama jak dla niewspółdzielonej pamięci procesu. Ponieważ jądro nie synchronizuje dostępu do pamięci współdzielonej, użytkownik sam musi o taką synchronizację zadbać. Aby używać współdzielonego segmentu pamięci, proces musi taki segment zaalokować, co powoduje utworzenie stron pamięci wirtualnej. Każdy proces, który chce go używać, musi taki segment dołączyć, co powoduje dodanie pozycji odwzorowujących adresy z jego pamięci wirtualnej na współdzielone strony segmentu. Gdy przestajemy używać segmentu, trzeba usunąć te odwzorowania. Podczas tworzenia segmentu pamięci współdzielonej tworzona jest systemowa struktura danych o nazwie `shmid_ds`. Definicję tej obsługiwanej przez system struktury można znaleźć w pliku nagłówkowym <sys/shm.h>.

Funkcje operujące na pamięci współdzielonej zdefiniowane są w plikach: <sys/ipc.h> i <sys/shm.h>.

Proces może utworzyć segment pamięci współdzielonej za pomocą funkcji `semget()`.

```
int shmget (key_t key, size_t size, int shmflags)
```

Pierwszym argumentem jest klucz, który jednoznacznie identyfikuje segment. Procesy – jeśli chcą korzystać z segmentu – podają wartość tego klucza. Drugi parametr określa liczbę bajtów w segmencie, zaokrągloną do całkowitej wielokrotności rozmiaru strony ze względu na alokację segmentu za pomocą stron. Trzecim parametrem są połączone operacją logiczną `or` flagi, m.in.:

- `IPC_CRATE` – tworzy nowy segment o podanej wartości klucza.
- `IPC_EXCL` – używana zawsze z `IPC_CREATE` powoduje, że jeśli podamy istniejący już klucz segmentu, to funkcja się nie powiedzie i dzięki temu proces wywołujący dostanie segment na wyłączność. Jeśli nie podamy tej flagi i uży-

jemy klucza istniejącego segmentu, funkcja nie utworzy nowego segmentu, ale zwróci klucz istniejącego.

- Flagi trybu – wartość utworzona z 9 bitów określających prawa dostępu (takie same jak do pliku) do segmentu.

Aby proces mógł skorzystać z segmentu pamięci współdzielonej, musi ten segment dołączyć do swojej przestrzeni adresowej za pomocą funkcji `shmat()`, której jako pierwszy argument trzeba przekazać identyfikator segmentu zwrócony przez funkcję `shmget()`:

```
void *shmat(int shmid, const void *addr, inf shmflags)
```

Drugim argumentem jest wskaźnik określający, gdzie w przestrzeni adresowej procesu chcemy odwzorować pamięć – jeśli podamy NULL, Linux wybierze dostępny adres.

Trzecim argumentem są flagi, m.in.:

- SHM_RND – adres podany trzeba zaokrąglić w dół do wielokrotności rozmiaru strony.
- SHM_RDONLY – segment służy tylko do odczytu.

Gdy segment współdzielonej pamięci przestaje być potrzebny, należy go odłączyć za pomocą funkcji `shmdt()`.

```
int shmdt(const void *addr)
```

Funkcja `shmctl()` zwraca informacje o współdzielonym segmencie pamięci i umożliwia jego modyfikację.

```
int shmclt(int shmid, int cmd,struct shmid_ds *buf)
```

Jeśli chcemy pobrać informacje o segmencie, pierwszym argumentem jest identyfikator segmentu, drugim stała IPC_STAT, trzecim zaś wskaźnik struktury `shmid_ds`. Aby usunąć segment, drugi argument musi być stałą IPC_RMID, a trzeci argument to NULL. Usunięcie segmentu nastąpi po odłączeniu segmentu przez ostatni proces, który z tego segmentu korzysta. Poniżej zamieszczony jest przykład wykorzystania pamięci współdzielonej:

```
#include <stdio.h>
#include <sys/shm.h>
```

```
#include<sys/stat.h>

int main()
{
int id_segmentu;
char* adres;
struct shmid_ds bufor;
int rozmiar_segmentu;
const int shared_segment_size =0x6400;

/* Alokacja współdzielonej pamięci */
id_segmentu=shmget(IPC_PRIVATE,rozmiar_segmentu,
IPC_CREAT|0600);
   if (id_segmentu==-1) {
     printf("Problemy z utworzeniem segmentu\n");
     exit(EXIT_FAILURE);}
   else printf("Pamiec utworzona %d\n",pamiec);

/* Dołączenie segmentu */
adres=(char*) shmat(id_segmentu,0,0);
    if (*adres==-1){
        printf("Problem z przydzieleniem adresu.\n");
        exit(EXIT_FAILURE);}
    else printf("Adres przydzielony : %s\n",adres);

/* Wpis do pamięci współdzielonej */
printf("Wpisz cos do pamieci:");
scanf("%s",adres);

/* Sprawdzenie rozmiaru segmentu    */
shmctl(id_segmentu,IPC_STAT,&bufor);
rozmiar_segmentu = shmbuffer.shm_segsz;
printf("Rozmiar segmentu: %d\n", rozmiar_segmentu);

/*Odłączenie segmentu */
shmdt(adres);

/* Dezalokacja segmentu pamięci  */
shmctl (id_segmentu,IPC_RMID,0);
  }
```

Kod 9.3. Przykład wykorzystania pamięci współdzielonej

Istnieją dwa polecenia na poziomie powłoki, które umożliwiają użycie urządzeń IPC. Pierwsze z nich, `ipcs`, drukuje informację o bieżącym stanie urządzeń IPC. Opcja `-m` dostarcza informacji o pamięci współdzielonej. Przykład użycia:

```
$ ipcs -m
```

Kolejne polecenie używane do usuwania z systemu pozostawionego segmentu pamięci (pod warunkiem, że użytkownik jest właścicielem urządzenia) to `ipcrm`:

```
$ ipcs -shm identyfikator segmentu
```

Współdzielone segmenty pamięci umożliwiają szybką, dwukierunkową komunikację między dowolną liczbą procesów. Niestety, system Linux nie gwarantuje wyłącznego dostępu, dlatego jeśli wiele procesów korzysta z danego współdzielonego segmentu, muszą uzgadniać użycie tego samego klucza.

9.4. Semafory

Semafory służą do synchronizacji procesów. Pozwalają na czasowe zabezpieczenie jakiegoś zasobu przed innymi procesami. Semafory procesów są alokowane, stosowane i dezalokowane tak samo jak współdzielone segmenty pamięci. Operacje semaforowe Linuxa są nastawione na pracę z zestawami semaforów, a nie z pojedynczymi obiektami. Funkcja `semget` służy do alokowania semaforów:

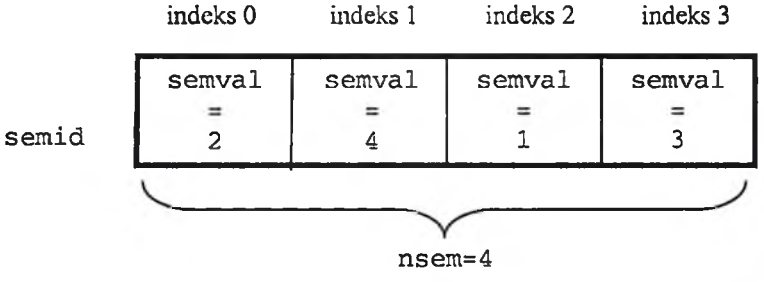
```
int semget(key_t key, int nsem, int permflags);
```

Ta funkcja na podstawie klucza tworzy lub umożliwia nam dostęp do zbioru semaforów. Parametr `key` jest kluczem do zbioru semaforów. Jeżeli różne procesy chcą uzyskać dostęp do tego samego zbioru semaforów, muszą użyć tego samego klucza. Parametr `nsem` to liczba semaforów, która ma znajdować się w tworzonym zbiorze. Parametr `permflags` określa prawa dostępu do semaforów oraz sposób wykonania funkcji. Może przyjmować następujące wartości:

- `IPC_CREAT` – uzyskanie dostępu do zbioru semaforów lub utworzenie nowego, gdy zbiór nie istnieje.
- `IPC_EXCL` – w połączeniu z `IPC_CREAT` zwraca błąd, gdy zbiór już istnieje.
- `prawa dostępu` – tak samo jak dla plików, np. `0600`.

Oczywiście, poszczególne flagi można łączyć ze sobą za pomocą sumy bitowej. Funkcja zwraca identyfikator zbioru semaforów lub –1, gdy wystąpił błąd (ustawiana jest zmienna `errno`).

Z każdym semaforem w zestawie związane są następujące wartości:



- `semval` – wartość semafora (zawsze dodatnia liczba całkowita). Musi być ustawiana za pomocą funkcji systemowej semafora – oznacza to, że semafor nie jest dostępny dla programu jako obiekt danych.
- `sempid` – identyfikator procesu, który ostatnio miał do czynienia z semaforem.
- `semncnt` – liczba procesów, które czekają aż semafor osiągnie wartość większą niż jego wartość aktualna.
- `semzcnt` – liczba procesów, które czekają aż semafor osiągnie wartość zerową.

Semafor zdefiniowany jest w następujący sposób:

```
union semnum
{
   int val;
   struct semid_ds *buf;
   unsigned short int *array;
}
```

Zaraz po ich stworzeniu semafory należy zainicjować, aby uniknąć późniejszych błędów. Służy do tego funkcja `semctl`.

```
semctl(semid, sem_num, commmand, semun ctl_arg);
```

Funkcja służy do sterowania semaforami i ma następujące parametry:

- `semid` – numer zbioru semaforów,
- `sem_num` – numer semafora w zbiorze (numeracja zaczyna sie od zera),
- `command` – polecenie, jakie ma być wykonane na zbiorze semaforów,
- `ctl_arg` – parametry polecenia.

Funkcja `semctl` należy do trzech kategorii standardowych funkcji IPC:

Tabela 9.2

Kategorie standardowych funkcji IPC

Standardowe funkcje IPC (struktura `semid_ds` zdefiniowana jest w <sys/sem.h>)	
`IPC_STAT`	umieszcza informację o stanie w `ctl_arg.stat`
`IPC_SET`	ustawia informację o prawach własności i dostępu z `ctl_arg.stat`
`IPC_RMID`	usuwa zestaw semaforów z systemu
Operacje na pojedynczym semaforze – wartości zwracane przez `semctl` (dotyczą semafora `sem_num`)	
`GETVAT`	zwraca wartość semafora, czyli `semval`
`SETVAL`	ustawia wartość semafora w `ctl_arg.val`
`GETPID`	zwraca wartość `sempid`
`GETNCNT`	zwraca `semncnt`
`GETZCNT`	zwraca `semzcnt`
Operacje na wszystkich semaforach	
`GETALL`	umieszcza wszystkie `semvals` w `ctl_arg.array`
`SETALL`	ustawia wszystkie `semvals` zgodnie z `ctl_arg.array`

Alokowanie i inicjowanie semaforów to dwie oddzielne operacje. Aby zainicjować semafor, musimy użyć funkcji `semctl`. Każdy semafor ma nieujemną wartość i umożliwia wykonanie operacji opuszczenia lub podniesienia semafora. Operacje te wykonuje wywołanie systemowe `semop`.

```
int semop(int semid, struct sembuf *op_array, size_t
num_ops);
```

- Parametr semid musi być ważnym identyfikatorem semafora (wynik semget).
- Parametr op_array to tablica struktur sembuf (struktura sembuf jest zdefiniowana w <sys/sem.h>).
- Parametr num_ops jest liczbą struktur sembuf w tablicy. Każda struktura sembuf zawiera specyfikację operacji do wykonania na semaforze.

Wywołanie semop wykonuje się niepodzielnie na zestawach semaforów. Struktura sembuf zawiera następujące składowe:

```
struct sembuf
{
unsigned short sem_num;/* indeks semafora w zestawie
*/
short sem_op; /* określa co zrobić  */
short sem_flag;
}
```

sem_op>0	operacja V – powoduje zwiększenie wartości semafora o sem_op; jeśli jakiś proces czeka na nową wartość semafora, zostanie obudzony
sem_op<0	operacja P – wstrzymuje proces lub powoduje zmniejszenie wartości semafora o sem_op
sem_op=0	operacja czeka do chwili, gdy wartość semafora stanie się zerem

Funkcja semop podejmuje próbę wykonania wszystkich wskazywanych operacji. Gdy chociaż jedna z operacji nie będzie możliwa do wykonania, nastąpi blokada procesu lub błąd wykonania funkcji semop, zależnie od ustawienia flagi. Aby operacja nie blokowała procesu, należy podać flagę IPC_NOWAIT, wtedy, jeżeli operacja miałaby blokować, wywołanie semop nie powiedzie się. Jeśli podamy flagę SEM_UNDO, Linux automatycznie cofnie operacje blokujące na semaforze przy zakończeniu procesu.

Implementacje operacji semaforowych na semaforze ogólnym, czyli operacji podnoszenia semafora (zwiększania wartości zmiennej semaforowej o 1) i operacji opuszczania semafora (zmniejszania wartości zmiennej semaforowej o 1), ilustruje poniższy przykład:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
static struct sembuf buf;
void podnies(int semid, int semnum){
    buf.sem_num = semnum;
    buf.sem_op = 1;
    buf.sem_flg = 0;
    if (semop(semid, &buf, 1) == -1){
          perror("Podnoszenie semafora");
          exit(1);
    }
}

void opusc(int semid, int semnum){
    buf.sem_num = semnum;
    buf.sem_op = -1;
    buf.sem_flg = 0;
    if (semop(semid, &buf, 1) == -1){
          perror("Opuszczenie semafora");
          exit(1);
    }
}
```

Kod 9.4. Przykład implementacji operacji semaforowych

10. Mechanizmy synchronizacji wątków[10]

Wątek może także wyłączyć możliwość anulowania siebie za pomocą funkcji:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL)
```

Pozwala ona na zrealizowanie sekcji krytycznej, czyli sekwencji kodu wykonywanego w całości bez przerwy aż do zakończenia. Aby przywrócić poprzednią wartość stanu anulowania, na końcu sekcji krytycznej zamiast ustawiania wartości PTHREAD_CANCEL_ENABLE, można wywołać funkcję `process_transaction()` z innej sekcji krytycznej. Używanie anulowania do kończenia wątków nie jest dobrym pomysłem, lepszą strategią jest przekazanie wątkowi informacji, że powinien się zakończyć.

Załóżmy, że program ma zestaw czekających w kolejce zadań, które są przetwarzane przez klika jednoczesnych wątków. Kolejka zadań realizowana jest za pomocą listy połączonej obiektów. Po zakończeniu działania każdy z wątków sprawdza kolejkę – jeśli wskaźnik nie jest pusty, wątek usuwa początek listy i ustawia wskaźnik na następne zadanie. Funkcja wątku mogłaby mieć następującą postać:

```
#include <malloc.h>
/* Elementy listy */
struct zadanie {
struct zadanie* nastepny;
/* Pozostałe pola ... */
};
/* Lista dowiązaniowa zadań do wykonania */
struct zadanie* kolejka_zadan;
extern void przetwarzaj_zadanie (struct zadanie*);
void* funkcja_watku (void* arg)
{
while (kolejka_zadan != NULL) {
struct zadanie* nastepne_zadanie = kolejka_zadan;
kolejka_zadan = kolejka_zadan->nast;
przetwarzaj_zadanie (nastepne_zadanie);
free (nastepne_zadanie);
}
return NULL;
}
```

Kod 10.1. Przykład funkcji wątku

[10] Na podstawie prac [6 i 8].

Przyjmijmy, że w kolejce pozostało jedno zadanie. Pierwszy wątek sprawdza, czy kolejka jest pusta, stwierdza, że jest zadanie do przetworzenia i zapamiętuje wskaźnik do elementu kolejki w zmiennej nastepne_zadanie. Drugi wątek też sprawdza, czy w kolejce są zadania do wykonania i zaczyna przetwarzać to samo zadanie. Aby wyeliminować tego typu wyścigi, trzeba uczynić niepodzielnymi pewne operacje, które jeśli się rozpoczną, nie mogą być przerwane, dopóki się nie zakończą.

Programowanie z użyciem wątków jest skomplikowane, ponieważ większość programów wielowątkowych w systemie wieloprocesorowym może być wykonywanych w tym samym czasie. Trudne jest debugowanie wielowątkowego programu, gdyż trudno jest powtórzyć zachowanie programu (raz będzie działał dobrze, innym razem się zawiesi – nie ma sposobu na to, aby zmusić system do uszeregowania wątków tak samo). Problemy z wielowątkowością wynikają z jednoczesnego korzystania z tych samych danych. Załóżmy, że program ma zestaw zadań, które są przetwarzane przez kilka jednoczesnych wątków. Rozwiązaniem jest umożliwienie tylko jednemu wątkowi dostępu w danej chwili do kolejki.

10.1. Muteksy

System Linux umożliwia korzystanie z muteksów, czyli specjalnych blokad wzajemnie się wykluczających. Gdy jeden wątek ma odblokowany muteks, to pozostałe są blokowane.

Aby powołać muteks, należy utworzyć zmienną typu `pthread_mutex_t` i przekazać jej wskaźnik funkcji `pthread_mutex_init`.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
pthread_mutexattr_t *attr);
```

Zmienna typu `pthread_mutex_t` może mieć dwa stany: otwarty – nie jest zajęta przez żaden wątek i zamknięty – jest zajęta przez jakiś wątek. Jeśli jeden wątek zamknie zmienną muteksową, a następnie drugi wątek próbuje zrobić to samo, to drugi wątek zostaje zablokowany do momentu, kiedy pierwszy wątek nie otworzy zmiennej muteksowej. Dopiero wtedy może wznowić działanie. Muteks można inicjalizować:

- za pomocą stałej, np. `PTHREAD_MUTEX_INITIALIZER` – domyślne atrybuty, drugi parametr `NULL`,
- z zastosowaniem funkcji `pthread_mutex_init()` – muteks ma swoje atrybuty, które można przekazać za pomocą drugiego argumentu funkcji.

```
pthread_mutex_t mojmuteks;
pthread_mutex_init(&mojmuteks, NULL);
```

Powyższy kod pokazuje sposób utworzenia muteksa z wartościami domyślnymi. Zmienna muteksa powinna być zainicjowana tylko raz.

Wątek może próbować zablokować muteks, wywołując na nim funkcję `pthread_mutex_lock()`. Jeśli muteks nic był zablokowany, funkcja blokuje go i kończy działanie. Jeśli był zablokowany przez inny wątek, funkcja ta blokuje wykonanie i powraca tylko wtedy, kiedy muteks zostanie odblokowany przez inny wątek. Na zablokowany muteks może czekać wiele wątków, a po odblokowaniu tylko jeden z nich (wybrany przypadkowo) zostaje wznowiony. Pozostałe wątki będą nadal czekać, ponieważ muteks ponownie będzie blokowany przez wznowiony wątek. Zazwyczaj funkcje `pthread_mutex_lock()` i `pthread_mutex_unlock()` stosuje się po to, aby chronić swoje współdzielone struktury danych, co ilustruje poniższy przykład:

```
#include <malloc.h>
#include <pthread.h>
struct zadanie {
struct zadanie* nastepny;
/* Pozostałe pola składowe ... */
};
struct zadanie* kolejka_zadan;
extern void przetwarzaj_zadanie (struct zadanie*);
pthread_mutex_t
muteks_kolejki=PTHREAD_MUTEX_INITIALIZER;
void* funkcja_watku (void* arg){
while (1) {
struct zadanie* nast_zadanie;
pthread_mutex_lock (&muteks_kolejki);
if (kolejka_zadan == NULL) nast_zadanie = NULL;
else {
nast_zadanie = kolejka_zadan;
kolejka_zadan = kolejka_zadan->nastepny;
}
pthread_mutex_unlock (&muteks_kolejki);
if (nastepne_zadanie == NULL) break;
przetwarzaj_zadanie(nast_zadanie);
free (nast_zadanie);
}
return NULL; }
```

Kod 10.2. Przykład ochrony współdzielonych struktur danych

Teraz kolejka jest chroniona przez muteks. Przed jej użyciem każdy wątek blokuje muteks. Gdy zostanie wykonana cała sekwencja sprawdzania kolejki, muteks jest odblokowywany, co zapobiega sytuacji wyścigu, która występowała w poprzednim przykładzie. Blokowanie wątków otwiera możliwość wystąpienia zakleszczeń, które pojawią się wówczas, gdy dojdzie do oczekiwania na coś, co się nigdy nie pojawi. Zachowanie zależy od rodzaju muteksów. Istnieją trzy rodzaje muteksów:

- Zablokowanie szybkiego muteksa spowoduje wystąpienie zakleszczenia (domyślny rodzaj).
- Zablokowanie rekurencyjnego muteksa nie spowoduje zakleszczenia, ponieważ można go bezpiecznie blokować wielokrotnie, gdyż pamięta on, ile razy wywołał na nim funkcję `pthread_mutex_lock()` wątek mający blokadę. Musi on wykonać tyle samo razy funkcję `pthread_mutex_unlock()`, zanim wątek zostanie odblokowany i inny wątek będzie mógł go zablokować.
- Oznaczenie podwójnego zamknięcia sprawdzającego błędy muteksa przy drugim wywołaniu funkcji `pthread_mutex_lock()` zwróci kod błędu EDEADLK.

Jeśli wątki będą pracowały zbyt szybko, kolejka zadań się opróżni, wątki zakończą swoje działanie i gdyby w kolejce pojawiło się nowe zadanie, nie będzie już wątków. Problem ten można rozwiązać używając semaforów.

10.2. Semafory dla wątków

Semafor jest licznikiem, którego można używać do synchronizacji wątków. Każdy semafor ma wartość licznika, która jest nieujemną liczbą całkowitą i umożliwia wykonanie dwóch podstawowych operacji:

- Operacja opuszczenia (`wait`) zmniejsza wartość semafora o 1. Jeżeli wartość była zero, to wątki są blokowane do czasu, aż wartość semafora stanie się dodatnia – wtedy wartość semafora jest zmniejszana o 1 i operacja kończy działanie.
- Operacja podniesienia (`post`) zwiększa wartość semafora o 1. Jeśli poprzednio semafor miał wartość równą zero i inne wątki były zablokowane na tym semaforze, jeden z nich jest odblokowywany i jego operacja `wait` kończy działanie.

Semafor jest reprezentowany przez zmienną `sem_t`, którą przed użyciem trzeba zainicjować za pomocą funkcji `sem_init()`. Pierwszy parametr przekazuje wskaźnik do zmiennej `sem_t`, drugi powinien być równy zero, a trzecim jest wartość początkowa semafora. Jeśli semafor nie jest już potrzebny, dobrze jest dezalokować go funkcją `sem_destroy()`. Do podniesienia semafora służy funkcja `sem_post()`. Istnieje jeszcze nieblokująca funkcja `sem_trywait()`. Aby opuścic semafor, należy użyć funkcji `sem_wait()`. Można pobrać wartość licznika semafora za pomocą funkcji `sem_getvalue()`. Przykład zamieszczony poniżej kontroluje kolejkę za pomocą semafora.

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>
struct zadanie {
struct zadanie * nastepny;
/* Pozostałe pola składowe... */
};
struct zadanie* kolejka_zadan;
extern void przetwarzaj_zadanie(struct zadanie*);
pthread_mutex_t muteks_kolejki = PTHREAD_MUTEX_INITIALIZER;
sem_t licznik_kolejki;
void inicjalizuj_kolejke_zadan (){
kolejka_zadan = NULL;
sem_init(&licznik_kolejki, 0, 0);
}
void* funkcja_watku (void* arg){
while (1) {
struct zadanie* nast_zadanie;
sem_wait (&licznik_kolejki);
pthread_mutex_lock (&muteks_kolejki);
nastepne_zadanie = kolejka_zadan;
kolejka_zadan = kolejka_zadan->nastepny;
pthread_mutex_unlock (&muteks_kolejki);
przetwarzaj_zadanie (nast_zadanie);
free (nast_zadanie);
}
return NULL;
}
void wstaw_zadanie_do_kolejki (/* dane zadania */)
{
struct zadanie *nowe_zadanie;
nowe_zadanie=(struct zadanie*) malloc(sizeof (struct
zadanie));
/* wpisz dane do pozostałych pól struktury zadania */
pthread_mutex_lock (&muteks_kolejki);
/* umieść zadanie na początku kolejki */
nowe_zadanie->nastepny = kolejka_zadan;
kolejka_zadan = nowe_zadanie;
/* prześlij informację do semafora o nowym zadaniu */
sem_post (&licznik_kolejki);
/* otwórz muteks */
pthread_mutex_unlock (&muteks_kolejki);
}
```

Kod 10.3. Przykład implementacji operacji semaforowych

Przed pobraniem zadania z początku kolejki każdy z wątków opuszcza najpierw semafor. Jeśli jego wartość wynosi zero, co oznacza, że kolejka jest pusta, wątek zostaje zablokowany do czasu, aż semafor przyjmie wartość dodatnią, oznaczającą dołączenie do kolejki nowego zadania.

10.3. Zmienne warunków

Zmienna warunku to trzeci mechanizm synchronizacji dostępny w systemie Linux, za pomocą którego można zrealizować bardziej skomplikowane warunki wykonywania wątków. Zmienne takie pozwalają na zaimplementowanie warunku, przy którym wątek będzie wykonywany, oraz warunku, przy którym wątek jest zablokowany. Wątek może czekać na zmienną warunku, podobnie jak w wypadku semaforów, do czasu aż inny wątek zasygnalizuje tę samą zmienną warunku. Zmienna warunku – w przeciwieństwie do semafora – nie ma licznika lub pamięci. Jeśli wątek B zasygnalizuje zmienną warunku, zanim wątek A zacznie na nią czekać, sygnał zostanie utracony i wątek A będzie nadal zablokowany. Zmienna warunku reprezentowana jest przez zmienną typu `pthread_cond_t()`. Każda zmienna warunkowa jest związana z muteksem, który chroni stan zasobu. Można ją zainicjować na dwa sposoby:

- za pomocą stałej, np. `PTHREAD_COND_INITIALIZER`,
- za pomocą funkcji `pthread_cond_init` – zmienna warunkowa ma swoje atrybuty, które można przekazać z wykorzystaniem drugiego argumentu funkcji; `NULL` oznacza atrybuty domyślne.

Istnieje wiele funkcji operujących na zmiennych warunku:

1. Funkcja `pthread_cond_wait` czeka na zmienną warunkową. Wątek jest budzony za pomocą sygnału lub rozgłaszania. Funkcja odblokowuje muteks przed rozpoczęciem czekania oraz blokuje go po zakończeniu czekania (nawet jeśli zakończy się ono niepowodzeniem lub zostanie anulowane) przed powrotem do funkcji, z której ją wywołano.
2. Funkcja `pthread_cond_timedwait` czeka na zmienną warunkową tylko przez określony czas, podany w trzecim argumencie funkcji.
3. Funkcja `pthread_cond_signal` ustawia zmienną warunkową `cond`, co powoduje obudzenie jednego z wątków. Korzysta się z niej wtedy, kiedy obudzony ma być tylko jeden wątek.
4. Funkcja `pthread_cond_broadcast` rozgłasza ustawienie zmiennej warunkowej `cond`.
5. Funkcja `pthread_cond_destroy` usuwa zmienną warunkową.

Prosta implementacja zmiennej warunku została przedstawiona w poniższym kodzie.

```
#include <pthread.h>
extern void wykonaj_prace ();
int flaga_watku;
pthread_mutex_t flaga_muteksu;
void initialize_flag (){
pthread_mutex_init (&flaga_muteksu, NULL);
flaga_watku = 0;
}
void* watek (void* thread_arg){
while (1) {
int flaga_ustawiona;
/* Chron flage za pomoca muteksu */
pthread_mutex_lock (&flaga_muteksu);
flaga_ustawiona = flaga_watku;
pthread_mutex_unlock (&flaga_muteksu);
if (flaga_ustawiona)
wykonaj_prace ();
/* W przeciwnym wypadku nic nie rób */
}
return NULL;
} void ustaw_flage_watku (int wartosc_flagi){
/* Chron flage za pomoca muteksu */
pthread_mutex_lock (&flaga_watku_mutex);
flaga_watku = wartosc_flagi;
pthread_mutex_unlock (&flaga_muteksu);
}
```

Kod 10.4. Przykład implementacji zmiennej warunku

Może również dochodzić do zakleszczenia wątków. Występują one wówczas, gdy co najmniej dwa wątki są zablokowane i oczekują na wystąpienie warunku, który może zostać spełniony przez pozostały wątek. Jeden z często popełnianych błędów powodujących zakleszczenie to próba zablokowania tego samego zestawu obiektów przez więcej niż jeden wątek.

10.4. Implementacja rozwiązania problemu producent–konsument za pomocą kolejki komunikatów

Poniżej przedstawiono przykładowy program umożliwiający komunikację pomiędzy klientem i serwerem z wykorzystaniem wątków. Proces serwera tworzy kolejkę komunikatów, z której pobierany jest komunikat. Proces klienta do wysyłania komunikatu wykorzystuje wątki.

klient.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/msg.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<pthread.h>

typedef struct strukura_komunikatu{
    long int typ_komunikatu;
    char wiadomosc[16];
    int nadawca;
} Komunikat;
int msgid;
int koniec;

void wysylanie(){
     Komunikat kom, *komunikat;
     komunikat=&kom;
     while( fgets(komunikat->wiadomosc,15,stdin) != NULL ){
 komunikat->typ_komunikatu=1;
 komunikat->nadawca=getpid();
 msgsnd(msgid,komunikat,sizeof(Komunikat)-sizeof(long
int),0);
    }
    koniec=1;
}
void odbieranie(){     Komunikat kom, *komunikat;
     komunikat=&kom;
 while(!koniec){
 msgrcv(msgid,komunikat,sizeof(Komunikat)-sizeof(long
int),getpid(),0);
 puts(komunikat->wiadomosc);
     }
}
```

```
int main(){
    pthread_t twysylanie, todbieranie;
    key_t klucz;
    koniec=0;
    if((klucz=ftok("/",'a')) == -1 ){
 printf("Blad tworzenia klucz\n");
 exit(-1);
    }
    if((msgid = msgget(klucz,IPC_CREAT|0666)) == -1){
 printf("Blad tworzenia kolejki\n");
 exit(-1);
    }
    pthread_create(&twysylanie,NULL,&wysylanie,NULL);
    pthread_create(&todbieranie,NULL,&odbieranie,NULL);
    pthread_join(&twysylanie,NULL);
    pthread_join(&todbieranie,NULL);
    return 0;
}
```

serwer.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/msg.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<ctype.h>
typedef struct strukura_komunikatu{
    long int typ_komunikatu;
    char wiadomosc[16];
    int nadawca;
} Komunikat;

int main(){
    key_t klucz;
    int msgid,i;
    char c;
    Komunikat kom,*komunikat;
    komunikat=&kom;
    printf("Tworzenie klucza\n");
    if((klucz=ftok("/",'a')) == -1 ){
printf("Blad tworzenia klucz\n");
exit(-1);
    }
    printf("Tworzenie kolejki\n");
```

```
    if( (msgid = msgget(klucz,IPC_CREAT|0666)) == -1){
  printf("Blad tworzenia kolejki\n");
  exit(-1);
    }
    printf("Serwer wystartował...\n");
    while(1){
  msgrcv(msgid,komunikat,sizeof(Komunikat)-sizeof(long
int),1,0);
  printf("Odebrano wiadomosc od %d:\n%s\n",komunikat-
>nadawca, komunikat->wiadomosc);
  for(i=0;i<5;i++) komunikat->wiadomosc[i]=toupper(komunikat-
>wiadomosc[i]);
  komunikat->typ_komunikatu=komunikat->nadawca;
  msgsnd(msgid,komunikat,sizeof(Komunikat)-sizeof(long
int),0);
    }
    msgctl(msgid,IPC_RMID,0);
}
```

Kod 10.5. Komunikacja między klientem i serwerem z wykorzystaniem wątków

11. Problemy synchronizacji procesów[11]

11.1. Klasyczne typy problemów

Do porównywania różnych mechanizmów synchronizacji zostały sformułowane trzy klasyczne abstrakcyjne problemy: producenta–konsumenta, czytelników i pisarzy oraz pięciu filozofów. Określają one sposoby współpracy i rywalizacji o zasoby współbieżnych procesów.

W **Problemie Producent–Konsument** występują dwa typy współpracujących procesów. Procesy producentów przygotowują pewne porcje informacji, które są pobierane i przetwarzane przez procesy konsumentów. Rozważmy rozwiązanie problemu z użyciem trzech semaforów: `s1`, `pusty` i `pełny`. Przyjmujemy następujące założenia:

- Operujemy na puli `n` buforów, z których każdy mieści jedną jednostkę informacji.
- Semafor `s1` umożliwia wzajemne wyłączanie dostępu do puli buforów i jest inicjowany wartością 1 (`s1=1`).
- Semafor `pusty` ma wartość równą liczbie pustych buforów i jest inicjowany wartością n (`pusty=n`).
- Semafor pełny ma wartość równą liczbie zapełnionych buforów i jest inicjowany wartością 0 (`pełny=0`).

```
        producent
begin
repeat
 produkuj jednostkę
      wait(pusty)
      wait(s1)
       dodaj do bufora
      signal(s1)
      signal(pełny)
end
```

```
        konsument
begin
repeat
      wait(pełny)
      wait(s1)
       pobranie jednostki
      signal(s1)
      signal(pusty)
end
```

Alg. 11.1. Pseudokody procesów producenta i konsumenta

[11] Na podstawie prac [1, 5–7, 10, 12].

Kolejny problem programowania współbieżnego, czyli **Problem Czytelników i Pisarzy**, charakteryzuje procesy korzystające z wspólnej bazy danych. Występują dwa typy procesów: procesy czytające i piszące, które współpracują ze sobą, ale równocześnie rywalizują o zasoby. Wiele procesów czytających może jednocześnie korzystać z zasobu, jednak pod warunkiem, że nie korzysta z niego w tym czasie żaden proces piszący. Procesy piszące wymagają wzajemnego wykluczania zarówno względem procesów czytających, jak i innych procesów piszących.

W rozwiązaniu zastosowano dwa binarne semafory `sp` i `w`:

- `sp` – w celu zapewnienia wykluczenia wzajemnego procesu piszącego względem wszystkich innych procesów; inicjowany wartością 1 (`sp=1`),
- `w` – w celu zapewnienia procesowi czytającemu wykluczenia wzajemnego względem innych procesów czytających w chwilach rozpoczynania i kończenia korzystania z zasobu; inicjowany wartością 1 (`w=1`).

W tym rozwiązaniu większy priorytet mają procesy czytające – uzyskują one bezzwłoczny dostęp do zasobu, gdy tylko nie korzysta z niego żaden proces piszący. Zmienna `lc` to licznik procesów czytających.

Oto pseudokody procesów czytającego i piszącego:

```
      czytanie-1
begin
repeat
  wait(w)
  lc=lc+1
  if lc=1 then wait(sp)
  signal(w)
  czytanie
  wait(w)
  lc=lc-1
  if lc=0 then signal(sp)
  signal(w)
end
```

```
      pisanie-1
begin
repeat
  wait(sp)
  pisanie
  signal(sp)
end
```

Alg. 11.2. Pseudokody procesów czytającego i piszącego

Aby – korzystając z prostych semaforów binarnych – zapewnić większy priorytet dla procesów piszących, czyli umożliwić pisanie procesowi piszącemu jak najszybciej się da, należy zastosować dwie zmienne licznikowe i pięć semaforów:

- `wl` – w celu wykluczenia wzajemnego procesów czytających w chwili rozpoczynania i kończenia korzystania z zasobu,
- `sp` – w celu wykluczenia wzajemnego procesu piszącego względem wszystkich innych procesów,

- sc – do ochrony wejścia do sekcji krytycznej procesu czytającego,
- w2 – w celu wykluczenia wzajemnego procesów piszących w chwili rozpoczynania i kończenia korzystania z zasobu,
- w3 – w celu zapewnienia priorytetu pisania nad czytaniem.

Wszystkie semafory inicjowane są wartością 1:

```
w1=w2=w3=sc=sp=1
```

Oto pseudokody dla procesów czytających i piszących:

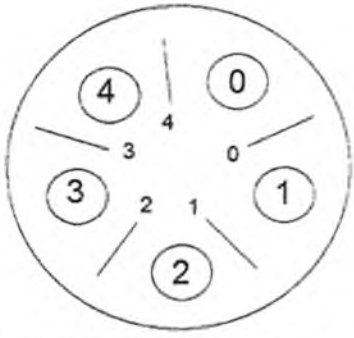
```
        czytanie-2
begin
repeat
  wait(w3)
  wait (sc)
  wait(w1)
  lc=lc+1
  if lc=1 then wait (sp)
  signal (w1)
  signal(sc)
  signal(w3)
  czytanie
  wait (w1)
  lc=lc-1
  iflc=0 then
signal(sp)
  signal (w1)
end
```

```
        pisanie-2
begin
repeat
 wait(w2)
 lp=lp+1
 if lp=1 then wait(sc)
 signal (w2)
 wait(sp)
 pisanie
 signal(sp)
 wait(w2)
 lp=lp-1;
 if lp=0 then signal(sc)
 signal(w2)
end
```

Alg. 11.3. Pseudokody procesów czytającego i piszącego – priorytet dla procesu piszącego

Ostatnim klasycznym problemem programowania współbieżnego jest zdefiniowany przez Dijkstrę **Problem Pięciu Filozofów**. Zadanie polega na napisaniu procedury określającej działania każdego z pięciu filozofów siedzących przy okrągłym stole (rys. 8.1). Każdy z nich powtarza cyklicznie swoje czynności życiowe: myślenie i jedzenie. Przyjęto następujące założenia:

- każdy z filozofów ma swoje miejsce przy okrągłym stole,
- przed każdym filozofem leży jeden talerz,
- po dwóch stronach każdego z talerzy leżą dwa widelce w taki sposób, że prawy widelec każdego filozofa jest jednocześnie lewym widelcem siedzącego po sąsiedzku filozofa,
- do jedzenia potrawy (spaghetti) konieczne są dwa widelce.



Rys. 11.1. Stół obiadujących filozofów

Konfliktowymi zasobami w zadaniu są widelce. Prawidłowa synchronizacja dostępu do zasobów wymaga, aby ten sam zasób (widelec) nie był jednocześnie używany przez dwóch jedzących filozofów.

W rozwiązaniu wykorzystano 5-elementową tablicę binarnych semaforów `sem[5]` – każdy semafor związany jest z jednym widelcem i inicjowany jest wartością 1.
Oto pseudokod procedury wykonywanej przez każdego filozofa:

```
Filozof-rozwiazanie1(name)
begin
repeat
      myślenie;
      wait(sem[name]);
      wait(sem[(name+1)mod 5]);
      jedzenie;
      signal(sem[name]);
      signal(sem[(name+1)mod 5]);
end
```

Alg. 11.4. Procedura wykonywana przez każdego filozofa

Zaproponowane rozwiązanie może prowadzić do zakleszczenia, gdy każdy z filozofów skutecznie podniesie swój lewy widelec (wykona operację `wait(sem[num])`. Wówczas każdy z nich będzie oczekiwał na swój prawy widelec nieskończenie długo (wstrzymanie procesu na operacji `wait(sem[(name+1)mod 5])`).

Istnieje kilka rozwiązań tego problemu, które wykluczają powstanie blokady. W jednym z nich wprowadzono tablicę określającą fazy życia (stany) filozofów i funkcję testującą stany sąsiadów. Każdy filozof może przyjmować następujące stany:

stan[i]=0 – myślenie
stan[i]=1 – chęć jedzenia
stan[i]=2 – jedzenie

Aby uzyskać dostęp do zasobów (obu widelców), proces filozofa wywołuje procedurę test, która sprawdza, czy żaden z sąsiadów nie jest akurat w trakcie jedzenia. Testowanie wykonywane jest w sekcji krytycznej związanej z binarnym semaforem w. Tablica binarnych semaforów sem związana jest z filozofami, a nie z widelcami, jak w poprzednim rozwiązaniu. Po zakończeniu jedzenia, czyli po zwolnieniu zasobów, filozof wykonuje procedurę test dla obu sąsiadów, aby umożliwić im uzyskanie zasobów. Rozwiązanie to może prowadzić do zagłodzenia filozofa w sytuacji, gdy dwóch jego sąsiadów będzie na zmianę żądało zasobów. Oto pseudokod procedury test oraz procedury filozofa:

```
procedure test(i);
begin
 if   stan[(i-1)   mod   5]<>2   and   stan[i]=1   and
 stan[(i+1) mod 5]<>2
 then
  stan[k] =2;
     signal(sem[k]);
end
```

```
                Filozof-rozwiazanie2(name)
begin
repeat
 myślenie;
 wait(w);
 stan[name]:=1;
 test(name);
 signal(w);
 wait(sem[name]);
 jedzenie;
 wait(w);
 stan[name]:=0;
 test((name+1)mod 5);
 test((name-1)mod 5);
 signal(w)
end
```

Alg. 11.5. Procedura filozofa z testowaniem stanów

Kolejne rozwiązanie Problemu Pięciu Filozofów polega na ograniczeniu liczby filozofów mogących przebywać jednocześnie w jadalni do 4. Jest to warunek wystarczający do niewystąpienia zakleszczenia. Wprowadzono dodatkowy semafor – `jadalnia`, który jest zainicjowany wartością 4. Tablica semaforów `sem` oznacza widelce i ma takie znaczenie, jak w rozwiązaniu pierwszym. Rozwiązanie to nie może prowadzić do zagłodzenia. Oto peudokod procedury filozofa dla rozwiązania z jadalnią:

```
Filozof-rozwiazanie3(name)
begin
repeat
 myślenie;
 wait(jadalnia);
 wait(sem[name]);
 wait(sem[(name+1)mod 5]);
 jedzenie;
 signal(sem[name]);
 signal(sem[(name+1)mod 5]);
 signal(jadalnia);
end
end
```

Alg. 11.6. Procedura filozofa dla rozwiązania z jadalnią

Kolejne, czwarte rozwiązanie, które jest również wolne od możliwości zagłodzenia filozofa, to rozwiązanie asymetryczne. Polega ono na tym, że przynajmniej jeden z filozofów wykonuje inny kod, w którym czeka najpierw na swój prawy widelec, a dopiero później na lewy. Sprowadza się to do zamiany kolejności wywołania operacji `wait` na semaforach związanych z widelcami danego filozofa.

We wszystkich zaproponowanych powyżej rozwiązaniach Problemu Pięciu Filozofów synchronizacja procesów – filozofów jest prawidłowa. W rozwiązaniu 1 istnieje możliwość zakleszczenia, w rozwiązaniu 2 może wystąpić zagłodzenie, natomiast w rozwiązaniach 3 i 4 nie może wystąpić ani blokada, ani zagłodzenie. Dowody poprawności można znaleźć w literaturze przedmiotu.

Inne, prostsze rozwiązania problemów synchronizacji procesów można przedstawić za pomocą rozszerzonych operacji semaforowych, które w łatwy sposób można zaimplementować w systemie Linux.

11.2. Rozszerzone operacje semaforowe

Zaproponowano wiele modyfikacji klasycznych operacji semaforowych, które dają nowe możliwości rozwiązywania problemów programowania współbieżnego. Do najpopularniejszych należą: jednoczesne operacje semaforowe, uogólnione operacje semaforowe, jednoczesne uogólnione operacje semaforowe oraz semafory Agerwali.

Jednoczesne operacje semaforowe zostały wprowadzone przez Dijkstrę i polegają na wykonywaniu operacji semaforowych na wielu semaforach jednocześnie.

Operacja PD:

```
PD(s1,s2,...i,..,sn)
zawieszenie procesu do czasu, gdy wartości
wszystkich semaforów si (i=1,...,n) są dodatnie;
 for i:=1 to n do si:=si-1;
```

Operacja VD:

```
VD(s1,s2,...si,...,sm)
 for j:=1 to m do sj:=sj+1;
```

Jednoczesne operacje semaforowe są idealnym narzędziem do rozwiązania Problemu Pięciu Filozofów bez możliwości zakleszczenia. Tablica semaforów sem jest związana z widelcami, analogiczne jak w rozwiązaniu podstawowym **Filozof-rozwiazanie1**. Oto pseudokod procedury filozofa z wykorzystaniem operacji PD i VD:

```
                Filozof-rozwiazanie5(name)
begin
repeat
 myślenie;
 PD(sem[name],sem[(name+1)mod 5]);
 jedzenie;
 VD(sem[name],sem[(name+1)mod 5]);
end
```

Alg. 11.7. Procedura filozofa z zastosowaniem jednoczesnych operacji semaforowych

Uogólnione operacje semaforowe umożliwiają modyfikację wartości semafora o dowolną liczbę całkowitą dodatnią.

n – nieujemne całkowite wyrażenie.

Operacja PN:

```
PN(s, n)
 zawieszenie procesu do czasu, gdy s>=n
 s:=s-n;
```

Operacja VN:

```
VN(s)
 s:=s+n;
```

Semafory uogólnione umożliwiają bardzo proste rozwiązanie problemu czytający–piszący. Wystarczy jeden semafor, który zostanie zainicjowany dużą wartością M, która będzie określała liczbę procesów czytających, mogących jednocześnie korzystać z zasobu. Procesy czytające przed wejściem do sekcji krytycznej będą zmniejszały wartość semafora o 1, natomiast procesy piszące będą mogły wejść do sekcji krytycznej tylko wtedy, gdy semafor będzie miał wartość M, a przy wejściu do sekcji krytycznej zmniejszą jego wartość do 0, blokując tym samym zasób przed wszystkimi innymi procesami. Jest to rozwiązanie preferujące procesy czytające. Oto pseudokody procesów czytającego i piszącego:

```
czytanie-3
begin
repeat
  PN(w,1);
  czytanie;
  VN(w,1)
end
```

```
pisanie-3
begin
repeat
  PN(w,M);
  pisanie
  VN(w,M)
end
```

Alg. 11.8. Algorytmy procesów czytających i piszących z zastosowaniem semaforów uogólnionych

Aby zapewnić wyższy priorytet procesom piszącym, potrzebne będą dwa semafory uogólnione w i r inicjowane również dużą wartością M. Oto pseudokody procesów dla takiego rozwiązania:

```
czytanie-4
begin
repeat
 PN(r,M);
 PN(w,1);
 VN(r,M-1);
 VN(r,1);
 czytanie;
 VN(w,1)
end
```

```
pisanie-4
begin
repeat
    PN(r,1);
    PN(w,M);
    pisanie;
    VN(w,M);
    VN(r,1);
end
```

Alg. 11.9. Algorytmy czytelników i pisarzy preferujące procesy piszące z zastosowaniem uogólnionych semaforów

Jednoczesne uogólnione operacje semaforowe są złożeniem operacji zwielokrotnionych i uogólnionych.

Operacja PA:

```
PA(s1,a1,s2,a2,...sn,an)
 zawieszenie procesu do czasu, gdy dla wszystkich
si (i=1,...,n): si > ai
 for i:=1 to n do si:=si-ai;
```

Operacja VA:

```
VA(s1,a1,s2,a2,...sm,am)
  for j:=1 to m do sj:=sj+aj;
```

Semafory Agerwali są jednoczesnymi operacjami semaforowymi, w których wstrzymanie procesu jest uzależnione od stanu podniesienia wybranych semaforów i opuszczenia innych semaforów.

Operacja PE:

```
PE(s1,s2,...si,...sn;~sn+1,...,~sj,...,~sn+m)
zawieszenie procesu do czasu, gdy dla wszystkich si
(i=1,...,n): si > 0
 i dla wszystkich sj (j=n+1,...,n+m): sj= 0
 for i:=1 to n do si:=si-1;
```

Operacja VE:

```
VE(s1, s2, ...sk, ..., sm)
   for k=1 to m do sk:=sk+1 end;
```

Semafory Agerwali mogą pełnić funkcje liczników procesów i w takiej roli zostały użyte w kolejnym rozwiązaniu problemu czytający–piszący. Zastosowano trzy semafory Agerwali: A, R, M o następującym znaczeniu:
A – licznik procesów piszących, inicjowany wartością 0 (A=0),
R – licznik procesów czytajacych, inicjowany wartością 0 (R=0),
M – zapewnia wykluczanie, inicjowany jest wartością 1 (M=1).
Oto pseudokody procesów:

```
czytanie-5
begin
repeat
 PE(M;A);
 VE(M,R);
 czytanie;
 PE(R);
end
```

```
pisanie-5
begin
repeat
     VE(A);
     PE(M;R);
     pisanie;
     VE(M);
     PE(A);
end
```

Alg. 11.10. Algorytmy czytelników i pisarzy preferujące procesy piszące z wykorzystaniem semaforów Agerwali

11.3. Przykłady implementacji klasycznych problemów synchronizacji procesów

11.3.1. Implementacja rozwiązania problemu producent–konsument za pomocą kolejki komunikatów i pamięci dzielonej

W zadaniu występują trzy typy procesów: główny, konsument i producent. Jako narzędzia synchronizacji wykorzystano pamięć dzieloną i kolejkę komunikatów. Pamięć dzielona składa się z puli MAX buforów przeznaczonych do wymiany informacji pomiędzy producentami i konsumentami (jest to bufor cykliczny) oraz dwóch pól, w których pamiętane są wskaźniki do zapisu (dla producenta) oraz do odczytu (dla konsumenta). Do synchronizacji między procesami producenckimi i konsumentami zastosowano kolejkę komunikatów z dwoma ich typami:

PEŁNY i PUSTY. Proces producenta odbiera komunikat PUSTY, dodaje porcję informacji do bufora cyklicznego i wysyła komunikat PEŁNY. Proces konsumenta pobiera komunikat PEŁNY, odbiera porcję informacji i wysyła komunikat PUSTY. Proces główny tworzy kolejkę komunikatów i obszar pamięci dzielonej, wysyła MAX komunikatów PUSTY, a następnie uruchamia odpowiednią liczbę procesów potomnych – producentów i konsumentów, a po ich zakończeniu zwalnia zasoby systemowe.

Rozwiązanie składa się z trzech plików:

- **konsument.c** – plik z programem konsumenta,
- **producent.c** – plik z programem producenta,
- **mainprog.c** – plik z programem głównym.

Poniżej przedstawione są pliki składające się na rozwiązanie problemu.

konsument.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/shm.h>
struct bufor{
  int mtype;
  int mvalue;
      };
int *pam;
#define MAX2 12
#define MAX 10
#define PELNY 2
#define PUSTY 1
#define zapis pam[MAX+1]
#define odczyt pam[MAX]
int main()
{
key_t klucz, kluczm;
int msgID, shmID;
int i;
struct bufor komunikat;
printf("konsument--------------------------------\n");
if ( (klucz = ftok(".", 'A')) == -1 )
   {
      printf("Blad ftok (A)\n");
      exit(2);
   };
msgID=msgget(klucz, IPC_CREAT | 0666);
if (msgID==-1)
```

```
      {printf("blad klejki komunikatow\n");exit(1);};
kluczm=ftok(".",'B');
shmID=shmget(kluczm, MAX2*sizeof(int), IPC_CREAT|0666);
pam=(int*)shmat(shmID,NULL,0);
if
(msgrcv(msgID,&komunikat,sizeof(komunikat.mvalue),PELNY,0)
==-1)
      {printf("blad odbioru pelnego komunikatu\n");exit(1);}
printf("odebralem komunikat PELNY\n");
printf("konsument - odczyt z bufora: %d\n",pam[odczyt]);
odczyt=(odczyt+1)%MAX;
komunikat.mtype=PUSTY;
if (msgsnd(msgID,&komunikat,sizeof(komunikat.mvalue), 0)==-1)
     {printf("balad wysłania pustego
komunikatu\n");exit(1);}
}
```

producent.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/shm.h>
struct bufor{
  int mtype;
  int mvalue;
      };
int *pam;
#define MAX 10
#define MAX2 12
#define PELNY 2
#define PUSTY 1
#define odczyt pam[MAX]
#define zapis pam[MAX+1]
int main()
{
key_t klucz, kluczm;
int msgID;
int shmID;
int i;
time_t czas;
struct bufor komunikat;
printf("producent----------------------------------\n");
   if ( (klucz = ftok(".", 'A')) == -1 )
```

```
    {printf("Blad ftok (A)\n"); exit(2);};
msgID=msgget(klucz, IPC_CREAT | 0666);
if (msgID==-1)
  {printf("blad klejki komunikatow\n");exit(1);}
kluczm=ftok(".",'B');
shmID=shmget(kluczm, MAX2*sizeof(int),IPC_CREAT|0666);
pam=(int*)shmat(shmID, NULL, 0);
if
(msgrcv(msgID,&komunikat,sizeof(komunikat.mvalue),PUSTY,0)
==-1)
      {printf("blad odbioru pustego komunikatu\n");exit(1);}
time(&czas);
i=((int)czas)%100;
pam[zapis]=i;
zapis=(zapis+1)%MAX;
komunikat.mtype=PELNY;
if (msgsnd(msgID,&komunikat,sizeof(komunikat.mvalue), 0)==-1)
      {
  printf("balad wyslania komunikatu\n");exit(1);};
printf("wyslalem komunikat PELNY\n");
}
```

mainprog.c

```
#include <stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#define P 50   // liczba procesow
#define MAX 10  //rozmiar bufora cyklicznego
#define MAX2 12
#define PUSTY 1   //typy komunikatów
#define PELNY 2
struct bufor{
  long mtype;
  int mvalue;
      };
int main()
{
key_t klucz, kluczm;  //klucze do mechanizmów IPC
int msgID;            //identyfikator kolejki komunikatow
int shmID;           //identyfikator pamieci dzielonej
int i;
struct bufor komunikat;
```

```
if ( (klucz = ftok(".", 'A')) == -1 )  //tworzenie klucza
   {printf("Blad ftok (main)\n"); exit(1);}
// tworzenie kolejki komunikatów
msgID=msgget(klucz,IPC_CREAT|IPC_EXCL|0666);
if (msgID==-1)
  {printf("blad kolejki komunikatow\n"); exit(1);}
kluczm=ftok(".",'B');
//tworzenie pamieci dzielonej
shmID=shmget(kluczm,MAX2*sizeof(int),PC_CREAT|IPC_EXCL|0666);
komunikat.mtype=PUSTY;
//wyslanie MAX komunikatow typu PUSTY
for(i=0;i<MAX;i++)
  {
  if(msgsnd(msgID,&komunikat,sizeof(komunikat.mvalue),0)==-1)
  {printf("blad wyslania kom. pustego\n");exit(1);};
  printf("wyslany pusty komunikat %d\n",i);
  }
//tworzenie procesow potomnych -producentow i konsumentow
   for (i = 0; i < P; i++)
      switch (fork())
      {
         case -1:
            perror("Blad fork (mainprog)");
            exit(2);
         case 0:
            execl("./prod","prod", NULL);
      }
   for(i=0;i<P;i++)
  switch (fork())
  {
  case -1:
      printf("Blad fork (mainprog)\n");
      exit(2);
  case 0:
   execl("./kons","kons",NULL);
  }
//czekanie na zakończenie procesow potomnych
for(i=0;i<2*P;i++)
   wait(NULL);
//zwalnianie zasobow IPC
   msgctl(msgID,IPC_RMID,NULL);
   shmctl(shmID,IPC_RMID, NULL);
   printf("MAIN: Koniec.\n");
}
```

Kod 11.1. Rozwiązanie problemu producent–konsument z zastosowaniem kolejki komunikatów i pamięci dzielonej

11.3.2. Implementacja rozwiązania problemu czytający–piszący z zastosowaniem semaforów Agerwali

Rozwiązanie składa się z pięciu plików:

operacje.h – plik nagłówkowy z deklaracjami funkcji,

operacje.c – plik zawierający definicje funkcji związanych z semaforami, a wśród nich funkcje semaforowe Agerwali: PE i VE opisane w poprzednim rozdziale,

czyt.c – plik z programem czytelnika,

pisz.c – plik z programem pisarza,

mainprog.c – program główny.

Podobnie jak w poprzednim zadaniu, procesy czytające i piszące współdzielą obszar pamięci dzielonej. Poniżej przedstawiono wspomniane pliki.

operacje.h

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/errno.h>
int alokujSem(key_t klucz, int number, int flagi);
void inicjalizujSem(int semID, int number, int val);
int zwolnijSem(int semID, int number);
int P(int semID, int number, int flags);
int PE(int semID; int *tab1, int t1, int *tab2, int t2);
void V(int semID, int number);
int VE(int semID, int *tab, int t);
int valueSem(int semID, int number);
```

operacje.c

```
#include "operacje.h"
int alokujSem(key_t klucz, int number, int flagi)
{
   int semID;
   if ( (semID = semget(klucz, number, flagi)) == -1)
     { perror("Blad semget (alokujSemafor): ");exit(1); }
   return semID;
}
```

```
int zwolnijSem(int semID, int number)
{
   return semctl(semID, number, IPC_RMID, NULL);
}

void inicjalizujSem(int semID, int number, int val)
{
   if ( semctl(semID, number, SETVAL, val) == -1 )
    {perror("Blad semctl (inicjalizujSemafor): "); exit(1);}
}

int P(int semID, int number, int flags)
{
   struct sembuf operacje;
   operacje.sem_num = number;
   operacje.sem_op = -1;
   operacje.sem_flg = 0 | flags;//SEM_UNDO;
   if ( semop(semID, &operacje, 1) == -1 )
    {perror("Blad semop (P)");return -1;}
   return 1;
}

void V(int semID, int number)
{
   struct sembuf operacje;
   operacje.sem_num = number;
   operacje.sem_op = 1;
   operacje.sem_flg = SEM_UNDO;
   if (semop(semID, &operacje, 1) == -1 )
     perror("Blad semop (V): ");
}

int valueSem(int semID, int number)
{
   return semctl(semID, number, GETVAL, NULL);
}

int PE(int semID, int *tab1, int t1, int *tab2, int t2)
{
struct sembuf operacje[4];
int i,j,k;
for(i=0;i<t1;i++)
{
  operacje[i].sem_num=tab1[i];
  operacje[i].sem_op=-1;
  operacje[i].sem_flg=SEM_UNDO;
}
```

```
k=i;
for(j=0;j<t2;j++)
{
  operacje[k].sem_num=tab2[j];
  operacje[k].sem_op=0;
  operacje[k].sem_flg=SEM_UNDO;
  k++;
}
   i=t1+t2;
   if (semop(semID, operacje, i) == -1 )
    {  perror("Blad semop (PE): ");
       return -1;
    }
return 1;
}

int VE(int semID,int *tab, int t)
{
struct sembuf operacje[4];
int i;
for(i=0;i<t;i++)
{
  operacje[i].sem_num=tab[i];
  printf("podnosze %d\n",tab[i]);
  operacje[i].sem_op=1;
  operacje[i].sem_flg=SEM_UNDO;
}
   if (semop(semID, operacje, t) == -1 )
      { perror("Blad semop (VE): ");return -1;}
return 1;
}
```

czyt.c

```
#include <stdio.h>
#include "operacje.h"
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>
#define MAX 10  //rozmiar puli buforow
#define MAX2 12
int main()
```

```
{
key_t kluczs,kluczm; //klucze do mechanizmow IPC
int semID;  //identyfikator zestawu semaforow
int shmID;  //identyfikator pamieci dzielonej
int *bufor; //bufor - pamiec dzielona
int N=3;    //liczba semaforow w zestawie
int i;
int tab1[2],tab2[2]; //tablice z numerami semaforow Agerwali
#define odczyt bufor[MAX] //wskaźnik do odczytu
#define zapis bufor[MAX+1]  //wsk. do zapisu - nie uzywany
printf("czytajacy---------------------------------\n");
if ((kluczs = ftok(".", 'A')) == -1 )//generowanie klucza
     {printf("Blad ftok (A)\n"); exit(2);};
//dolaczenie do zestawu semaforow
semID = alokujSem(kluczs, N, IPC_CREAT | 0666);
if (semID==-1)
    {printf("blad semafora - producent\n"); exit(1);}
kluczm=ftok(".",'B');
//dolaczenie do pamieci dzielonej
shmID=shmget(kluczm, MAX2 * sizeof(int), IPC_CREAT | 0666);
if (shmID==-1)
   {printf("blad pamieci dzielonej\n"); exit(1);}
//przylaczenie pamieci dzielonej
bufor=(int*)shmat(shmID, NULL, 0);
tab1[0]=1; //numer semafora, który ma być>0
tab2[0]=0; //numer semafora, który ma być=0
PE(semID,tab1,1,tab2,1);
tab1[0]=1; //numery semaforow do inkrementacji
tab1[1]=2;
VE(semID,tab1,2);
//odczyt calej puli buforow
for(i=0;i<MAX;i++)
 fprintf(stderr,"czytam %d \n",bufor[i]);
tab1[0]=2; //numer semafora do dekrementacji wartosci
PE(semID,tab1,1,tab2,0);
printf("czytajacy skonczyż\n");
}
```

pisz.c

```
#include <stdio.h>
#include "operacje.h"
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
#include <time.h>
#include <sys/sem.h>
#define MAX 10 //rozmiar puli buforow
#define MAX2 12
int main()
{
key_t kluczs,kluczm; //klucze do IPC
int semID;        //identyfikator zestawu semaforow
int shmID;        //identyfikator pamieci dzielonej
int *bufor;  //bufor - pamiec dzielona
int N=3;          //liczba semaforow w zestawie
int i;
int tab1[2], tab2[2]; //tablice z numerami semaforow do PE
#define odczyt bufor[MAX] //wskaxnik do odczytu
#define zapis bufor[MAX+1]
printf("piszacy--------------------------------\n");
if ( (kluczs = ftok(".", 'A')) == -1 ) //tworzenie klucza
   {printf("Blad ftok (A)\n"); exit(2);};
//dolaczenie do zestawu semaforow
semID = alokujSem(kluczs, N, IPC_CREAT | 0666);
if (semID==-1)
 {printf("blad semafora - piszcy\n");exit(1);}
kluczm=ftok(".",'B');
//dolaczenie do pamieci dzielonej
shmID=shmget(kluczm, MAX2 * sizeof(int), IPC_CREAT | 0666);
if (shmID==-1)
{printf("blad pamieci dzielonej\n");exit(1);}
//przylaczenie pamieci dzielonej
bufor=(int*)shmat(shmID, NULL, 0);
tab1[0]=0;
VE(semID,tab1,1);
tab1[0]=1;
tab2[0]=2;
PE(semID,tab1,1,tab2,1);
//zapis do puli buforow
for(i=0;i<MAX;i++)
 {
  bufor[zapis]=i;
  zapis=(zapis+1)% MAX;
  fprintf(stderr,"zapisalem %d\n",bufor[i]);
 }

tab1[0]=0;
PE(semID,tab1,1,tab2,0);
}
```

mainprog.c

```
#include <stdio.h>
#include <string.h>
#include "operacje.h"
#include<sys/shm.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#define P 3   // liczba procesow
#define MAX 10  //rozmiar puli buforow
#define MAX2 12
int main()
{
key_t kls,klm;  //klucz do semaforow i pam. dzielonej
int semID;   //identyfikator zestawu semaforow
int N = 3;   //liczba semaforow w zestawie
int i;
int shmID;  //inentyfikator pamieci dzielonej
if ( (kls = ftok(".", 'A')) == -1 )   //tworzenie klucza
   {printf("Blad ftok (main)\n"); exit(1);}
semID = alokujSem(kls, N, IPC_CREAT | IPC_EXCL | 0666);
//tworzenie zestawu 3 semaforow
//inicjalizacja semaforow
inicjalizujSem(semID, 0, 0); //licznik procesow piszących A=0
inicjalizujSem(semID, 1, 1); //M=1
inicjalizujSem(semID, 2, 0); //licznik proc. czytajacych R=0
printf("Semafor gotowy!\n");
klm=ftok(".",'B');  //klucz do pamieci dzielonej
//tworzenie pamieci dzielonej
shmID=shmget(klm,MAX2*sizeof(int),IPC_CREAT|IPC_EXCL|0666);
if (shmID==-1)
  {printf("blad shm\n"); exit(1);}
fflush(stdout);
//tworzenie P procesow piszacych
for (i = 0; i < P; i++)
  switch (fork())
       {
       case -1:
         perror("Blad fork (mainprog)");
         exit(2);
       case 0:
         execl("./pisz","pisz", NULL);
       }
//tworzenie P procesow czytajacych
   for(i=0;i<P;i++)
  switch (fork())
  {
  case -1:
```

```
    printf("Blad fork (mainprog)\n");
        exit(2);
  case 0:
    execl("./czyt","czyt",NULL);
  }
//czekanie na zakonczenie procesow czytajacych i piszacych
for(i=0;i<2*P;i++)
   wait(NULL);
zwolnijSem(semID, N);  //zwolnienie zestawu semaforow
shmctl(shmID,IPC_RMID,NULL); //zwolnienie pamieci
dzielonej
printf("MAIN: Koniec.\n");
}
```

Kod 11.2. Rozwiązanie problemu czytający-piszący z wykorzystaniem semaforów Agerwali

11.3.3. Implementacja rozwiązania Problemu Pięciu Filozofów za pomocą semaforów (rozwiązanie 3 – z jadalnią)

W rozwiązaniu tym korzysta się z klasycznych operacji semaforowych P i V, które są zdefiniowane w pliku operacje.c w podrozdziale 10.3.2, oraz pamięci dzielonej, w której przechowywane są widelce. Każdy widelec przechowuje numer filozofa, który go używa, lub ma wartość –1, gdy jest wolny. Rozwiązanie składa się z 4 plików:

- operacje.c i operacje.h – jak w przykładzie z podrodziału 10.3.2,
- filozof.c – kod procesu filozofa,
- mainprog.c – program główny, który tworzy mechanizmy IPC, uruchamia procesy filozofów, czeka na ich zakończenie i zwalnia zasoby.

Oto kody programów:

filozof.c

```
#include <stdio.h>
#include <string.h>
#include "operacje.h"
#include<sys/shm.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int main(int argc, char* argv[])
{
key_t kluczs,kluczm;  //klucz do semaforow i pam. dziel.
int M=5;              //ile razy filozofowie wykonuja petle
int semID;            //identyfikator zestawu semaforow
int N = 6;            //liczba semaforow - widelce+jadalnia
```

```
int i;
int numer;          //numer filozofa
int shmID;          //inentyfikator pamieci dzielonej
char bufor[3];
int *widelec;       //tablica widelcow
if ( (kluczs = ftok(".", 'A')) == -1 )
 { printf("Blad ftok (main)\n"); exit(1); }
//dostanie sie do zestawu semaforow
semID = alokujSem(kluczs, N, IPC_CREAT |  0666);
kluczm=ftok(".",'B'); //tworzenie klucza
//dostep pamieci dzielonej
shmID=shmget(kluczm,5*sizeof(int),IPC_CREAT|0666);
if (shmID==-1)
 {printf("blad shm\n"); exit(1);}
fflush(stdout);
widelec = (int*)shmat(shmID,0,0); //przyłączenie pam. dziel.
numer= atoi(argv[1]); //pobranie numeru filozofa
 for(i=0;i<M;i++)
{
      printf("Filozof %d mysli\n",numer);
 P(semID,5,SEM_UNDO); //podniesienie semafora - jadalnia
      printf("fil. %d wchodzi do jadalni %d raz\n",numer,i);
 P(semID,numer,SEM_UNDO);    //czekanie na lewy widelec
 P(semID,(numer+1)%5,SEM_UNDO);// czekanie na prawy wid.
 //podniesienie widelcow; ustawienie numeru filozofa
 widelec[(numer+1)%5] = numer;
 widelec[numer]= numer;
 printf("\nFilozof  %d je\n",numer); //jedzenie
//zwolnienie widelcow
 widelec[(numer+1)%5] =-1;
 widelec[numer]=-1;
 V(semID,numer);
 V(semID,(numer+1)%5);
      V(semID,5); //zwolnienie jadalni
      printf("filozof %d wychodzi z jadalni\n",numer);
}
}
```

mainprog.c

```
#include <stdio.h>
#include <string.h>
#include "operacje.h"
#include<sys/shm.h>
```

```
#include<sys/ipc.h>
#include<sys/sem.h>
int main(int argc, char *argv[])
{
key_t kluczs,kluczm;  //klucz do semaforow i pam. dzielonej
int semID;        //identyfikator zestawu semaforow
int N = 6;        //liczba semaforow - widelce+jadalnia
int i;
int shmID;        //inentyfikator pamieci dzielonej
char bufor[3];
int *widelec;  //tablica widelcow - pamiec dzielona
if ( (kluczs = ftok(".", 'A')) == -1 )   // klucz
 { printf("Blad ftok (main)\n"); exit(1); }
//tworzenie zestawu 6 semaforow - widelcow + jadalnia
semID = alokujSem(kluczs, N, IPC_CREAT | IPC_EXCL | 0666);
for(i=0;i<5;i++)
inicjalizujSem(semID, i, 1);  //inicj.  semaforow- widelcow
inicjalizujSem(semID, 5, 4);  //inicjalizacja sem. jadalni
kluczm=ftok(".",'B');  //klucz do pamieci dzielonej
//tworzenie pamieci dzielonej
shmID=shmget(kluczm,5*sizeof(int),IPC_CREAT|IPC_EXCL|0666);
if (shmID==-1)
 {printf("blad shm\n"); exit(1);}
fflush(stdout);
widelec = (int*)shmat(shmID,NULL,0);//przydzielenie pam. dz.
for (i=0; i<5; i++)
 widelec[i] = -1;
//tworzenie  procesow filozofow
for (i = 0; i < 5; i++)
    switch (fork())
    {
     case -1:
            perror("Blad fork (mainprog)");
            exit(2);
     case 0:
            sprintf(bufor,"%d",i); // przekazanie numeru
            execl("./filozof","filozof",bufor, NULL);
      }
 for (i=0; i<5; i++)
      wait(NULL);  //czekanie na zakończenie procesow-fil.
zwolnijSem(semID, N);        //zwolnienie zestawu semaforow
shmctl(shmID,IPC_RMID,NULL); //zwolnienie pamieci dzielonej
printf("MAIN: Koniec.\n");
}
```

Kod 11.3. Rozwiązanie Problemu Pięciu Filozofów (z jadalnią)

Literatura

[1] Ben-Ari M., *Podstawy programowania współbieżnego i rozproszonego*, WNT, Warszawa 1996.

[2] Coulouris G., Dollimore J., Kindberg T., *Systemy rozproszone*, Wydawnictwa Nukowo-Techniczne, Warszawa 1998.

[3] Frisch Æ., *Unix Administracja systemu*, Wydawnictwo RM, Warszawa 1997.

[4] Haviland K., Gray D., Salama B., *UNIX Programowanie systemowe*, Wydawnictwo RM, Warszawa 1999.

[5] Iszkowski W., Maniecki M., *Programowanie współbieżne*, WNT, Warszawa 1982.

[6] Linux – manuale.

[7] Mattew N., Stones R., *LINUX Programowanie*, Wydawnictwo RM, Warszawa 1999.

[8] Mitchell M., Oldham J., Samuel A., *LINUX Programowanie dla zaawansowanych*, Wydawnictwo RM, Warszawa 2002.

[9] Petersen R., *Arkana Linux*, Wydawnictwo RM, Warszawa 1997.

[10] Rochkind M.J., *Programowanie w systemie Unix dla zaawansowanych*, WNT, Warszawa 1993.

[11] Silberschatz A., Galvin P., *Podstawy systemów operacyjnych*, WNT, Warszawa 2002.

[12] Shaw C., *Projektowanie logiczne systemów operacyjnych*, WNT, Warszawa 1980.

[13] Tanenbaum A.S., *Rozproszone systemy operacyjne*, PWN, Warszawa 1997.

Spis rysunków

Spis tabel

Spis przykładów

Spis algorytmów

Spis kodów programów