Funkcje jądra systemu operacyjnego UNIX*

Dariusz Wawrzyniak
Dariusz.Wawrzyniak@cs.put.poznan.pl

20 października 2008

Spis treści

*W przypadku wykrycia jakichkolwiek błędów proszę o mail na podany adres.

Lista programów

1 Wstęp

Dostęp do usług jądra systemu operacyjnego UNIX odbywa się poprzez wywołanie odpowiedniej funkcji systemowej. Z punktu widzenie programisty interfejs funkcji systemowych nie różni się niczym od interfejsu funkcji bibliotecznych języka C. Wykonanie funkcji systemowej różni się natomiast od wykonania zwykłej funkcji języka C, gdyż następuje wówczas przełączenie trybu pracy procesora w tzw. *tryb jądra* zwany też *trybem systemowym*.

Wynik wykonania funkcji systemowej może mieć istotny wpływ na działanie nie tylko procesu wywołującego tę funkcję, ale również na inne procesy działające współbieżnie w systemie. Proces wywołujący otrzymuje jednak najczęściej pewne wartości wynikowe, które są przekazywane przez parametry wyjściowe lub jako wartość zwrotna funkcji. Szczególne znaczenie ma wartość zwrotna –1, oznaczająca, że wykonanie funkcji systemowej zakończyło się błędem. Wartość większa lub równa zero oznacza zakończenie poprawne. Jeśli wartość zwrotna nie ma żadnej semantyki, to w przypadku poprawnego zakończenia jest to wartość 0. W nielicznych przypadkach funkcje systemowe nie zwracają żadnej wartości.

W celu stwierdzenie przyczyny wystąpienia błędu w wykonaniu funkcji systemowej po jej zakończeniu, a przed następnym wywołaniem funkcji systemowej należy sprawdzić wartość zmiennej globalnej `errno`. Dla użytkownika wartość zmiennej `errno` jest mało czytelna, więc można uzyskać komunikat związany z daną przyczyną błędu. Wygodnym sposobem przekazania komunikatu o błędzie jest użycie funkcji `perror`, której parametrem jest łańcuch znaków definiowany przez programistę i informujący o miejscu wystąpienia błędu, a wynikiem działania jest przekazanie na tzw. standardowe wyjście awaryjne komunikatu złożonego z przekazanego do funkcji łańcucha znaków oraz systemowego komunikatu o przyczynie błędu. Należy zwrócić uwagę, że **wywołanie funkcji `perror` ma sens tylko wówczas, gdy wykonana wcześniej funkcja systemowa zakończyła się błędem**, czyli zwróciła wartość –1.

Listing 1 prezentuje przykład programu, w którym funkcję `perror` wykorzystano do stwierdzenie przyczyny błędu w otwarciu pliku przez funkcję systemową `open`.

Listing 1: Przykład użycia funkcji `perror`

```
#include <fcntl.h>

main(int argc, char* argv[]){
    int d;

    if ( argc < 2 )
       exit (1);

    d = open(argv[1], O_RDONLY);
    if ( d == -1 ) {
       perror("Blad otwarcia pliku");
       exit (1);
    }

    exit (0);
}
```

2 Pliki

Plik jest pojęciem, z którym spotyka się niemal każdy użytkownik systemu komputerowego, nawet użytkownik końcowy, zajmujący się obsługą komputera w elementarnym zakresie. Popularność tego pojęcia wynika z faktu, że plik jest podstawową formą przechowywania i udostępniania informacji. Dla użytkownika końcowego plik jest zatem abstrakcyjnym obrazem informacji w systemie komputerowym.

Z punktu widzenia systemu operacyjnego lub działających w nim aplikacji plik jest zbiorem odpowiednio powiązanych ze sobą danych, umieszczonych najczęściej w pamięci nieulotnej. Z plikiem związane są pewne atrybuty, z których najważniejszą rolę pełnią identyfikatory, pozwalające w sposób jednoznaczny wskazać konkretny plik w systemie (np. nazwa pliku).

2.1 Operacje na plikach zwykłych

Jądro systemu operacyjnego UNIX udostępnia dwie podstawowe operacje na plikach — *odczyt* i *zapis* — realizowane odpowiednio przez funkcje systemowe `read` i `write`. Z punktu widzenie jądra w systemie UNIX plik nie ma żadnej struktury, tzn. nie jest podzielony na przykład na rekordy. **Plik jest traktowany jako tablica bajtów**, zatem operacje odczytu lub zapisu mogą dotyczyć dowolnego fragmentu pliku, określonego z dokładnością do bajtów.

2.1.1 Otwieranie pliku

Wykonanie operacji wymaga wskazania pliku, na którym operacja ma zostać wykonana. Plik w systemie UNIX identyfikowany jest przez nazwę (w szczególności podaną w postaci ścieżki katalogowej), przy czym podawanie nazwy pliku przy każdym odwołaniu do niego wymagałoby każdorazowego przeszukiwania odpowiednich katalogów w celu ostatecznego ustalenia jego lokalizacji. W celu uniknięcia czasochłonnego przeszukiwania katalogów podczas lokalizowania pliku przy każdej operacji na nim, wprowadzona została funkcja systemowa `open`, której zadaniem jest przydział niezbędnych zasobów w jądrze, umożliwiających wykonywanie dalszych operacji na pliku bez potrzeby przeszukiwania katalogów. Funkcja `open` zwraca *deskryptor*, który jest przekazywany jako parametr aktualny, identyfikujący plik, do funkcji systemowych związanych z operacjami na otwartych plikach. Przy otwieraniu pliku przekazywany jest tryb otwarcia, określający dopuszczalne operacje, jakie można wykonać w ramach tego otwarcia (czyli na uzyskanym deskryptorze), np. *tylko zapis*, *tylko odczyt* lub *zapis i odczyt*. Tryb otwarcia może mieć również wpływ na sposób wykonania tych operacji, np. każda operacja zapisu dopisuje dane na końcu pliku.

`int open(const char *`*pathname*`, int` *flags*`)` — otwarcie pliku. Funkcja zwraca deskryptor otwartego pliku lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
flags	tryb otwarcia:
	`O_WRONLY` — tylko do zapisu,
	`O_RDONLY` — tylko do odczytu,
	`O_RDWR` — do zapisu lub do odczytu.

2.1.2 Zapis i odczyt pliku

Zwrócony przez funkcję `open` (lub inną funkcję systemową) deskryptor służy do identyfikacji pliku w celu wykonania podstawowych operacji dostępu, czyli zapisu lub odczytu. Odczyt pliku polega na skopiowaniu pewnego fragmentu zawartości pliku, określonego z dokładnością do bajtów, do wskazanego obszaru pamięci w przestrzeni adresowej procesu. Odczyt pliku w systemie UNIX realizowany jest przez funkcję systemową `read`. Zapis pliku polega z kolei na umieszczeniu w pliku danych pochodzących z obszaru pamięci w przestrzeni adresowej procesu. Podobnie jak w przypadku odczytu, odpowiedni blok danych określony jest z dokładnością do bajta.

`ssize_t read(int` *fd*`, void *`*buf*`, size_t` *count*`)` — odczyt z pliku. Funkcja zwraca liczbę odczytanych bajtów, lub –1 w przypadku błędu. Zwrócenie wartości 0 oznacza osiągnięcie końca pliku.

Opis parametrów:

fd deskryptor pliku, z którego następuje odczyt danych,

buf adres początku obszaru pamięci, w którym zostaną umieszczone odczytane dane,

count liczba bajtów do odczytu z pliku (nie może być większa, niż rozmiar obszaru pamięci przeznaczony na odczytywane dane).

`ssize_t write(int` *fd*`, const void *`*buf*`, size_t` *count*`)` — zapis do pliku. Funkcja zwraca liczbę zapisanych bajtów, lub –1 w przypadku błędu.

Opis parametrów:

fd deskryptor pliku, do którego zapisywane są dane,

buf adres początku obszaru pamięci, zawierającego blok danych do zapisania w pliku,

count liczba bajtów do zapisania w pliku (rozmiar zapisywanego bloku).

Analizując interfejs funkcji `read` lub `write` łatwo zauważyć, że nie ma w nich możliwości wskazania miejsca w pliku, z którego mają pochodzić odczytywane dane lub w którym należałoby te dane umieścić. Miejsce to wyznacza *wskaźnik bieżącej pozycji*, związany z każdym otwartym plikiem. Początkowa wartość tego wskaźnika (zaraz po otwarciu pliku) wynosi 0, co oznacza pierwszy bajt pliku[1]. Operacja odczytu odczytuje zatem określoną liczbę bajtów począwszy do tego wskaźnika. Wskaźnik z kolei przesuwany jest o liczbę odczytanych bajtów w przód (w kierunku końca pliku). W ten sposób kolejna operacja odczytu spowoduje pobranie kolejnego fragmentu zawartości pliku. W przypadku zapisu ten sam wskaźnik wyznacza miejsce, od którego umieszczone zostaną bajt po bajcie zapisywane dane. Jeśli wskaźnik bieżącej pozycji wskazuje miejsce w środku pliku, nowo zapisywane wartości bajtów nadpiszą wartości istniejące. Jeśli wskaźnik bieżącej pozycji wskazuje na koniec pliku lub zapisywany blok nie mieści się w dotychczasowym rozmiarze pliku, nastąpi rozszerzenie pliku, czyli powiększenie jego rozmiaru.

Wskaźnik bieżącej pozycji można zmieniać niezależnie od wykonywania operacji zapisu lub odczytu. Służy do tego funkcja systemowa `lseek`.

`off_t lseek(int` *fd*`, off_t` *offset*`, int` *whence*`)` — przesunięcie wskaźnika bieżącej pozycji. Funkcja zwraca wartość wskaźnika bieżącej pozycji po przesunięciu, liczoną względem początku pliku lub –1 w przypadku błędu.

Opis parametrów:

fd deskryptor otwartego pliku,

offset wielkość przesunięcia (wartość ujemna oznacza cofanie wskaźnika, tj. przesuwanie w kierunku początku, a wartość dodatnia oznacza przesuwanie do przodu, tj. w kierunku końca pliku),

whence odniesienie dla przesunięcia, może przyjąć jedną z wartości:
`SEEK_SET` — przesunięcie liczone jest względem początku pliku,
`SEEK_END` — przesunięcie liczone jest względem końca pliku,
`SEEK_CUR` — przesunięcie liczone jest względem bieżącej pozycji.

System UNIX nie udostępnia bezpośredniego mechanizmu rozszerzania pliku przez wstawienie danych wewnątrz pliku z jednoczesnym przesunięciem dotychczasowych danych w kierunku końca. Taka operacja musi zostać zrealizowana w ramach programu użytkownika. Podobnie nie jest dostępny bezpośredni mechanizm usuwania fragmentu znajdującego się wewnątrz pliku. Można natomiast usunąć fragment pliku począwszy od określonego miejsca aż do końca pliku, a dokładniej — zredukować rozmiar pliku do określonej długości. Jest to operacja skracania pliku i realizowana jest przez funkcję systemową `truncate` lub `ftruncate`. Funkcje te umożliwiają skrócenie pliku do długości określonej w bajtach i przekazanej jako drugi parametr aktualny.

[1] Indeksowanie bajtów w pliku rozpoczyna się od wartości 0, podobnie jak tablic w języku C.

`int truncate(const char *`*pathname*`, off_t` *length*`)` — skrócenie pliku identyfikowanego przez nazwę. Funkcja zwraca 0 w przypadku pomyślnego zakończenia lub –1 w przypadku błędu.

`int ftruncate(int` *fd*`, off_t` *length*`)` — skrócenie pliku identyfikowanego przez deskryptor. Funkcja zwraca 0 w przypadku pomyślnego zakńczenia lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
fd	deskryptor pliku,
length	rozmiar w bajtach, do którego nastąpi skrócenie.

2.1.3 Zamykanie pliku

Zasoby jądra przydzielone na potrzeby dostępu do pliku można zwolnić poprzez wywołanie funkcji systemowej `close`. Funkcja `close` zamyka otwarty plik, a dokładniej deskryptor tego pliku. W przypadku pliku zwykłego deskryptor należy zamknąć wówczas, gdy nie jest używany do wykonywania operacji na pliku w dalszej części programu lub gdy brakuje zasobów jądra (np. wolnych deskryptorów) na potrzeby otwarcia innych plików. **Wszystkie deskryptory otwartych plików, istniejące w danym procesie są zamykane przez system operacyjny wraz z zakończeniem tego procesu**.

`int close(int` *fd*`)` — zamknięcie deskryptora pliku. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

fd	deskryptor pliku, który zostanie zamknięty.

2.1.4 Tworzenie plików zwykłych

Omówione dotychczas funkcje systemowe dotyczyły plików istniejących już w systemie. Jądro systemu operacyjnego dostarcza też mechanizm tworzenia nowych plików . Mechanizm tworzenia plików zwykłych dostępny jest przez funkcję systemową `creat`, która tworzy plik o nazwie podanej jako parametr aktualny i otwiera utworzony plik w trybie do zapisu, zwracając odpowiedni deskryptor. Jeśli plik o takiej nazwie już istnieje, a proces wywołujący funkcję `creat` ma prawo do zapisu tego pliku, to jego zawartość jest usuwana.

`int creat(const char *`*pathname*`, mode_t` *mode*`)` — utworzenie nowego pliku lub usunięcie jego zawartości, gdy już istnieje oraz otwarcie go do zapisu. Funkcja zwraca deskryptor pliku do zapisu lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa),
mode	prawa dostępu do nowo tworzonego pliku.

2.1.5 Usuwanie pliku

Istniejący w systemie plik można usunąć za pomocą funkcji `unlink`. Formalnie, funkcja `unlink` usuwa dowiązanie do pliku, czyli wpis katalogowy wyspecyfikowany przez nazwę, podaną jako parametr aktualny. Jeśli jest to ostatnie dowiązanie do danego pliku (w szczególności zatem jedyne dowiązanie), plik usuwany jest z systemu. Jeśli nie jest to ostatnie dowiązanie, zmniejszany jest tylko licznik dowiązań, a plik jest dalej dostępny przez inne nazwy.

`int unlink(const char *`*pathname*`)` — usunięcie dowiązania do pliku. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

pathname	nazwa pliku (w szczególności nazwa ścieżkowa).

2.2 Przykłady zastosowania operacji plikowych

Listing 2 przedstawia program do kopiowania pliku. W programie wykorzystano funkcje systemowe `open`, `creat`, `read`, `write` i `close`. Nazwy plików przykazywane są jako argumenty w linii poleceń przy uruchamianiu programu. Jako pierwszy argument przekazywana jest nazwa istniejącego pliku źródłowego, a jako drugi argument przekazywana jest nazwa pliku docelowego, który może zostać dopiero utworzony.

Listing 2: Kopiowanie pliku

```
#include <fcntl.h>
#include <stdio.h>
#define MAX 512

int main(int argc, char* argv[]){
   char buf[MAX];
   int desc_zrod, desc_cel;
   int lbajt;

   if (argc<3){
      fprintf(stderr, "Za malo argumentow. Uzyj:\n");
      fprintf(stderr, "%s <plik zrodlowy> <plik docelowy>\n", argv
         [0]);
      exit(1);
   }

   desc_zrod = open(argv[1], O_RDONLY);
   if (desc_zrod == -1){
      perror("Blad otwarcia pliku zrodlowego");
      exit(1);
   }

   desc_cel = creat(argv[2], 0640);
   if (desc_cel == -1){
      perror("Blad utworzenia pliku docelowego");
      exit(1);
   }

   while((lbajt = read(desc_zrod, buf, MAX)) > 0){
      if (write(desc_cel, buf, lbajt) == -1){
         perror("Blad zapisu pliku docelowego");
         exit(1);
      }
   }
   if (lbajt == -1){
      perror("Blad odczytu pliku zrodlowego");
      exit(1);
   }

   if (close(desc_zrod) == -1 || close(desc_cel) == -1){
      perror("Blad zamkniecia pliku");
      exit(1);
   }

   exit(0);
}
```

Opis programu: W liniach 10–14 następuje sprawdzenie poprawności przekazania argumentów z linii poleceń. Następnie otwierany jest w trybie *tylko do odczytu* plik źródłowy i sprawdzana jest poprawność wykonania tej operacji (linie 16–20). Podobnie tworzony jest

i otwierany w trybie *tylko do zapisu* plik docelowy (linie 22–26). Właściwe kopiowanie zawartości pliku źródłowego do pliku docelowego następuje w pętli w liniach 28–33. Wyjście z pętli `while` następuje w wyniku zwrócenia przez funkcję `read` wartości 0 lub –1. Wartość –1 oznacza błąd, co sprawdzane jest zaraz po zakończeniu pętli w liniach 34–37. Po każdym błędzie funkcji systemowej wyświetlany jest odpowiedni komunikat i następuje zakończenie procesu przez wywołanie funkcji systemowej `exit`. Jeśli wywołania funkcji systemowych zakończą się bezbłędnie, sterowanie dochodzi do linii 39, gdzie następuje zamknięcie plików.

Listing 3 przedstawia program do wyświetlania rozmiaru pliku. W programie wykorzystano funkcje systemowe `open`, `lseek` i `close`. Nazwa pliku przykazywana jest jako argument w linii poleceń przy uruchamianiu programu.

Listing 3: Wyprowadzanie rozmiaru pliku

```
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char* argv[]){
   int desc;
   long rozm;

   if (argc < 2){
      fprintf(stderr, "Za malo argumentow. Uzyj:\n");
      fprintf(stderr, "%s <nazwa pliku>\n", argv[0]);
      exit(1);
   }

   desc = open(argv[1], O_RDONLY);
   if (desc == -1){
      perror("Blad otwarcia pliku");
      exit(1);
   }

   rozm = lseek(desc, 0, SEEK_END);
   if (rozm == -1){
      perror("Blad w pozycjonowaniu");
      exit(1);
   }

   printf("Rozmiar pliku %s: %ld\n", argv[1], rozm);

   if (close(desc) == -1){
      perror("Blad zamkniecia pliku");
      exit(1);
   }

   exit(0);
}
```

Opis programu: W liniach 8–12 następuje sprawdzenie poprawności przekazania argumentów z linii poleceń. Następnie otwierany jest w trybie *tylko do odczytu* plik o nazwie podanej jako argument w linii poleceń i sprawdzana jest poprawność wykonania tej operacji (linie 14–18). Po otwarciu pliku następuje przesunięcie wskaźnika bieżącej pozycji za pomocą funkcji `lseek` na koniec pliku i zarazem odczyt położenia tego wskaźnika względem początku pliku (linia 20). Uzyskany wynik działania funkcji `lseek`, jeżeli nie jest to wartość –1, jest rozmiarem pliku w bajtach. Wartość ta jest wyświetlana na standardowym wyjściu (linia 26), po czym plik jest zamykany (linia 28).

Listing 4 zawiera rozbudowaną wersję programu z listingu 3, w ten sposób, że wyświetlane są rozmiary wszystkich plików, których nazwy zostały przekazane jako argumenty w linii poleceń.

Listing 4: Wyprowadzanie rozmiaru wielu plików

```
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char* argv[]){
   int desc, i;
   long rozm;

   if (argc < 2){
      fprintf(stderr, "Za malo argumentow. Uzyj:\n");
      fprintf(stderr, "%s <nazwa pliku> ...\n", argv[0]);
      exit(1);
   }

   for (i=1; i<argc; i++) {
      desc = open(argv[i], O_RDONLY);
      if (desc == -1){
         char s[50];
         sprintf(s, "Blad otwarcia pliku %s", argv[i]);
         perror(s);
         continue;
      }

      rozm = lseek(desc, 0, SEEK_END);
      if (rozm == -1){
         perror ("Blad w pozycjonowaniu");
         exit(1);
      }

      printf("Rozmiar pliku %s: %ld\n", argv[i], rozm);

      if (close(desc) == -1){
         perror("Blad zamkniecia pliku");
         exit(1);
      }
   }

   exit(0);
}
```

2.3 Deskryptory otwartych plików

Deskryptor otwartego pliku jest indeksem pozycji w *tablicy otwartych plików procesu*, zwanej również *tablicą deskryptorów*. Deskryptory, podobnie jak indeksy tablic w języku C, przyjmują wartości od 0 do wartości o 1 mniejszej od rozmiaru tablicy. Rozmiar tablicy deskryptorów jest najczęściej potęgą liczby 2, np. 64, 128. Rozmiar tablicy deskryptorów limituje liczbę jednocześnie otwartych plików w danym procesie.

Wszystkie funkcje przydzielające deskryptory (np. `open`, `creat`) alokują deskryptor o najniższym wolnym numerze. Programista nie ma bezpośredniego wpływu na przydzielony numer deskryptora i w większości przypadków numer ten nie ma istotnego znaczenia. We wszystkich programach standardowych zakłada się jednak, że pewne deskryptory odgrywają szczególną rolę, polegającą na identyfikacji standardowych strumieni danych w następujący sposób:

- deskryptor nr 0 — standardowe wejście,

- deskryptor nr 1 — standardowe wyjście,
- deskryptor nr 2 — standardowe wyjście awaryjne.

Odczyt danych ze standardowego wejścia sprowadza się do wywołania funkcji `read` na pliku identyfikowanym przez deskryptor nr 0. Podobnie, zapis na standardowe wyjście oznacza wywołanie funkcji `write` na pliku identyfikowanym przez deskryptor nr 1. Deskryptor nr 2 identyfikuje plik, w którym umieszczane są np. komunikaty o błędach.

Szczególna rola wymienionych deskryptorów wynika z faktu, że programy systemowe UNIX'a oraz pewne funkcje biblioteczne przekazują wyniki swojego działania właśnie na standardowe wyjście, czyli do jakiegoś pliku, którego deskryptor ma ustalony numer — 1. Podobnie komunikaty o błędach przekazywane są na standardowe wyjście awaryjne. Tak działają na przykład programy `ls`, `ps`, funkcje biblioteczne `printf`, `perror` itp.

Wykonując zatem w systemie UNIX jakieś polecenie standardowe (np. `ls`, `ps`), uruchamiany jest proces, który zapisuje strumień danych pod deskryptor nr 1. Jeśli standardowe wyjście nie zostanie przekierowane, dane trafią na terminal. Terminal jest plikiem specjalnym, stanowiącym domyślne standardowe wyjście większości poleceń systemowych. Jeśli uruchomimy polecenie z przekierowaniem standardowego wyjścia do pliku (np. `ls > lista.txt`), w odpowiednim procesie przed rozpoczęciem wykonywania programu danego polecenia wskazany plik zostanie otwarty do zapisu i umieszczony pod deskryptorem nr 1.

Programy `ls` i `ps` nie pobierają żadnych danych wejściowych (jedynie argumenty i opcje przekazane w linii poleceń), istnieje natomiast duża grupa programów, które na standardowe wyjście przekazują wyniki przetwarzania danych wejściowych. Typowymi przykładami takich programów są *filtry*: `more`, `grep`, `sort`, `tr`, `cut` itp. Plik z danymi wejściowymi dla tych programów może być przekazany przez podanie jego nazwy jako jednego z argumentów w linii poleceń. Jeśli jednak filtr zostanie użyty bez nazwy pliku w argumentach linii poleceń, proces będzie próbował czytać dane ze standardowego wejścia, czyli otwartego pliku o ustalonym numerze deskryptora — 1. Domyślnym standardowym wejściem jest również terminal, chociaż można na nie skierować np. plik zwykły (`cat < lista.txt`).

Warto w tym miejscu podkreślić, że z punktu widzenie programu nie jest istotne, jaki plik lub jaki rodzaj pliku identyfikowany jest przez dany deskryptor. Ważne jest, jakie operacje można na takim pliku wykonać. W ten sposób przejawia się niezależności plików od urządzeń. Najczęściej wykonywanymi funkcjami na plikach identyfikowanych przez deskryptory 0–2 są `read` i `write`. Warto też zwrócić uwagę na fakt, że funkcja systemowa `lseek` może być wykonywana na pliku o dostępie bezpośrednim (swobodnym), nie może być natomiast wykonana na pliku o dostępie sekwencyjnym, czyli urządzeniu lub łączu komunikacyjnym. Za pomocą `more` można więc cofać się w przeglądanym pliku tylko wówczas, gdy jego nazwa jest przekazana jako parametr w linii poleceń, gdyż tylko wówczas jest on otwierany przez funkcję `open` w procesie `more` i traktowany jest jako plik o dostępie bezpośrednim. W przypadku odczytu danych ze standardowego wejścia plik otwierany jest w procesie powłoki i przekazywany do procesu `more`, który nie wie, z jakiego typu strumieniem ma do czynienia. Wyodrębnienie funkcji `lseek` i ograniczenie jej stosowalności wyłącznie do plików o dostępie bezpośrednim umożliwia zachowanie takiego samego interfejsu dostępu do zawartości pliku (funkcje `read` i `write`) zarówno w przypadku plików o dostępie sekwencyjnym, jak i bezpośrednim.

W związku ze szczególnym charakterem deskryptorów standardowych strumieni danych może okazać się konieczne wymuszenie przydziału takiego deskryptora do jakiegoś pliku tak, żeby plik ten pełnił określoną rolę dla danego procesu, np. rolę standardowego wejściu lub standardowego-wyjścia. Jak już wcześniej wspomniano, nie można wymusić przydziału deskryptora bezpośrednio, można natomiast zmienić numer deskryptora danego pliku już po jego przydzieleniu. W celu zmiany deskryptora należy go najpierw powielić, czyli uzyskać duplikat deskryptora pod oczekiwanym numerem, a następnie za pomocą funkcji systemowej `close` zamknąć dotychczasowy deskryptor, jeśli nie jest już potrzebny. Powielenie deskryptorów umożliwiają funkcje systemowe `dup` i `dup2`.

Funkcja `dup` przydziela deskryptor zgodnie z zasadą alokacji najniższego wolnego numeru. Jeśli ma to być oczekiwany numer, należy zagwarantować, że w momencie wywołania funkcji

`dup` jest on najniższym wolnym numerem. Przykład takiego programu znajduje się na listingu 8 (str. 22)

Większą funkcjonalność oferuje `dup2`, w przypadku której można wskazać numer duplikatu, specyfikując go jako drugi parametr. Jeśli wyspecyfikowany numer jest zajęty nastąpi zamknięcie odpowiadającego mu deskryptora i dopiero wówczas utworzenie duplikatu.

`int dup(int` *fd*`)` — powielenie (duplikacja) deskryptora. Funkcja zwraca numer nowo przydzielonego deskryptora, lub –1 w przypadku błędu.
Opis parametrów:
fd deskryptor, który ma zostać powielony.

`int dup2(int` *oldfd*`, int` *newfd*`)` — powielenie (duplikacja) deskryptora we wskazanym miejscu w tablicy deskryptorów. Funkcja zwraca numer nowo przydzielonego deskryptora, lub –1 w przypadku błędu.
Opis parametrów:
oldfd deskryptor, który ma zostać powielony,
newfd numer nowo przydzielanego deskryptora.

Zastosowanie funkcji `dup2` do przekierowania standardowych strumieni danych pokazane zostało wraz z użyciem łącza nienazwanego na listingu 3 (str. 29) i 4 (str. 30).

2.4 Zadania

2.1. Napisać program do rozpoznawania, czy plik o nazwie podanej jako argument linii poleceń jest plikiem tekstowym.
Wskazówka: wykorzystaj fakt, że plik tekstowy zawiera znaki o kodach 0–127 (można w tym celu użyć funkcji `isascii`).

2.2. Napisać program konwertujący małe litery na duże w pliku o nazwie podanej jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia, a wyniki zapisane na standardowe wyjście.
Wskazówka: odczytaj blok z pliku do bufora, sprawdź kody poszczególnych znaków i jeśli odpowiadają małym literom, to dodaj do kodu znaku różnicę pomiędzy kodem litery `A` i `a` (można też użyć funkcji `toupper`), a następnie zapisz zmodyfikowany blok w tym samym miejscu pliku, z którego był odczytany (cofnij odpowiednio wskaźnik bieżącej pozycji) lub na standardowym wyjściu, jeśli dane pochodziły ze standardowego wejścia.

2.3. Napisać program ustalający liczbę znaków w najdłuższej linii w pliku o nazwie podanej jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.4. Napisać program wyświetlający najdłuższą linię w pliku o nazwie podanej jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.5. Napisać program, który w pliku o nazwie podanej jako ostatni argument zapisze połączoną zawartość wszystkich plików, których nazwy zostały podane linii poleceń przed ostatnim argumentem.

2.6. Napisać program do wyznaczania częstości występowania liter w pliku tekstowym o nazwie podanej jako argument linii poleceń. Wynikiem działania programu powinien być wydruk na standardowym wyjściu określający procentową zawartość poszczególnych liter w całym tekście z pominięciem białych znaków oraz znaków interpunkcji. Jeśli nazwa pliku nie została podana, tekst powinien zostać odczytany ze standardowego wejścia.

2.7. Napisać program, który policzy wszystkie słowa w pliku o nazwie podanej jako argument linii poleceń, przyjmując, że słowa składają się z małych i dużych liter alfabetu oraz cyfr i znaku podkreślenia, a wszystkie pozostałe znaki są separatorami słów. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.8. Napisać program, który wyodrębni wszystkie słowa w pliku o nazwie podanej jako argument linii poleceń oraz zapisze wyodrębnione słowa na standardowym wyjściu, oddzielając je znakiem nowej linii. Założyć, że słowa składają się z małych i dużych liter alfabetu oraz cyfr i znaku podkreślenia, a wszystkie pozostałe znaki są separatorami słów. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia.

2.9. Napisać program do porównywania dwóch plików o nazwach przekazanych jako argumenty linii poleceń. Wynikiem działania programu ma być komunikat, że *pliki są identyczne*, *pliki różnią się począwszy od znaku nr <numer znaku> w linii <numer linii>* lub — gdy jeden z plików zawiera treść drugiego uzupełnioną na końcu o jakieś dodatkowe znaki — *plik <nazwa> zawiera <liczba> znaków więcej niż zawartość pliku <nazwa>*.

2.10. Napisać program do sortowania w porządku alfabetycznym linii pliku, którego nazwa została przekazana jako argument linii poleceń. Jeśli nazwa pliku nie została podana, dane powinny zostać odczytane ze standardowego wejścia a wynik zapisany na standardowym wyjściu.

2.11. Napisać program do filtrowania linii tekstu odczytywanego ze standardowego wejścia w taki sposób, że jeśli linia odczytanego tekstu zawiera łańcuch znaków przekazanych jako argument linii poleceń, to jest ona zapisywana na standardowym wyjściu, w przeciwnym przypadku jest pomijana.

2.12. Napisać program do znajdowania łańcucha znaków podanego jako pierwszy argument linii poleceń w plikach o nazwach podanych jako pozostałe argumenty linii poleceń. Program powinien informować o nazwie pliku i miejscu (numer linii, numer znaku w linii), w którym dany łańcuch został znaleziony.

2.13.* Napisać program do formatowania akapitów tekstu w plikach o nazwach podanych jako argumenty linii poleceń w taki sposób, żeby długość linii (liczba znaków w linii) nie przekraczała wartości podanej jako pierwszy argument linii poleceń. Przyjąć, że w wyniku formatowania nie może nastąpić podział słowa pomiędzy dwa wiersze i że separatorem formatowanego akapitu jest pusta linia.

2.14.* Zaimplementować mechanizm przyspieszający dostępu do danych w pliku `/etc/passwd` za pomocą pliku indeksowego o strukturze B-drzewa lub B+-drzewa.

2.15.* Zaimplementować mechanizm przyspieszający dostępu do danych w pliku `/etc/passwd` za pomocą tablicy hashowej.

2.5 Pytania kontrolne

1. Jaką wartość zwraca funkcja systemowa w przypadku błędu w jej wykonaniu?
2. Jakie możliwości dostępu do plików oferuje jądro systemu operacyjnego UNIX?
3. Jakie parametry należy przekazać do funkcji `open` w celu otwarcia pliku?
4. Co jest wartością zwrotną funkcji `open`?
5. W jakim trybie można otworzyć plik?
6. Co służy do identyfikacji pliku w celu wykonania operacji odczytu (read) lub zapisu (write)?
7. Jaka jest zasada przydziału numeru deskryptora pliku?

8. Jakie numery deskryptorów przypisane są do standardowych strumieni danych?
9. Jakie funkcje systemowe służą do duplikowania deskryptorów plików?
10. Jakiej funkcji systemowej należy użyć w celu zwolnienia miejsca w tablicy deskryptorów?
11. Kiedy następuje zamknięcie deskryptora otwartego pliku?
12. Jaka funkcja systemowa służy do tworzenia pliku?
13. Co robi funkcja systemowa `creat`?
14. W jakim trybie funkcja systemowa `creat` otwiera plik?
15. Co się dzieje z istniejącym plikiem przy próbie jego utworzenia za pomocą funkcji `creat`?
16. Od jakiego miejsca w pliku odczytywany jest blok danych przez funkcję systemową `read`?
17. Od jakiego miejsca w pliku zapisywany jest blok danych przez funkcję systemową `write`?
18. Jakiej wielkości blok odczytywany jest przez funkcję systemową `read`?
19. Jakiej wielkości blok zapisywany jest przez funkcję systemową `write`?
20. Jaka funkcja systemowa służy do zmiany wskaźnika bieżącej pozycji w pliku?
21. Jaka jest wartość wskaźnika bieżącej pozycji zaraz po otwarciu pliku?
22. Względem czego można określić ustawienie wskaźnika bieżącej pozycji w pliku za pomocą funkcji `lseek`?
23. Jakie parametry należy przekazać do funkcji `lseek`?
24. W jaki sposób można uzyskać rozmiar otwartego pliku?

3 Procesy

Pojęcie *procesu* jest kluczowe dla zrozumienia funkcjonowania wielozadaniowych systemów operacyjnych. Trudność w zrozumieniu tego pojęcia i tym samym zgłębienie mechanizmu obsługi procesów systemu operacyjnego wynika z faktu, że pojęcie procesu utożsamiane jest często z pojęciem programu, który wykonywany jest w ramach procesu. Poza tym samo pojęcie procesu jest dość trudne do zdefiniowania, gdyż z punktu widzenie programisty proces jest pojęciem dość abstrakcyjnym.

W celu przybliżenia pojęcia *proces* należy sobie uświadomić, jakie zasoby sprzętowe i programowe systemu komputerowego są niezbędne do wykonania programu. Do zasobów sprzętowych warunkujących wykonanie programu należą:

- procesor — wykonuje instrukcje zawarte w programie,
- pamięć operacyjna — przechowuje dane do przetworzenia, tymczasowe dane pomocnicze, wyniki pośrednie oraz końcowe, a także zgodnie z architekturą von Neumana — program, czyli ciąg instrukcji,
- urządzenia zewnętrzne — służą do komunikacji ze światem zewnętrznym, czyli umożliwiają pobieranie danych do przetwarzania oraz składowanie i prezentację wyników działania programów.

Oprócz wymienionych zasobów sprzętowych, istotne też są zasoby programowe, tworzone i udostępniane najczęściej przez system operacyjny. Do tej grupy zasobów należą np. pliki, mechanizmy komunikacji między procesami, mechanizmy synchronizacji procesów itp.

Zasoby systemu komputerowego, nadzorowanego przez wielozadaniowy system operacyjny, mogą być wykorzystywane na potrzeby współbieżnego wykonania wielu programów (w szczególności może to być kilka różnych wykonań tego samego programu). Na potrzeby każdego wykonania należy przydzielić odpowiednie zasoby, a zadaniem systemu operacyjnego jest rozwiązywanie konfliktów w dostępie do tych zasobów w przypadku, gdy są one jednocześnie potrzebne dla kilku wykonań. W celu właściwej ewidencji użytkowania zasobów, czyli ich przydziału, kontroli ich wykorzystania oraz odzyskiwania, synchronizacji dostępu do nich itp., wprowadzone zostało właśnie pojęcie procesu, oznaczające wykonanie programu.

Program jest zatem rozumiany jako ciąg instrukcji do wykonania przez procesor, do wykonania przez interpreter lub do dalszego przetworzenia na inny ciąg instrukcji (do kompilacji). Proces natomiast jest abstrakcyjnym pojęciem, określającym program w trakcie wykonywania wraz z niezbędnymi do tego wykonania zasobami systemu komputerowego. Proces identyfikuje zatem przetwarzanie danych, realizowane przez system komputerowy.

3.1 Identyfikacja procesów w systemie UNIX

Każdy proces w systemie musi mieć swój unikalny identyfikator, odróżniający go od innych procesów i umożliwiający jednoznaczne wskazanie konkretnego procesu, gdy wymagają tego mechanizmy systemu operacyjnego. W środowisku systemu operacyjnego UNIX powszechnie używanym skrótem terminu *identyfikator procesu* jest PID (ang. Process IDentyfier). Jednym z atrybutów procesu jest też identyfikator jego przodka, określany skrótem PPID.

PID jest liczbą całkowitą typu `pid_t` (w praktyce `int`). Każdy nowo tworzony proces otrzymuje jako identyfikator kolejną liczbę typu `pid_t`, a po dojściu do końca zakresu liczb danego typu przydział rozpoczyna się od początku. Przydzielane są oczywiście tylko identyfikatory nie używane w danej chwili przez inne procesy.

PID danego procesu nie może ulec zmianie, można go natomiast łatwo uzyskać wywołując funkcję systemową `getpid`. Identyfikator przodka można uzyskać za pomocą funkcji `getppid`.

`pid_t getpid(void)` — uzyskanie własnego identyfikatora (PID) przez proces. Funkcja zwraca identyfikator procesu lub –1 w przypadku błędu.

`pid_t getppid(void)` — uzyskanie identyfikatora procesu macierzystego (PPID) przez potomka. Funkcja zwraca identyfikator procesu lub –1 w przypadku błędu.

Procesy powstają na potrzeby realizacji zadań, zlecanych przez użytkowników, w związku z czym działają one na rzecz tych użytkowników i z użytkownikiem związana jest domena ochrony w systemie UNIX [3]. Dla autoryzacji dostępu kluczowe są zatem dwa identyfikatory: *identyfikatora użytkownika* i *identyfikator grupy*, do której dany użytkownik należy. Identyfikator te dziedziczone są od przodka i jeśli są to identyfikatory zwykłego użytkownika, to nie mogą być zmieniane. Właściciel procesu nie może więc ulec zmianie.

Trwałe związanie praw dostępu procesów do zasobów systemu (np. do plików) może czasami nadmiernie ograniczać dostępność istotnych dla danego procesu informacji. Typowym przykładem może być proces zmiany hasła użytkownika. Proces taki musi mieć dostęp do pliku `/etc/shadow`, do którego zwykły użytkownik nie ma żadnych praw (nawet prawa odczytu!). Modyfikacja pliku `/etc/shadow` na zlecenie zwykłego użytkownika byłaby więc niemożliwa.

W systemie UNIX wyróżniono zatem dwa rodzaje identyfikatorów: *rzeczywisty* (ang. real) i *obowiązujący* (ang. effective). Rzeczywisty identyfikator użytkownika oraz rzeczywisty identyfikator grupy identyfikują właściciela procesu i w przypadku procesu użytkownika zwykłego nie ulegają zmianie przez cały czas życia procesu. Obowiązujący identyfikator użytkownika oraz obowiązujący identyfikator grupy decydują o prawach dostępu, tzn. te identyfikatory brane są pod uwagę przy autoryzacji dostępu do zasobów. Proces ma więc takie prawa dostępu, jakie ustawione są dla użytkownika o identyfikatorze obowiązującym lub dla grupy o identyfikatorze obowiązującym.

Jak już wcześniej wspomniano, identyfikatory użytkownika i grupy dziedziczone są od przodka i najczęściej identyfikatory obowiązujące równe są rzeczywistym. Zmiana obowiązującego identyfikatora użytkownika następuje w wyniku uruchomienia w procesie programu z pliku wykonywalnego, w którym ustawiono *bit zmiany obowiązującego identyfikatora użytkownika*. Obowiązujący identyfikator użytkownika przyjmuje wartość identyfikatora właściciela pliku zawierającego program. Podobnie, zmiana obowiązującego identyfikatora grupy następuje, jeśli w pliku z programem w którym ustawiono *bit zmiany obowiązującego identyfikatora grupy*, a obowiązujący identyfikator grupy przyjmuje wartość identyfikatora grupy pliku z programem.

Kolejnym atrybutem procesu jest identyfikator grupy procesów, do której dany proces należy. Grupy procesów definiowane są na potrzeby przekazywania sygnałów. Identyfikator grupy procesów jest PID'em lidera tej grupy. Proces uruchamiany w reakcji na wydanie polecenia przez użytkownika należy do grupy procesu powłoki. Proces może odłączyć się od swojej grupy poprzez utworzenie własnej grupy, dla której będzie liderem. Do grupy tej wejdą wszystkie nowo utworzone procesy potomne, o ile któryś z potomków nie utworzy własnej grupy.

3.2 Obsługa procesów w systemie UNIX

W zakresie obsługi procesów system UNIX udostępnia mechanizm tworzenia nowych procesów, usuwania procesów oraz uruchamiania programów. Każdy proces, z wyjątkiem procesu systemowego o identyfikatorze 0, tworzony jest przez jakiś inny proces, który staje się jego *przodkiem* zwanym też *procesem macierzystym*, *procesem rodzicielskim* lub krótko *rodzicem* (ang. parent). Nowo utworzony proces nazywany jest *potomkiem* lub *procesem potomnym* (child). Procesy w systemie UNIX tworzą zatem drzewiastą strukturę hierarchiczną, podobnie jak katalogi.

3.2.1 Tworzenie procesu

Potomek tworzony jest w wyniku wywołania przez przodka funkcji systemowej `fork`. Po utworzeniu potomek kontynuuje wykonywanie programu swojego przodka od miejsca wywołania funkcji `fork`.

`pid_t fork(void)` — utworzenie procesu potomnego. W procesie macierzystym funkcja zwraca identyfikator (pid) procesu potomnego (wartość większą od 0, w praktyce większą od 1), a

w procesie potomnym wartość 0. W przypadku błędu funkcja zwraca –1, a proces potomny nie jest tworzony.

Listingi 1 przedstawia program, który ma zasygnalizować początek i koniec swojego działania przez wyprowadzenia odpowiedniego tekstu na standardowe wyjście.

Listing 1: Przykład działania funkcji `fork`

```
#include <stdio.h>
2
main(){
4    printf("Poczatek\n");
     fork();
6    printf("Koniec\n");
}
```

Opis programu: Program jest początkowo wykonywany przez jeden proces. W wyniku wywołania funkcji systemowej `fork` (linia 5) następuje rozwidlenie i tworzony jest proces potomny, który kontynuuje wykonywanie programu swojego przodka od miejsca utworzenia. Inaczej mówiąc, od momentu wywołania funkcji `fork` program wykonywany jest przez dwa współbieżne procesy. Wynik działania programu jest zatem następujący:

```
Poczatek
Koniec
Koniec
```

3.2.2 Uruchamianie programu

W ramach istniejącego procesu może nastąpić uruchomienie innego programu wyniku wywołania jednej z funkcji systemowych `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`. Funkcje te będą określane ogólną nazwą `exec`. Uruchomienie nowego programu oznacza w rzeczywistości zmianę programu wykonywanego dotychczas przez proces, czyli zastąpienie wykonywanego programu innym programem, wskazanym odpowiednio w parametrach aktualnych funkcji `exec`.

```
int execl(const char *path, const char *arg, ...)
int execlp(const char *file, const char *arg, ...)
int execle(const char *path, const char *arg , ..., char *const envp[])
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execve(const char *file, char *const argv[], char *const envp[])
```

Uruchomienie programu w ramach procesu. Funkcja zwraca –1 w przypadku błędu lub nic nie zwraca ze względu na zmianę programu w przypadku poprawnego zakończenia.

Opis parametrów:

path	nazwa ścieżkowa pliku z programem (w przypadku podania samej nazwy przyjmuje się, że jest to nazwa w katalogu bieżącym),
file	nazwa pliku z programem,
arg	argument łinii poleceń",
argv	wektor (tablica) argumentów łinii poleceń",
envp	wektor zmiennych środowiskowych.

Bezbłędne wykonanie funkcji `exec` oznacza zatem bezpowrotne zaprzestanie wykonywania bieżącego programu i rozpoczęcie wykonywania programu, którego nazwa jest przekazana jako argument. W konsekwencji, z funkcji systemowej `exec` nie ma powrotu do programu, gdzie nastąpiło jej wywołanie, o ile wykonanie tej funkcji nie zakończy się błędem. Wyjście z funkcji `exec` można więc traktować jako jej błąd bez sprawdzania zwróconej wartości.

Listingi 2 przedstawia program, który — podobnie jak program na listingu 1 — ma zasygnalizować początek i koniec swojego działania, przy czym w programie następuje uruchomienie innego programu, znajdującego się w pliku o nazwie `ls`.

Listing 2: Przykład działania funkcji `exec`

```
#include <stdio.h>

main(){
   printf("Poczatek\n");
   execlp("ls", "ls", "-a", NULL);
   printf("Koniec\n");
}
```

Opis programu: W wyniku wywołania funkcji systemowej `execlp` (linia 5) następuje zmiana wykonywanego programu, zanim sterowanie dojdzie do instrukcji wyprowadzenia napisu `Koniec` na standardowe wyjście (linia 6). Zmiana wykonywanego programu powoduje, że sterowanie nie wraca już do poprzedniego programu i napis `Koniec` nie pojawia się na standardowym wyjściu w ogóle.

Listing 3 przedstawia program, w którym zastosowano zarówno funkcję `fork` do utworzenia procesu, jak i funkcję `execlp` do uruchomienia innego programu w procesie potomnym.

Listing 3: Przykład uruchamiania programów bez synchronizacji przodka z potomkiem

```
#include <stdio.h>

main(){
   printf("Poczatek\n");
   if (fork() == 0){
      execlp("ls", "ls", "-a", NULL);
      perror("Blad uruchmienia programu");
      exit(1);
   }
   printf("Koniec\n");
}
```

Opis programu: Zmiana wykonywanego programu przez wywołanie funkcji `execlp` (linia 6) odbywa się tylko w procesie potomnym, tzn. wówczas, gdy wywołana wcześniej funkcja `fork` zwróci wartość 0 (linia 5). Funkcja `fork` zwraca natomiast 0 tylko procesowi potomnemu. Ponieważ przodek i potomek działają współbieżnie, nie ma żadnych gwarancji, że przodek wyświetli napis `Koniec` po zakończeniu procesu potomnego. Wynik działania może się zatem nieco różnić od oczekiwań.

3.2.3 Zakończenie procesu

Proces może się zakończyć dwojako: w sposób normalny, tj. przez wywołanie funkcji systemowej `exit` lub w sposób awaryjny, czyli przez wywołanie funkcji systemowej `abort` lub w wyniku reakcji na sygnał. Funkcja systemowa `exit` wywoływana jest niejawnie na końcu wykonywania programu przez proces lub może być wywołana jawnie w każdym innym miejscu programu. Funkcja kończy proces i powoduje przekazanie w odpowiednie miejsce tablicy procesów kodu wyjścia, czyli wartości, która może zostać odebrana i zinterpretowana przez proces macierzysty.

`void exit(int` *status*`)` — zakończenie procesu.

Opis parametrów:

status kod wyjścia przekazywany procesowi macierzystemu.

Zakończenie procesu w wyniku otrzymania sygnału nazywane jest zabiciem. Proces może otrzymać sygnał wysłany przez jakiś inny proces (również przez samego siebie) za pomocą funkcji systemowej `kill` lub wysłany przez jądro systemu operacyjnego.

Proces macierzysty może się dowiedzieć o sposobie zakończenie bezpośredniego potomka przez wywołanie funkcji systemowej `wait`. Jeśli wywołanie funkcji `wait` nastąpi przed zakończeniem potomka, przodek zostaje zawieszony w oczekiwaniu na to zakończenie. Po zakończeniu potomka w procesie macierzystym następuje wyjście z funkcji `wait`, przy czym pod adresem wskazanym w parametrze aktualnym umieszczony zostanie *status zakończenia*, który zawiera albo numer sygnału (najmniej znaczące 7 bitów), albo kod wyjścia (bardziej znaczący bajt), przekazany przez potomka jako wartość parametru funkcji `exit`. Najbardziej znaczący bit młodszego bajtu wskazuje, czy nastąpił zrzut zawartości pamięci, czyli czy został utworzony plik `core`.

`pid_t wait(int *`*status*`)` — oczekiwanie na zakończenie potomka. Funkcja zwraca identyfikator (pid) procesu potomnego, który się zakończył lub –1 w przypadku błędu.

Opis parametrów:

status adres słowa w pamięci, w którym umieszczony zostanie status zakończenia.

Listing 4 przedstawia program, w którym obok mechanizmu `fork-exec`, zastosowanego w przykładzie na listingu 3 do uruchomienia programu w procesie potomnym, użyto również funkcji `wait` do zsynchronizowania procesu macierzystego z potomnym. Synchronizacja obu procesów polega w tym przykładzie na zablokowaniu procesu macierzystego do momentu zakończenia wykonywania programu przez potomka. W rezultacie poniższy program sygnalizuje koniec swojego działania zgodnie z oczekiwaniami, tzn. napis `Poczatek` pojawia się przed wynikiem wykonania programu (polecenia) `ls`, a napis `Koniec` pojawia się po zakończeniu wykonywania `ls`.

Listing 4: Przykład uruchamiania programów z synchronizacją przodka z potomkiem

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    if (fork() == 0){
       execlp("ls", "ls", "-a", NULL);
       perror("Blad uruchmienia programu");
       exit(1);
    }
    wait(NULL);
    printf("Koniec\n");
}
```

Opis programu: Podobnie jak w przypadku programu w lisingu 3, zmiana wykonywanego programu przez wywołanie funkcji `execlp` (linia 6) odbywa się tylko w procesie potomnym. W tym przypadku jednak, w celu uniknięcia sytuacji, w której proces macierzysty wyświetli napis `Koniec` zanim nastąpi wyświetlenie listy plików, proces macierzysty wywołuje funkcję `wait`. Funkcja ta powoduje zawieszenie wykonywania procesu macierzystego do momentu zakończenie potomka.

W powyższym programie (listing 4), jak również w innych programach w tym rozdziale założono, że funkcje systemowe wykonują się bez błędów. Program na listingu 5 jest modyfikacją poprzedniego programu, polegającą na sprawdzaniu poprawności wykonania funkcji systemowych.

Listing 5: Przykład uruchamiania programów z kontrolą poprawności

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    switch (fork()){
       case -1:
```

```
            perror("Blad utworzenia procesu potomnego");
            break;
        case 0: /* proces potomny */
            execlp("ls", "ls", "-a", NULL);
            perror("Blad uruchmienia programu");
            exit(1);
        default: /* proces macierzysty */
            if (wait(NULL) == -1)
                perror("Blad w oczekiwaniu na zakonczenie potomka");
    }
    printf("Koniec\n");
}
```

Listingi 6 i 7 przedstawiają program, którego zadaniem jest zademonstrować wykorzystanie funkcji `wait` do przekazywania przodkowi przez potomka *statusu zakończenia procesu*.

Listing 6: Przykład działania funkcji `wait` w przypadku naturalnego zakończenia procesu

```
#include <stdio.h>

main(){
    int pid1, pid2, status;

    pid1 = fork();
    if (pid1 == 0) /* proces potomny */
        exit(7);
    /* proces macierzysty */
    printf("Mam potomka o identyfikatorze %d\n", pid1);
    pid2 = wait(&status);
    printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

Listing 7: Przykład działania funkcji `wait` w przypadku zabicia procesu

```
#include <stdio.h>

main(){
    int pid1, pid2, status;

    pid1 = fork();
    if (pid1 == 0){ /* proces potomny */
        sleep(10);
        exit(7);
    }
    /* proces macierzysty */
    printf("Mam potomka o identyfikatorze %d\n", pid1);
    kill(pid1, 9);
    pid2 = wait(&status);
    printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

3.3 Dziedziczenie tablicy deskryptorów

Proces dziedziczy tablicę deskryptorów od swojego przodka. Jeśli nie nastąpi jawne wskazanie zmiany, standardowym wejściem, wyjściem i wyjściem diagnostycznym procesu uruchamianego przez powłokę w wyniku interpretacji polecenia użytkownika jest terminal, gdyż terminal jest też standardowym wejściem, wyjściem i wyjściem diagnostycznym powłoki. Zmiana standardowego wejścia lub wyjścia możliwa jest dzięki temu, że funkcja systemowa `exec` nie zmienia stanu tablicy deskryptorów. Możliwa jest zatem podmiana odpowiednich deskryptorów w procesie przed

wywołaniem funkcji `exec`, a następnie zmiana wykonywanego programu. Nowo uruchomiony program w ramach istniejącego procesu zastanie ustawione odpowiednio deskryptory otwartych plików i pobierając dane ze standardowego wejścia (z pliku o deskryptorze 0) lub przekazując dane na standardowe wyjście (do pliku o deskryptorze 1) będzie lokalizował je w miejscach wskazanych jeszcze przed wywołaniem funkcji `exec` w programie. Jest to jeden z powodów, dla których oddzielono w systemie UNIX funkcje tworzenie procesu (`fork`) od funkcji uruchamiania programu (`exec`).

Jednym ze sposobów zmiany standardowego wejścia, wyjścia lub wyjścia diagnostycznego jest wykorzystanie faktu, że funkcje alokujące deskryptory (między innymi `creat`, `open`) przydzielają zawsze deskryptor o najniższym wolnym numerze (patrz 2.3, str. 2.3). W programie przedstawionym na listingu 8 następuje przeadresowanie standardowego wyjścia do pliku o nazwie `ls.txt`, a następnie uruchamiany jest program `ls`, którego wynik trafia właśnie do tego pliku.

Listing 8: Przykład przeadresowania standardowego wyjścia

```
#include <stdio.h>

main(int argc, char* argv[]){
   close(1);
   creat("ls.txt", 0600);
   execvp("ls", argv);
}
```

Opis programu: W linii 4 zamykany jest deskryptor dotychczasowego standardowego wyjścia. Zakładając, że standardowe wejście jest otwarte (deskryptor 0), deskryptor numer 1 jest wolnym deskryptorem o najmniejszej wartości. Funkcja `creat` przydzieli zatem deskryptor 1 do pliku `ls.txt` i plik ten będzie standardowym wyjściem procesu. Plik ten pozostanie standardowym wyjściem również po uruchomieniu innego programu przez wywołanie funkcji `execvp` w linii 5. Wynik działania programu `ls` trafi zatem do pliku o nazwie `ls.txt`.

Warto zwrócić uwagę, że wszystkie argumenty z linii poleceń przekazywane są w postaci wektora `argv` do programu `ls`. Program z listingu 8 umożliwia więc przekazanie wszystkich argumentów i opcji, które są argumentami polecenia `ls`. Do argumentów tych nie należy znak przeadresowania standardowego wyjścia do pliku lub potoku (np. `ls > ls.txt` lub `ls | more`). Znaki `>`, `>>`, `<` i `|` interpretowane są przez powłokę i proces powłoki dokonuje odpowiednich zmian standardowego wejścia lub wyjścia przed uruchomieniem programu żądanego przez użytkownika. Nie są to zatem znaki, które trafiają jako argumenty do programu uruchamianego przez powłokę.

3.4 Sieroty i zombi

Jak już wcześniej wspomniano, prawie każdy proces w systemie UNIX tworzony jest przez inny proces, który staje się jego przodkiem. Przodek może zakończyć swoje działanie przed zakończeniem swojego potomka. Taki proces potomny, którego przodek już się zakończył, nazywany jest *sierotą* (ang. orphan). Sieroty adoptowane są przez proces systemowy `init` o identyfikatorze 1, tzn. po osieroceniu procesu jego przodkiem staje się proces `init`.

Program na listingu 9 tworzy proces-sierotę, który będzie istniał przez około 30 sekund.

Listing 9: Utworzenie sieroty

```
#include <stdio.h>

main(){
   if (fork() == 0){
      sleep(30);
      exit(0);
```

```
7     }
      exit(0);
9  }
```

Opis programu: W linii 4 tworzony jest proces potomny, który wykonuję część warunkową (linie 5–6). Proces potomny śpi zatem przez 30 sekund (linia 5), po czym kończy swoje działanie przez wywołanie funkcji systemowej `exit`. Współbieżnie działający proces macierzysty kończy swoje działanie zaraz po utworzeniu potomka (linia 8), osierocając go w ten sposób.

Po zakończeniu działania proces kończy się i przekazuje status zakończenia. Status ten może zostać pobrany przez jego przodka w wyniku wywołania funkcji systemowej `wait`. Do czasu wykonania funkcji `wait` przez przodka status przechowywany jest w tablicy procesów na pozycji odpowiadającej zakończonemu procesowi. Proces taki istnieje zatem w tablicy procesów pomimo, że zakończył już wykonywanie programu i zwolnił wszystkie pozostałe zasoby systemu, takie jak pamięć, procesor (nie ubiega już się o przydział czasu procesora), czy pliki (pozamykane zostały wszystkie deskryptory). Proces potomny, który zakończył swoje działanie i czeka na przekazanie statusu zakończenia przodkowi, określany jest terminem *zombi*.

Program na listingu 10 tworzy proces-zombi, który będzie istniał około 30 sekund.

Listing 10: Utworzenie zombi

```
1 #include <stdio.h>

3 int main(){
      if (fork() == 0)
5        exit(0);
      sleep(30);
7     wait(NULL);
  }
```

Opis programu: W linii 4 tworzony jest proces potomny, który natychmiast kończy swoje działanie przez wywołanie funkcji `exit` (linia 5), przekazując przy tym status zakończenie. Proces macierzysty zwleka natomiast z odebraniem tego statusu śpiąc przez 30 sekund (linia 6), a dopiero później wywołuje funkcję `wait`, co usuwa proces-zombi.

Zombi nie jest tworzony wówczas, gdy jego przodek ignoruje sygnał SIGCLD (używa się też mnemonika SIGCHLD). Szczegóły znajdą się w rozdziale 6.

3.5 Zadania

3.1. Które ze zmiennych `pid1` – `pid5` na listingu 11 będą miały równe wartości?

Listing 11:

```
1 #include <stdio.h>

  main(){
4     int pid1, pid2, pid3, pid4, pid5;

      pid1 = fork();
7     if (pid1 == 0){
         pid2 = getpid();
         pid3 = getppid();
10    }
      pid4 = getpid();
      pid5 = wait(NULL);
13 }
```

3.2. Ile procesów zostanie utworzonych w wyniku uruchomienia programu przedstawionego na listingu 12?

Listing 12:

```
#include <stdio.h>

main(){
   fork();
   fork();
   if (fork() == 0)
      fork();
   fork();
}
```

3.3. Ile procesów zostanie utworzonych w wyniku uruchomienia programu przedstawionego na listingu 13?

Listing 13:

```
#include <stdio.h>

main(){
   fork();
   fork();
   if (fork() == 0)
      exit(0);
   fork();
}
```

3.4. Jaki będzie wynik działania programu (jaka wartość zostanie wyświetlona jako status), jeśli program przedstawiony na listingu 14 zostanie uruchomiony:

(a) z argumentem 1 (uśpienie przodka na czas 1 sekundy przed wywołaniem funkcji systemowej `kill`),

(b) z argumentem 5 (uśpienie przodka na czas 5 sekund przed wywołaniem funkcji systemowej `kill`)?

Listing 14:

```
#include <stdio.h>

main(int argc, char* argv[]){
   int pid1, pid2, status;

   pid1 = fork();
   if (pid1 == 0) { /* proces potomny */
      sleep(3);
      exit(7);
   }
   /* proces macierzysty */
   printf("Mam potomka o identyfikatorze %d\n", pid1);
   sleep(atoi(argv[1]));
   kill(pid1, 9);
   pid2 = wait(&status);
   printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

3.5. Jaki będzie wynik działania programu (co zostanie wypisane na standardowym wyjściu) w wyniku jego wykonania programu z listingu 15, gdy:

(a) CZAS_POTOMKA >> CZAS_RODZICA
(b) CZAS_POTOMKA << CZAS_RODZICA?

Listing 15:

```
#define CZAS_POTOMKA (?)
#define CZAS_RODZICA (?)

int main() {
    int pid;
    pid = fork();
    if ( pid == 0 ){
       sleep( CZAS_POTOMKA );
       exit(7);
    }
    else {
       int status;
       sleep( CZAS_RODZICA );
       kill( pid, 9 );
       wait( &status );
       printf( "Status = %x\n", status );
    }
}
```

3.6. W jaki sposób wynik wykonania programu przedstawionego na listingu 16 zależy od katalogu bieżącego, tzn. co pojawi się na standardowym wyjściu w zależności od tego, jaka jest zawartość katalogu bieżącego?

Listing 16:

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    execl("ls", "ls", "-l", NULL);
    printf("Koniec\n");
}
```

3.6 Pytania kontrolne

1. Ile dodatkowych procesów tworzonych jest przez jednokrotne wywołanie funkcji systemowej `fork`?
2. Jakie parametry przekazujemy do funkcji systemowej `fork` w celu utworzenia procesu potomnego?
3. Co jest wartością zwrotną funkcji `fork`?
4. Czy proces macierzysty może zakończyć działanie przed swoim potomkiem?
5. Kiedy (w jakich warunkach) nastąpi wykonanie następnej instrukcji po wywołaniu funkcji systemowej `exec`?
6. Na którego potomka oczekuje proces macierzysty w funkcji systemowej `wait`?
7. Co jest wartością zwrotną funkcji systemowej `wait`?
8. Jaką funkcję systemową należy wywołać w celu zakończenia procesu?
9. Co jest parametrem funkcji systemowej `exit`?
10. W jaki sposób może nastąpić zakończenie procesu?

11. Co się dzieje z deskryptorami otwartych plików po zakończeniu procesu?
12. W jaki sposób wywołanie funkcji `exec` wpływa na zawartość tablicy deskryptorów procesu?
13. Jaki program wykonuje proces potomny zaraz po utworzeniu?
14. Jakie pliki otwarte są w procesie potomnym zaraz po jego utworzeniu?
15. Jaki proces nie wykonuje żadnego programu?
16. W jakim celu system operacyjny utrzymuje procesy *zombi*?
17. Co dzieje się z procesem po zakończeniu jego przodka?
18. Kto jest przodkiem procesu-sieroty?

4 Łącza

Łącza w systemie UNIX są plikami specjalnymi, służącymi do komunikacji pomiędzy procesami. Łącza mają kilka cech typowych dla plików zwykłych, czyli posiadają swój i-węzeł, posiadają bloki z danymi (chociaż ograniczoną ich liczbę), na otwartych łączach można wykonywać operacje zapisu i odczytu. Łącza od plików zwykłych odróżniają następujące cechy:

- ograniczona liczba bloków — łącza mają rozmiar 4KB – 8KB w zależności od konkretnego systemu,
- dostęp sekwencyjny — na łączach można wykonywać tylko operacje zapisu i odczytu, nie można natomiast przemieszczać wskaźnika bieżącej pozycji (nie można wykonywać funkcji `lseek`),
- sposób wykonywania operacji zapisu i odczytu — dane odczytywane z łącza są zarazem usuwane (nie można ich odczytać ponownie),
- proces jest blokowany w funkcji `read` na pustym łączu, jeśli jest otwarty jakiś deskryptor tego łącza do zapisu,
- proces jest blokowany w funkcji `write`, jeśli w łączu nie ma wystarczającej ilości wolnego miejsca, żeby zmieścić zapisywany blok[2].

Jak już wspomniano przy opisie funkcji systemowej `read` (str. 7), zwrócenie przez nią wartości 0 jest wskazaniem osiągnięcia końca pliku. Koniec odczytu danych z łącza następuje dopiero, gdy wszystkie dane zostaną odczytane i **wszystkie deskryptory do zapisu zostaną zamknięte**. Pozostawienie otwartego deskryptora do zapisu jest interpretowane przez system jako możliwość dalszego przekazywania kolejnych danych, czego konsekwencją jest blokowanie procesów odczytujących w funkcji `read`. Sytuacja taka może prowadzić do zakleszczenia (ang. deadlock). Istotne jest zatem zamykanie zbędnych deskryptorów, zwłaszcza deskryptorów łącza do zapisu.

W systemie UNIX wyróżnia się dwa rodzaje łączy — *łącza nazwane* i *łącza nienazwane*. Zwyczajowo przyjęło się określać łącza nazwane terminem *kolejki FIFO*, a łącza nienazwane terminem *potoki*. Łącze nazwane ma dowiązanie w systemie plików, co oznacza, że jego nazwa jest widoczna w jakimś katalogu i może ona służyć do identyfikacji łącza. Łącze nienazwane nie ma nazwy w żadnym katalogu i istnieje tak długo po utworzeniu, jak długo otwarty jest jakiś deskryptor tego łącza.

4.1 Sposób korzystania z łącza nienazwanego

Ponieważ łącze nienazwane nie ma dowiązania w systemie plików, nie można go identyfikować przez nazwę. Jeśli procesy chcą się komunikować za pomocą takiego łącza, muszą znać jego deskryptory. Oznacza to, że procesy muszą uzyskać deskryptory tego samego łącza, nie znając jego nazwy. Jedynym sposobem przekazania informacji o łączu nienazwanym jest przekazanie jego deskryptorów procesom potomnym dzięki dziedziczeniu tablicy otwartych plików od swojego procesu macierzystego. Za pomocą łącza nienazwanego mogą się zatem komunikować procesy, z których jeden otworzył łącze nienazwane, a następnie utworzył pozostałe komunikujące się procesy, które w ten sposób otrzymają w tablicy otwartych plików deskryptory istniejącego łącza.

Łącze nienazwane tworzy funkcja systemowa `pipe`, zwracając 2 deskryptory — jeden do odczytu danych z łącza, a drugi do zapisu danych do łącza.

[2] Wyjątkiem od tej zasady jest przypadek, w którym łącze funkcjonuje w trybie bez blokowania (jest ustawiona flaga `O_NDELAY`).

`int pipe(int fd[2])` — utworzenie łącza nienazwanego. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

fd tablica 2 deskryptorów, która jest parametrem wyjściowym (`fd[0]` jest deskryptorem potoku do odczytu, a `fd[1]` jest deskryptorem potoku do zapisu).

Listing 1 pokazuje przykładowe użycie łącza do przekazania napisu (ciągu znaków) `Hallo!` z procesu potomnego do macierzystego.

Listing 1: Przykład użycia łącza nienazwanego w komunikacji przodek-potomek

```
main() {
   int pdesk[2];

   if (pipe(pdesk) == -1){
      perror("Tworzenie potoku");
      exit(1);
   }

   switch(fork()){
      case -1:    // blad w tworzeniu procesu
         perror("Tworzenie procesu");
         exit(1);
      case 0:     // proces potomny
         if (write(pdesk[1], "Hallo!", 7) == -1){
            perror("Zapis do potoku");
            exit(1);
         }
         exit(0);
      default: {  // proces macierzysty
         char buf[10];
         if (read(pdesk[0], buf, 10) == -1){
            perror("Odczyt z potoku");
            exit(1);
         }
         printf("Odczytano z potoku: %s\n", buf);
      }
   }
}
```

Opis programu: Do utworzenia i zarazem otwarcia łącza nienazwanego służy funkcja systemowa `pipe`, wywołana przez proces macierzysty (linia 4). Następnie tworzony jest proces potomny przez wywołanie funkcji systemowej `fork` w linii 9, który dziedziczy tablicę otwartych plików swojego przodka. Warto zwrócić uwagę na sposób sprawdzania poprawności wykonania funkcji systemowych zwłaszcza w przypadku funkcji `fork`, która kończy się w dwóch procesach — macierzystym i potomnym. Proces potomny wykonuje program zawarty w liniach 14–19 i zapisuje do potoku ciąg 7 bajtów spod adresu początkowego napisu `Hallo!`. Zapis tego ciągu polega na wywołaniu funkcji systemowej `write` na odpowiednim deskryptorze, podobnie jak w przypadku pliku zwykłego.

Proces macierzysty (linie 20–25) próbuje za pomocą funkcji `read` na odpowiednim deskryptorze odczytać ciąg 10 bajtów i umieścić go w buforze wskazywanym przez `buf` (linia 21). `buf` jest adresem początkowym tablicy znaków, zadeklarowanej w linii 20. Odczytany ciąg znaków może być krótszy, niż to wynika z rozmiaru bufora i wartości trzeciego parametru funkcji `read` (odczytane zostanie mniej niż 10 bajtów). Zawartość bufora, odczytana z potoku, wraz z odpowiednim napisem zostanie przekazana na standardowe wyjście.

Listing 2 zawiera zmodyfikowaną wersję przykładu przedstawionego na listingu 1. W poniższym przykładzie zakłada się, że wszystkie funkcje systemowe wykonują się poprawnie, w związku z czym w kodzie programu nie ma reakcji na błędy.

Listing 2: Przykład odczytu z pustego łącza

```
1  main() {
       int pdesk[2];

4      pipe(pdesk);

       if (fork() == 0){ // proces potomny
7          write(pdesk[1], "Hallo!", 7);
           exit(0);
       }
10     else { // proces macierzysty
           char buf[10];
           read(pdesk[0], buf, 10);
13         read(pdesk[0], buf, 10);
           printf("Odczytano z potoku: %s\n", buf);
       }
16 }
```

Opis programu: Podobnie, jak w przykładzie na listingu 1, proces potomny przekazuje macierzystemu przez potok ciąg znaków `Hallo!`, ale proces macierzysty próbuje wykonać dwa razy odczyt zawartości tego potoku. Pierwszy odczyt (linia 12) będzie miał taki sam skutek jak w poprzednim przykładzie. Drugi odczyt (linia 13) spowoduje zawieszenie procesu, gdyż potok jest pusty, a proces macierzysty ma otwarty deskryptor do zapisu. Gdyby się tak zdarzyło, że proces macierzysty nie odczyta całego bloku zapisanego przez potomka — co teoretycznie jest możliwe — drugie wywołanie funkcji `read` nie zablokuje procesu macierzystego, lecz zwróci kolejną porcję danych zapisanych przez potomka.

Listing 3 pokazuje sposób przejęcia wyniku wykonania standardowego programu systemu UNIX (w tym przypadku `ls`) w celu dalszego przetworzenia (w tym przypadku konwersji małych liter na duże). Pobranie argumentów z linii poleceń umożliwia przekazanie ich do programu wykonywanego przez proces potomny.

Listing 3: Konwersja wyniku polecenia `ls`

```
1  #define MAX 512

   main(int argc, char* argv[]) {
4      int pdesk[2];

       if (pipe(pdesk) == -1){
7          perror("Tworzenie potoku");
           exit(1);
       }
10
       switch(fork()){
           case -1: // blad w tworzeniu procesu
13             perror("Tworzenie procesu");
               exit(1);
           case 0: // proces potomny
16             dup2(pdesk[1], 1);
               execvp("ls", argv);
               perror("Uruchomienie programu ls");
19             exit(1);
           default: { // proces macierzysty
               char buf[MAX];
22             int lb, i;

               close(pdesk[1]);
25             while ((lb=read(pdesk[0], buf, MAX)) > 0){
```

```
            for(i=0; i<lb; i++)
               buf[i] = toupper(buf[i]);
28          if (write(1, buf, lb) == -1){
               perror ("Zapis na standardowe wyjscie");
               exit(1);
31          }
         }
         if (lb == -1){
34          perror("Odczyt z potoku");
            exit(1);
         }
37    }
   }
}
```

Opis programu: Program jest podobny do przykładu z listingu 1, przy czym w procesie potomnym następuje przekierowanie standardowego wyjścia do potoku (linia 16), a następnie uruchamiany jest program `ls` (linia 17). W procesie macierzystym dane z potoku są sukcesywnie odczytywane (linia 25), małe litery w odczytanym bloku konwertowane są na duże (linie 26–27), a następnie blok jest zapisywany na standardowym wyjściu procesu macierzystego. Powyższa sekwencja powtarza się w pętli (linie 25–32) tak długo, aż funkcja systemowa `read` zwróci wartość 0 (lub –1 w przypadku błędu). Istotne jest zamknięcie deskryptora potoku do zapisu (linia 24) w celu uniknięcia zawieszenia procesu macierzystego w funkcji `read`.

Przykład na listingu 4 pokazuje realizację programową potoku `ls | tr a-z A-Z`, w którym proces potomny wykonuje polecenie `ls`, a proces macierzysty wykonuje polecenie `tr`. Funkcjonalnie jest to odpowiednik programu z listingu 3.

Listing 4: Programowa realizacja potoku `ls | tr a-z A-Z` na łączu nienazwanym

```
main(int argc, char* argv[]) {
2   int pdesk[2];

    if (pipe(pdesk) == -1){
5      perror("Tworzenie potoku");
       exit(1);
    }
8
    switch(fork()){
       case -1: // blad w tworzeniu procesu
11        perror("Tworzenie procesu");
          exit(1);
       case 0: // proces potomny
14        dup2(pdesk[1], 1);
          execvp("ls", argv);
          perror("Uruchomienie programu ls");
17        exit(1);
       default: { // proces macierzysty
          close(pdesk[1]);
20        dup2(pdesk[0], 0);
          execlp("tr", "tr", "a-z", "A-Z", 0);
          perror("Uruchomienie programu tr");
23        exit(1);
       }
    }
26 }
```

Opis programu: Program procesu potomnego (linie 16–19) jest taki sam, jak w przykładzie na listingu 3. W procesie macierzystym następuje z kolei przekierowanie standardowego wejścia na pobieranie danych z potoku (linia 22), po czym następuje uruchomienie programu `tr` (linia 23). W celu zagwarantowania, że przetwarzanie zakończy się w sposób naturalny konieczne jest zamknięcie wszystkich deskryptorów potoku do zapisu. Deskryptory w procesie potomnym zostaną zamknięte wraz z jego zakończeniem, a deskryptor w procesie macierzystym zamykany jest w linii 21.

4.2 Sposób korzystania z łącza nazwanego

Operacje zapisu i odczytu na łączu nazwanym wykonuje się tak samo, jak na łączu nienazwanym, inaczej natomiast się je tworzy i otwiera. Łącze nazwane tworzy się poprzez wywołanie funkcji `mkfifo` w programie procesu lub przez wydanie polecenia `mkfifo` na terminalu. Funkcja `mkfifo` tworzy plik specjalny typu łącze podobnie, jak funkcja `creat` tworzy plik zwykły. Funkcja `mkfifo` nie otwiera jednak łącza i tym samym nie przydziela deskryptorów. Łącze nazwane otwierane jest funkcją `open` podobnie jak plik zwykły, przy czym łącze musi zostać otwarte jednocześnie w trybie do zapisu przez jeden proces i w trybie do odczytu przez inny proces. W przypadku wywołania funkcji `open` tylko w jednym z tych trybów proces zostanie zablokowany aż do momentu, gdy inny proces nie wywoła funkcji `open` w trybie komplementarnym.

`int mkfifo(const char *`*pathname*`, mode_t` *mode*`)` — utworzenie pliku specjalnego typu łącze. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.
Opis parametrów:
pathname nazwa pliku (w szczególności nazwa ścieżkowa),
mode prawa dostępu do nowo tworzonego pliku.

Program na listingu 5 pokazuje przykładowe tworzenie łącza i próbę jego otwarcia w trybie do odczytu.

Listing 5: Przykład tworzenie i otwierania łącza nazwanego

```
#include <fcntl.h>

main(){
    mkfifo("kolFIFO", 0600);
    open("kolFIFO", O_RDONLY);
}
```

Opis programu: Funkcja `mkfifo` (linia 4) tworzy plik specjalny typu łącze o nazwie `kolFIFO` z prawem zapisu i odczytu dla właściciela. W linii 5 następuje próba otwarcia łącza w trybie do odczytu. Proces zostanie zawieszony w funkcji `open` do czasu, aż inny proces będzie próbował otworzyć tę samą kolejkę w trybie do zapisu.

Listing 6 pokazuje realizację przykładu z listingu 1, w której wykorzystane zostało łącze nazwane.

Listing 6: Przykład tworzenie i otwierania łącza nazwanego

```
#include <fcntl.h>

main() {
    int pdesk;

    if (mkfifo("/tmp/fifo", 0600) == -1){
       perror("Tworzenie kolejki FIFO");
       exit(1);
    }
```

```
    switch(fork()){
12      case -1: // blad w tworzeniu procesu
           perror("Tworzenie procesu");
           exit(1);
15      case 0:
           pdesk = open("/tmp/fifo", O_WRONLY);
           if (pdesk == -1){
18            perror("Otwarcie potoku do zapisu");
              exit(1);
           }
21         if (write(pdesk, "Hallo!", 7) == -1){
              perror("Zapis do potoku");
              exit(1);
24         }
           exit(0);
        default: {
27         char buf[10];

           pdesk = open("/tmp/fifo", O_RDONLY);
30         if (pdesk == -1){
              perror("Otwarcie potoku do odczytu");
              exit(1);
33         }
           if (read(pdesk, buf, 10) == -1){
              perror("Odczyt z potoku");
36            exit(1);
           }
           printf("Odczytano z potoku: %s\n", buf);
39      }
    }
}
```

Opis programu: Łącze nazwane (kolejka FIFO) tworzona jest w wyniku wykonania funkcji `mkfifo` w linii 6. Następnie tworzony jest proces potomny (linia 11) i łącze otwierane jest przez oba procesy (potomny i macierzysty) w sposób komplementarny (odpowiednio linia 16 i linia 29). W dalszej części przetwarzanie przebiega tak, jak w przykładzie na listingu 1.

Listing 7 jest programową realizacją potoku `ls | tr a-z A-Z`, w której wykorzystane zostało łącze nazwane podobnie, jak łącze nienazwane w przykładzie na listingu 4.

Listing 7: Programowa realizacja potoku `ls | tr a-z A-Z` na łączu nazwanym

```
#include <stdio.h>
#include <fcntl.h>
3
main(int argc, char* argv[]) {
   int pdesk;
6
   if (mkfifo("/tmp/fifo", 0600) == -1){
      perror("Tworzenie kolejki FIFO");
9     exit(1);
   }

12 switch(fork()){
      case -1: // blad w tworzeniu procesu
         perror("Tworzenie procesu");
15       exit(1);
      case 0: // proces potomny
```

```
            close(1);
18          pdesk = open("/tmp/fifo", O_WRONLY);
            if (pdesk == -1){
               perror("Otwarcie potoku do zapisu");
21             exit(1);
            }
            else if (pdesk != 1){
24             fprintf(stderr, "Niewlasciwy deskryptor do zapisu\n");
               exit(1);
            }
27          execvp("ls", argv);
            perror("Uruchomienie programu ls");
            exit(1);
30       default: { // proces macierzysty
            close(0);
            pdesk = open("/tmp/fifo", O_RDONLY);
33          if (pdesk == -1){
               perror("Otwarcie potoku do odczytu");
               exit(1);
36          }
            else if (pdesk != 0){
               fprintf(stderr, "Niewlasciwy deskryptor do odczytu\n");
39             exit(1);
            }
            execlp("tr", "tr", "a-z", "A-Z", 0);
42          perror("Uruchomienie programu tr");
            exit(1);
         }
45    }
   }
```

Opis programu: W linii 7 tworzona jest kolejka FIFO o nazwie `fifo` w katalogu `/tmp` z prawem do zapisu i odczytu dla właściciela. Kolejka ta otwierana jest przez proces potomny i macierzysty w trybie odpowiednio do zapisu i do odczytu (linia 18 i linia 33). Następnie sprawdzana jest poprawność wykonania operacji otwarcia (linie 19 i 34) oraz poprawność przydzielonych deskryptorów (linie 23 i 38. Sprawdzanie poprawności deskryptorów polega na upewnieniu się, że deskryptor łącza do zapisu ma wartość 1 (łącze jest standardowym wyjściem procesu potomnego), a deskryptor łącza do odczytu ma wartość 0 (łącze jest standardowym wejściem procesu macierzystego). Później następuje uruchomienie odpowiednio programów `ls` i `tr` podobnie, jak w przykładzie na listingu 4.

4.3 Przykłady błędów w synchronizacji procesów korzystających z łączy

Operacje zapisu i odczytu na łączach realizowane są w taki sposób, że procesy podlegają synchronizacji zgodnie ze modelem producent-konsument. Nieodpowiednie użycie dodatkowych mechanizmów synchronizacji może spowodować konflikt z synchronizacją na łączu i w konsekwencji prowadzić do stanów niepożądanych typu zakleszczenie (ang. deadlock).

Listing 8 przedstawia przykład programu, w którym może nastąpić zakleszczenie, gdy pojemność łącza okaże się zbyt mała dla pomieszczenia całości danych przekazywanych przez polecenie `ls`.

Listing 8: Możliwość zakleszczenia w operacji na łączu nienazwanym

```
1  #define MAX 512

   main(int argc, char* argv[]) {
4     int pdesk[2];
```

```
    if (pipe(pdesk) == -1){
7       perror("Tworzenie potoku");
        exit(1);
    }
10
    if (fork() == 0){ // proces potomny
        dup2(pdesk[1], 1);
13      execvp("ls", argv);
        perror("Uruchomienie programu ls");
        exit(1);
16  }
    else { // proces macierzysty
        char buf[MAX];
19      int lb, i;

        close(pdesk[1]);
22      wait(0);
        while ((lb=read(pdesk[0], buf, MAX)) > 0){
            for(i=0; i<lb; i++)
25              buf[i] = toupper(buf[i]);
            write(1, buf, lb);
        }
28  }
}
```

Opis programu: Podobnie jak w przykładzie na listingu 3 proces potomny przekazuje dane (wynik wykonania programu `ls`) do potoku (linie 12–15), a proces macierzysty przejmuje i przetwarza te dane w pętli w liniach 23–27. Przed przejściem do wykonania pętli proces macierzysty oczekuje na zakończenie potomka (linia 22). Jeśli dane generowane przez program `ls` w procesie potomnym nie zmieszczą się w potoku, proces ten zostanie zablokowany gdzieś w funkcji `write` w programie `ls`. Proces potomny nie będzie więc zakończony i tym samym proces macierzysty nie wyjdzie z funkcji `wait`. Odblokowanie potomka może nastąpić w wyniku zwolnienia miejsca w potoku przez odczyt znajdujących się w nim danych. Dane te powinny zostać odczytane przez proces macierzysty w wyniku wykonania funkcji `read` (linia 23), ale proces macierzysty nie przejdzie do linii 23 przed zakończeniem potomka. Proces macierzysty blokuje zatem potomka, nie zwalniając miejsca w potoku, a proces potomny blokuje przodka w funkcji `wait`, nie kończąc się. Wystąpi zatem zakleszczenie. Zakleszczenie nie wystąpi w opisywanym programie, jeśli wszystkie dane, generowane przez program `ls`, zmieszczą się w całości w potoku. Wówczas proces potomny będzie mógł się zakończyć po umieszczeniu danych w potoku, w następstwie czego proces macierzysty będzie mógł wyjść z funkcji `wait` i przystąpić do przetwarzania danych z potoku.

Przykład na listingu 9 pokazuje zakleszczenie w wyniku nieprawidłowości w synchronizacji przy otwieraniu łącza nazwanego.

Listing 9: Możliwość zakleszczenia przy otwieraniu łącza nazwanego

```
#include <fcntl.h>
#define MAX 512
3
main(int argc, char* argv[]) {
    int pdesk;
6
    if (mkfifo("/tmp/fifo", 0600) == -1){
        perror("Tworzenie kolejki FIFO");
9       exit(1);
    }
```

```
12    if (fork() == 0){ // proces potomny
         close(1);
         open("/tmp/fifo", O_WRONLY);
15       execvp("ls", argv);
         perror("Uruchomienie programu ls");
         exit(1);
18    }
      else { // proces macierzysty
         char buf[MAX];
21       int lb, i;

         wait(0);
24       pdesk = open("/tmp/fifo", O_RDONLY);
         while ((lb=read(pdesk, buf, MAX)) > 0){
            for(i=0; i<lb; i++)
27             buf[i] = toupper(buf[i]);
            write(1, buf, lb);
         }
30    }
  }
```

Opis programu: Proces potomny w linii 13 próbuje otworzyć kolejkę FIFO do zapisu. Zostanie on zatem zablokowany do momentu, aż inny proces wywoła funkcję `open` w celu otwarcia kolejki do odczytu. Jeśli jedynym takim procesem jest proces macierzysty (linia 23), to przejdzie on do funkcji `open` dopiero po zakończeniu procesu potomnego, gdyż wcześniej zostanie zablokowany w funkcji `wait`. Proces potomny nie zakończy się, gdyż będzie zablokowany w funkcji `open`, więc będzie blokował proces macierzysty w funkcji `wait`. Proces macierzysty nie umożliwi natomiast potomkowi wyjścia z `open`, gdyż nie może przejść do linii 23. Nastąpi zatem zakleszczenie.

4.4 Zadania

4.1. Jaki będzie wynik wykonania programu na listingu 10?

Listing 10:

```
1 main() {
     int pdesk[2];
     char buf[20];
4
     pipe(pdesk);

7    if (fork() == 0){   // proces potomny
        read(pdesk[0], buf, 20);
        printf("Odczytano z potoku: %s\n", buf);
10      write(pdesk[1], "Hallo od potomka!", 18);
        exit(0);
     }
13   else {              // proces macierzysty
        read(pdesk[0], buf, 20);
        printf("Odczytano z potoku: %s\n", buf);
16      write(pdesk[1], "Hallo od przodka!", 18);
     }
  }
```

4.2. Zrealizować na łączach nienazwanych oraz nazwanych następujące potoki:

a) `finger | cut -d' ' -f1`

b) `ps -ef | grep ^root`

c) `cat /etc/passwd | cut -f5 -d:`

d) `ls -l | grep ^d | more`

e) `ps -ef | tr -s ' ' | cut -f2 -d' '`

f) `finger | tr -s ' '| cut -f2,3 -d' '`

g) `who | cut -f1 -d' ' | sort | uniq -c`

h) `history | cut -c7- | tail -30 | sort | uniq`

i) `ps -ef | tr -s \ : | cut -f1 -d: | sort | uniq -c | sort -n`

4.3. Napisać program do zabijania wszystkich procesów użytkownika, którego nazwa podana jako argument linii poleceń.

4.4. Napisać program do obliczania łącznej liczby znaków we wszystkich nazwach plików w katalogu podanym jako argument linii poleceń.

4.5 Pytania kontrolne

1. Ile deskryptorów tworzy funkcja systemowa `pipe`?
2. Czym się różni łącze nienazwane od nazwanego? Na czym polega różnica w sposobie użycia jednego i drugiego rodzaju łącza?
3. Jakie ograniczenia na możliwość komunikacji pomiędzy procesami wprowadza potok (łącze nienazwane)?
4. W czym są podobne, a czym się różnią od siebie funkcje `creat` i `mkfifo`?
5. Co się stanie przy próbie odczytu danych z pustego łącza?
6. Czym się różni sposób dostępu do plików zwykłych od dostępu do łączy komunikacyjnych?
7. Jaka operacja dostępu do pliku (zwykłego lub specjalnego) i w jaki sposób informuje o osiągnięciu końca pliku?
8. Jakie funkcje systemowe zwracają deskryptory plików?

5 Mechanizmy IPC

Mechanizmy IPC (ang. Interprocess Communication) obejmują pamięć współdzieloną, semafory i kolejki komunikatów. Semafory są raczej mechanizmem synchronizacji, niż komunikacji procesów. Ponieważ dostęp współbieżnych procesów do pamięci współdzielonej wymaga najczęściej odpowiedniej synchronizacji, w sekcji 5.3 zaprezentowane jest łączne użycie semaforów razem z pamięcią współdzieloną do rozwiązania problemu producenta i konsumenta z ograniczonym buforem cyklicznym.

Wszystkie funkcje dotyczące mechanizmów IPC mają przedrostek zależny od rodzaju mechanizmu. Dla pamięci współdzielonej jest to *shm*, dla semaforów jest to *sem*, a dla kolejek komunikatów — *msg*. Niezależnie od rodzaju mechniazmu można wyodrębnić pewne ich cechy wspólne. Obiekt reprezentujący mechanizam z grupy IPC tworzony jest funkcją typu *get*, czyli funkcją o nazwie „get" poprzedzonej przedrostkiem właściwym dla danego typu mechanizmu. Podobnie, pewne operacje sterujące i kontrolne można wykonać funkcją typu *ctl*. Właściwe operacje umożliwiające komunikację lub synchronizację pomiędzy procesami zależne są od rodzaju mechanizmu. Funkcje te zebrane zostały w tab. 5.1.

	tworzenie *get*	operacje *op*	kontrola *ctl*
pamięć współdzielona	`shmget`	`shmat` `shmdt`	`shmctl`
semafory	`semget`	`semop`	`semctl`
kolejki komunikatów	`msgget`	`msgsnd` `msgrcv`	`msgctl`

Tablica 5.1: Funkcje systemowe mechanizmów IPC

Pierwszym argumentem funkcji *get* jest *klucz*, który jest wartością całkowitą typu `key_t`. Klucz jest odnośnikiem do konkretnego obiektu w ramach danego rodzaju mechanizmów i jest odwzorowywany na identyfikator, który jest wartością zwrotną funkcji *get* (jest to wartość typu `int`). Za pomocą klucza różne procesy mogą odnosić się do tego samego obiektu (uzyskać jego identyfikator), co jest warunkiem wykorzystania obiektu do komunikacji lub synchronizacji pomiędzy tymi procesami. Każdy rodzaj mechanizmu IPC ma osobną przestrzeń wartości kluczy. Użycie zatem tej samej wartości klucza dla semafora i dla segmentu pamięci współdzielonej nie powoduje konfliktów. Jeśli nie ma potrzeby przekazywania innym procesom klucza, gdyż identyfikator danego obiektu uzyskiwany jest w inny sposób (np. dziedziczony jest przez potomka od procesu macierzystego), zamiast konkretnej wartości klucza do funkcji *get* można przekazać stałą `IPC_PRIVATE`. Ostatni parametr funkcji *get* — *flg* — określa prawa dostępu do nowo tworzonego obiektu reprezentującego mechanizm IPC. W celu utworzenia obiektu do praw dostępu należy dołożyć poprzez sumę bitową flagę `IPC_CREAT` (tzn. `IPC_CREAT|0640`). Jeśli obiekt o podanym konkretnym kluczu (innym niż `IPC_PRIVATE`) już istnieje, zwrócony zostanie jego identyfikator i nowy obiekt nie jest tworzony. Jeśli dodatkowo ustawiona zostanie flaga `IPC_EXCL` (tzn. `IPC_CREAT|IPC_EXCL|0640`), to w przypadku istnienia obiektu o takim kluczu zwrócona zostanie wartość –1.

Identyfikator zwrócony przez funkcję *get* jest pierwszym argumentem pozostałych funkcji do obsługi mechanizmów IPC. Pozostałe parametry tych funkcji zależne są od konkretnego mechanizmu. W podobny, niezależny od rodzaju mechanizmu, sposób można jednak usunąć istniejący obiekt IPC oraz pobrać lub ustawić atrybuty takiego obiektu. W tym celu wywołuję się odpowiednią funkcję typu *ctl*, przekazując odpowiednio stałą `IPC_RMID`, `IPC_STAT` lub `IPC_SET` jako wartość parametru *cmd*.

5.1 Pamięć współdzielona

Segment pamięci współdzielonej tworzony jest przez wywołanie funkcji systemowej `shmget`. Specyficznym parametrem tej funkcji jest rozmiar segmentu współdzielonej pamięci.

`int shmget(key_t` *key*`, int` *size*`, int` *shmflg*`)` — utworzenie segmentu pamięci współdzielonej. Funkcja zwraca identyfikator segmentu pamięci współdzielonej w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

key klucz,
size rozmiar obszaru współdzielonej pamięci w bajtach,
shmflg flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`).

W celu udostępnienia segmentu pamięci współdzielonej w procesie, należy go włączyć w przestrzeń adresową danego procesu za pomocą funkcji `shmat`. Segment taki można odłączyć za pomocą funkcji `shmdt`. Po włączeniu segmentu proces uzyskuje adres początku współdzielonego obszaru pamięci w swojej przestrzeni adresowej.

`void *shmat(int` *shmid*`, const void *`*shmaddr*`, int` *shmflg*`)` — włączenie segmentu współdzielonej pamięci w przestrzeń adresową procesu. Funkcja zwraca adres segmentu w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

shmid identyfikator obszaru współdzielonej pamięci, zwrócony przez funkcję `shmget`,
shmaddr adres w przestrzeni adresowej procesu, pod którym ma być dostępny segment współdzielonej pamięci (wartość `NULL` oznacza wybór adresu przez system),
shmflg flagi, specyfikujące sposób przyłączenia (np. `SHM_RDONLY` — przyłączenie tylko do odczytu).

`int shmdt(const void *`*shmaddr*`)` — wyłączenie segmentu z przestrzeni adresowej procesu. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

shmaddr adres początku obszaru współdzielonej pamięci w przestrzeni adresowej procesu.

Funkcja systemowa `shmctl` umożliwia przeprowadzanie operacji kontrolnych, np. pobranie atrybutów segmentu współdzielonej pamięci lub jego usunięcie.

`int shmctl(int` *shmid*`, int` *cmd*`, struct shmid_ds *`*buf*`)` — wykonanie operacji kontrolnych na segmencie pamięci współdzielonej. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

shmid identyfikator obszaru współdzielonej pamięci, zwrócony przez funkcję `shmget`,
cmd specyfikacja wykonywanej operacji kontrolnej (np. `IPC_STAT`, `IPC_SET`, `IPC_RMID`),
shmid_ds wskaźnik na strukturę opisującą atrybuty segmentu współdzielonej pamięci (np. właściciel, prawa dostępu).

Listing 1 przedstawia program, w którym następuje cykliczny zapis bufora umieszczonego we współdzielonym obszarze pamięci. Listing 2 przedstawia program, w którym jest analogiczny odczyt bufora cyklicznego.

Listing 1: Zapis bufora cyklicznego

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

#define MAX 10

main(){
   int shmid, i;
   int *buf;

   shmid = shmget(45281, MAX*sizeof(int), IPC_CREAT|0600);
```

```
12    if (shmid == -1){
         perror("Utworzenie segmentu pamieci wspoldzielonej");
         exit(1);
15    }

      buf = (int*)shmat(shmid, NULL, 0);
18    if (buf == NULL){
         perror("Przylaczenie segmentu pamieci wspoldzielonej");
         exit(1);
21    }

      for (i=0; i<10000; i++)
24       buf[i%MAX] = i;
  }
```

Opis programu: W linii 11 tworzony jest segment współdzielonej pamięci o kluczu `45281`, o rozmiarze `MAX*sizeof(int)` i prawach do zapisu i odczytu przez właściciela. Jeśli obszar o takim kluczu już istnieje, zwracany jest jego identyfikator, czyli nie jest tworzony nowy obszar i tym samym rozmiar podany w drugim parametrze oraz prawa dostępu są ignorowane. W linii 17 utworzony segment włączony zostaje do segmentów danego procesu i zwracany jest adres tego segmentu. Zwrócony adres podstawiany jest pod zmienną `buf`. `buf` jest zatem adresem tablicy o rozmiarze `MAX` i typie składowym `int`. Pętla w liniach 23–24 oznacza cykliczny zapis tego bufora, tzn. indeks pozycji, na której zapisujemy jest równy `i%MAX`, czyli zmienia się cyklicznie od 0 do `MAX-1`.

Listing 2: Odczyt bufora cyklicznego

```
1 #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/shm.h>
4
  #define MAX 10

7 main(){
     int shmid, i;
     int *buf;
10
     shmid = shmget(45281, MAX*sizeof(int), IPC_CREAT|0600);
     if (shmid == -1){
13       perror("Utworzenie segmentu pamieci wspoldzielonej");
         exit(1);
     }
16
     buf = (int*)shmat(shmid, NULL, 0);
     if (buf == NULL){
19       perror("Przylaczenie segmentu pamieci wspoldzielonej");
         exit(1);
     }
22
     for (i=0; i<10000; i++)
        printf("Numer: %5d   Wartosc: %5d\n", i, buf[i%MAX]);
25 }
```

Opis programu: Powyższy program jest analogiczny, jak program na listingu 1, przy czym w pętli w liniach 23–24 następuje cykliczny odczyt, czyli odczyt z pozycji w buforze, zmieniającej się cyklicznie od 0 do `MAX-1`.

5.2 Semafory

Funkcja `semget` umożliwia utworzenie zbioru semaforów, przy czym operacje semaforowe można wykonywać niezależnie na każdym semaforze z utworzonego zbioru lub w sposób atomowy na kilku (w szczególności na wszystkich) semaforach w zbiorze. Specyficznym parametrem funkcji `semget` jest liczba semaforów w tworzonym zbiorze.

`int semget(key_t` *key*`, int` *nsems*`, int` *semflg*`)` — utworzenie zbioru (tablicy) semaforów. Funkcja zwraca identyfikator zbioru semaforów w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

key klucz,
nsems liczba semaforów w tworzonym zbiorze,
semflg flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`).

Wykonanie operacji semaforowej polega na wywołaniu funkcji systemowej `semop`. Same operacje wyspecyfikowane są w tablicy odpowiednich struktur. Każdy element tej tablicy definiuje operację na jednym semaforze z danego zbioru. Operacja rozumiana jest jako próba zmodyfikowania zmiennej semaforowej przez dodanie do niej podanej w odpowiednim polu struktury wartości dodatniej lub ujemnej. Wartość dodatnia oznacza podnoszenie semafora, a wartość ujemna oznacza jego opuszczanie. Próba opuszczenia semafora może oznaczać zablokowanie procesu, jeśli wartość zmiennej semaforowej po operacji miałaby być ujemna. Specjalne znaczenie ma podanie 0, jako wartości modyfikującej. Oznacza ona zablokowanie procesu w oczekiwaniu na osiągnięcie przez zmienną semaforową wartości 0.

`int semop(int` *semid*`, struct sembuf *`*sops*`, unsigned` *nsops*`)` — wykonanie operacji semaforowej. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

semid identyfikator zbioru semaforów, zwrócony przez funkcję `semget`,
sops adres tablicy struktur, w której każdy element opisuje operację na jednym semaforze w zbiorze, według następującej definicji:

```
struct sembuf {
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

Znaczenie poszczególnych pól struktury `sembuf` jest następujące:

sem_num numer semafora, na którym ma być wykonana operacja, w zbiorze
sem_op wartość, która ma zostać dodana do zmiennej semaforowej,
sem_flg flagi operacji (`IPC_NOWAIT` — wykonanie bez blokowania, `SEM_UNDO` — wycofanie operacji w przypadku zakończenia procesu).

nsops rozmiar tablicy adresowanej przez *sops* (liczba elementów o strukturze `sembuf`).

Funkcja `semctl` umożliwia wykonanie podobnych operacji na obiekcie, reprezentującym zbiór semaforów, jak w przypadku segmentu pamięci współdzielonej. Dodatkowo, funkcja ta umożliwia ustawienie konkretnej wartości określonej zmiennej ze zbioru semaforów lub wszystkich zmiennych z tego zbioru oraz odczytanie wartości zmiennej semaforowej. Możliwość ustawienie wartości zmiennej semaforowej jest szczególnie istotna w stanie początkowym przetwarzania w celu odpowiedniej inicjalizacji mechanizmów synchronizacji.

`int semctl(int` *semid*`, int` *semnum*`, int` *cmd*`, ...)` — wykonanie operacji kontrolnych na semaforze lub zbiorze semaforów. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

semid	identyfikator zbioru semaforów, zwrócony przez funkcję `semget`,
semnum	numer semafora, na którym ma być wykonana operacja, w identyfikowanym zbiorze,
cmd	specyfikacja wykonywanej operacji kontrolnej (np. `IPC_STAT`, `IPC_SET`, `IPC_RMID`, `SETVAL`, `GETVAL`, `SETALL` itp.).

Listing 3 prezentuje 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).

Listing 3: Realizacja semafora ogólnego

```
#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);
   }
}
```

Opis programu: W celu wykonania operacji semaforowej konieczne jest przygotowanie zmiennej o odpowiedniej strukturze. Ponieważ opisywane operacje wykonywane są tylko na jednym elemencie tablicy semaforów, zmienna ta jest typu pojedynczej struktury (linia 5), składającej się z trzech pól. Poprzedzenie deklaracji zmiennej słowem `static` oznacza, że zmienna ta będzie widoczna tylko wewnątrz pliku, w którym znajduje się jej deklaracja. Poszczególne pola zmiennej `buf` są wypełniane wartościami stosownymi do przekazanych parametrów i rodzaju operacji na semaforze (linie 8–10 i 18–20). W przypadku operacji podnoszenia semafora w pole `sem_op` podstawiane jest wartość `+1` (linia 9), w przypadku operacji opuszczania `-1` (linia 19). O taką liczbę ma zmienić się wartość zmiennej semaforowej.

5.3 Problem producenta i konsumenta z wykorzystaniem semaforów i pamięci współdzielonej

Listingi 4 i 5 przedstawiają rozwiązanie problemu producenta i konsumenta (odpowiednio program producenta i program konsumenta) przy założeniu, że istnieje jeden proces producenta i jeden proces konsumenta, które przekazują sobie dane (wyprodukowane elementy) przez bufor w pamięci współdzielonej i synchronizują się przez semafory. W prezentowanym rozwiązaniu zakłada się, że producent uruchamiany jest przed konsumentem, w związku z czym w programie producenta (listing 4) tworzone i inicjalizowane są semafory oraz segment pamięci współdzielonej.

Listing 4: Synchronizacja producenta w dostępie do bufora cyklicznego

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define MAX 10

main(){
    int shmid, semid, i;
    int *buf;

    semid = semget(45281, 2, IPC_CREAT|0600);
    if (semid == -1){
       perror("Utworzenie tablicy semaforow");
       exit(1);
    }
    if (semctl(semid, 0, SETVAL, (int)MAX) == -1){
       perror("Nadanie wartosci semaforowi 0");
       exit(1);
    }
    if (semctl(semid, 1, SETVAL, (int)0) == -1){
       perror("Nadanie wartosci semaforowi 1");
       exit(1);
    }

    shmid = shmget(45281, MAX*sizeof(int), IPC_CREAT|0600);
    if (shmid == -1){
       perror("Utworzenie segmentu pamieci wspoldzielonej");
       exit(1);
    }

    buf = (int*)shmat(shmid, NULL, 0);
    if (buf == NULL){
       perror("Przylaczenie segmentu pamieci wspoldzielonej");
       exit(1);
    }

    for (i=0; i<10000; i++){
       opusc(semid, 0);
       buf[i%MAX] = i;
       podnies(semid, 1);
    }
}
```

Opis programu: W linii 12 tworzona jest tablica semaforów o rozmiarze 2 (obejmująca 2 semafory). W liniach 17 i 21 semaforom tym nadawane są wartości początkowe. Semafor nr 0 otrzymuje wartość początkową `MAX` (linia 17), semafor nr 1 otrzymuje 0 (linia 21). W liniach 26–36 tworzony jest obszar pamięci współdzielonej, podobnie jak w przykładach w sekcji 5.1. Zapis bufora cyklicznego (linia 40) poprzedzony jest wykonaniem operacji opuszczenia semafora nr 0 (linia 39). Semafor ten ma początkową wartość `MAX`, zatem można wykonać MAX operacji zapisu, czyli zapełnić całkowicie bufor, którego rozmiar jest równy MAX. Semafor osiągnie tym samym wartość 0 i przy kolejnym obrocie pętli nastąpi zablokowanie procesu w operacji opuszczania tego semafora, aż do momentu podniesienia go przez inny proces. Będzie to proces konsumenta (odczytujący), a operacja wykonana będzie po odczytaniu (linia 33 na listingu 5). Wartość semafora nr 0 określa więc liczbę wolnych pozycji w buforze. Po każdym wykonaniu operacji zapisu podnoszony jest semafor nr 1. Jego wartość odzwierciedla zatem poziom zapełnienie bufora i początkowo jest

równa 0 (bufor jest pusty). Semafor nr 1 blokuje konsumenta przed dostępem do pustego bufora (linia 31 na listingu 5).

Listing 5: Synchronizacja konsumenta w dostępie do bufora cyklicznego

```
1 #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/shm.h>
4 #include <sys/sem.h>

  #define MAX 10
7
  main(){
     int shmid, semid, i;
10   int *buf;

     semid = semget(45281, 2, 0600);
13   if (semid == -1){
        perror("Uzyskanie identyfikatora tablicy semaforow");
        exit(1);
16   }

     shmid = shmget(45281, MAX*sizeof(int), 0600);
19   if (shmid == -1){
        perror("Uzyskanie identyfikatora segmentu pamieci
           wspoldzielonej");
        exit(1);
22   }

     buf = (int*)shmat(shmid, NULL, 0);
25   if (buf == NULL){
        perror("Przylaczenie segmentu pamieci wspoldzielonej");
        exit(1);
28   }

     for (i=0; i<10000; i++){
31      opusc(semid, 1);
        printf("Numer: %5d   Wartosc: %5d\n", i, buf[i%MAX]);
        podnies(semid, 0);
34   }
  }
```

Opis programu: W programie konsumenta nie jest tworzony segment pamięci współdzielonej ani tablica semaforów. Są tylko pobierane ich identyfikatory (linia 12 i 18). Konsument może pobrać jakiś element, jeśli bufor nie jest pusty. Przed dostępem do pustego bufora chroni semafor nr 1. Konsument nie może go opuścić, jeśli ma on wartość 0. Semafor ten zwiększany jest po umieszczeniu kolejnego elementu w buforze przez producenta (linia 41 na listingu 4).

Listingi 6 i 7 prezentuje rozwiązanie problemu producentów i konsumentów, czyli dopuszczają istnienie jednocześnie wielu producentów i wielu konsumentów. W tym programie założono, że proces, który pierwszy utworzy tablicę semaforów, dokona inicjalizacji odpowiednich struktur.

Listing 6: Synchronizacja wielu producentów w dostępie do bufora cyklicznego

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define MAX 10

main(){
   int shmid, semid, i;
   int *buf;

   shmid = shmget(45281, (MAX+2)*sizeof(int), IPC_CREAT|0600);
   if (shmid == -1){
      perror("Utworzenie segmentu pamieci wspoldzielonej");
      exit(1);
   }

   buf = (int*)shmat(shmid, NULL, 0);
   if (buf == NULL){
      perror("Przylaczenie segmentu pamieci wspoldzielonej");
      exit(1);
   }

   #define indexZ buf[MAX]
   #define index0 buf[MAX+1]

   semid = semget(45281, 4, IPC_CREAT|IPC_EXCL|0600);
   if (semid == -1){
      semid = semget(45281, 4, 0600);
      if (semid == -1){
         perror("Utworzenie tablicy semaforow");
         exit(1);
      }
   }
   else{
      indexZ = 0;
      index0 = 0;
      if (semctl(semid, 0, SETVAL, (int)MAX) == -1){
         perror("Nadanie wartosci semaforowi 0");
         exit(1);
      }
      if (semctl(semid, 1, SETVAL, (int)0) == -1){
         perror("Nadanie wartosci semaforowi 1");
         exit(1);
      }
      if (semctl(semid, 2, SETVAL, (int)1) == -1){
         perror("Nadanie wartosci semaforowi 2");
         exit(1);
      }
      if (semctl(semid, 3, SETVAL, (int)1) == -1){
         perror("Nadanie wartosci semaforowi 3");
         exit(1);
      }
   }

   for (i=0; i<10000; i++){
      opusc(semid, 0);
      opusc(semid, 2);
      buf[indexZ] = i;
```

```
60          indexZ = (indexZ+1)%MAX;
            podnies(semid, 2);
            podnies(semid, 1);
63      }
    }
```

Opis programu: W linii 27 następuje próba **utworzenia** tablicy semaforów. Flaga `IPC_EXCL` powoduje zwrócenie przez funkcję `semget` wartości –1, gdy tablica o podanym kluczu już istnieje. Zwrócenie przez funkcję wartości –1 oznacza, że tablica już istnieje i pobierany jest tylko jej identyfikator (linia 29). W przeciwnym przypadku (tzn. wówczas, gdy tablica rzeczywiście jest tworzona) proces staje się inicjatorem struktur danych na potrzeby komunikacji, wykonując fragment programu w liniach 36–53.

Do prawidłowej synchronizacji producentów i konsumentów potrzebne są cztery semafory. Dwa z nich (semafory numer 0 i 1 w tablicy) służą do kontroli liczby wolnych i zajętych pozycji w buforze, podobnie jak w przypadku jednego producenta i jednego konsumenta. W przypadku wielu producentów i wielu konsumentów zarówno producenci jak i konsumenci muszą współdzielić indeks pozycji odpowiednio do zapisu i do odczytu. Indeksy te przechowywane są we współdzielonym segmencie pamięci zaraz za buforem. Stąd rozmiar tworzonego segmentu ma wynosi `MAX+2` pozycji typu `int` (linia 12). Indeks kolejnej pozycji do zapisu przechowywany jest pod indeksem `MAX` we współdzielonej tablicy (linia 24), a indeks kolejnej pozycji do odczytu przechowywany jest pod indeksem `MAX+1` w tej tablicy (linia 25). Pozycje 0 – MAX-1 stanowią bufor do komunikacji. Pozostałe dwa (semafory numer 2 i 3) służą zatem do zapewnienia wzajemnego wykluczania w dostępie do współdzielonych indeksów pozycji do zapisu i do odczytu. Poszczególne semafory są inicjalizowane odpowiednimi wartościami w liniach 38, 42, 46 i 50. W liniach 36 i 37 inicjalizowane są wartościami zerowymi współdzielone indeksy pozycji do zapisu i od odczytu.

Działanie producentów polega na uzyskaniu wolnej pozycji w buforze dzięki opuszczeniu semafora nr 0, podobnie jak w poprzednim przykładzie, oraz uzyskaniu wyłączności dostępu do indeksu pozycji do zapisu. Wyłączność dostępu do indeksu jest konieczna, gdyż w przeciwnym razie dwaj producenci mogliby jednocześnie modyfikować ten indeks, w efekcie czego uzyskaliby tę samą wartość i próbowaliby umieścić „wyprodukowane" elementy na tej samej pozycji w buforze. Wyłączność dostępu do indeksu uzyskiwana jest przez opuszczenie semafora nr 2 (linia 58), którego wartość początkowa jest 1. Po opuszczeniu przyjmuje on wartość 0, co uniemożliwia opuszczenie go przez inny proces do momentu podniesienia w linii 61. Przed dopuszczeniem innego procesu do możliwości aktualizacji indeksu następuje umieszczenie „wyprodukowanego" elementu w buforze (linia 59) oraz aktualizacja indeksu do zapisu tak, żeby wskazywał on na następną pozycję w buforze (linia 60). W linii 62 następuje podniesienie semafora nr 1, wskazującego na liczbę elementów do „skonsumowania" w buforze.

Listing 7: Synchronizacja wielu konsumentów w dostępie do bufora cyklicznego

```
1  #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/shm.h>
4  #include <sys/sem.h>

   #define MAX 10
7
   main(){
      int shmid, semid, i;
10    int *buf;

      shmid = shmget(45281, (MAX+2)*sizeof(int), IPC_CREAT|0600);
13    if (shmid == -1){
```

```
        perror("Utworzenie segmentu pamieci wspoldzielonej");
        exit(1);
16    }

      buf = (int*)shmat(shmid, NULL, 0);
19    if (buf == NULL){
        perror("Przylaczenie segmentu pamieci wspoldzielonej");
        exit(1);
22    }

      #define indexZ buf[MAX]
25    #define indexO buf[MAX+1]

      semid = semget(45281, 4, IPC_CREAT|IPC_EXCL|0600);
28    if (semid == -1){
        semid = semget(45281, 4, 0600);
        if (semid == -1){
31        perror("Utworzenie tablicy semaforow");
          exit(1);
        }
34    }
      else{
        indexZ = 0;
37      indexO = 0;
        if (semctl(semid, 0, SETVAL, (int)MAX) == -1){
          perror("Nadanie wartosci semaforowi 0");
40        exit(1);
        }
        if (semctl(semid, 1, SETVAL, (int)0) == -1){
43        perror("Nadanie wartosci semaforowi 1");
          exit(1);
        }
46      if (semctl(semid, 2, SETVAL, (int)1) == -1){
          perror("Nadanie wartosci semaforowi 2");
          exit(1);
49      }
        if (semctl(semid, 3, SETVAL, (int)1) == -1){
          perror("Nadanie wartosci semaforowi 3");
52        exit(1);
        }
      }
55
      for (i=0; i<10000; i++){
        opusc(semid, 1);
58      opusc(semid, 3);
        printf("Numer: %5d   Wartosc: %5d\n", i, buf[indexO]);
        indexO = (indexO+1)%MAX;
61      podnies(semid, 3);
        podnies(semid, 0);
      }
64  }
```

Opis programu: Program konsumenta w liniach 1–56 jest identyczny, jak program producenta. W pozostałych liniach jest on „symetryczny” w tym sensie, że opuszczany jest semafor nr 1, kontrolujący liczbę zajętych pozycji (elementów do „skonsumowania”, linia 57), a po pobraniu elementu podnoszony jest semafor nr 0, kontrolujący liczbę wolnych pozycji (linia 62). Wzajemne wykluczanie w dostępie do współdzielonego indeksu do zapisu zapewnia semafor nr 3, który jest opuszczany w linii 58, a podnoszony w linii 61.

5.4 Kolejki komunikatów

Listingi 4 i 5 przedstawiają rozwiązanie problemu producenta i konsumenta (odpowiednio program producenta i program konsumenta) na kolejce komunikatów. W rozwiązaniu zakłada się ograniczone buforowanie, tzn. nie może być więcej nie skonsumowanych elementów, niż pewna założona ilość (pojemność bufora). Rozwiązanie dopuszcza możliwość istnienia wielu producentów i wielu konsumentów.

Listing 8: Impelmentacja zapisu ograniczonego bufora za pomocą kolejki komunikatów

```
1  #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/msg.h>
4
   #define MAX 10

7  struct buf_elem {
       long mtype;
       int mvalue;
10 };
   #define PUSTY 1
   #define PELNY 2
13
   main(){
       int msgid, i;
16     struct buf_elem elem;

       msgid = msgget(45281, IPC_CREAT|IPC_EXCL|0600);
19     if (msgid == -1){
          msgid = msgget(45281, IPC_CREAT|0600);
          if (msgid == -1){
22           perror("Utworzenie kolejki komunikatów");
             exit(1);
          }
25     }
       else{
          elem.mtype = PUSTY;
28        for (i=0; i<MAX; i++)
             if (msgsnd(msgid, &elem, sizeof(elem.mvalue), 0) == -1){
                perror("Wyslanie pustego komunikatu");
31              exit(1);
             }
       }
34
       for (i=0; i<10000; i++){
          if (msgrcv(msgid, &elem, sizeof(elem.mvalue), PUSTY, 0) == -1){
37           perror("Odebranie pustego komunikatu");
             exit(1);
          }
40        elem.mvalue = i;
          elem.mtype = PELNY;
          if (msgsnd(msgid, &elem, sizeof(elem.mvalue), 0) == -1){
43           perror("Wyslanie elementu");
             exit(1);
          }
46     }
   }
```

Opis programu: Podobnie jak w programach na lisingach 6 i 7 w linii 18 jest próba utworzenia kolejki komunikatów. Jeśli kolejka już istnieje, funkcja `msgget` zwróci wartość –1 i nastąpi

pobranie identyfikatora już istniejącej kolejki (linia 20). Jeśli kolejka nie istnieje, zostanie ona utworzona w linii 18 i nastąpi wykonanie fragmentu programu w liniach 27–32, w wyniku czego w kolejce zostanie umieszczonych `MAX` komunikatów typu `PUSTY`. Umieszczenie elementu w buforze, reprezentowanym przez kolejkę komunikatów, polega na zastąpieniu komunikatu typu `PUSTY` komunikatem typu `PELNY`. Pusty komunikat jest pobierany w linii 36. Brak pustych komunikatów oznacza całkowite zapełnienie bufora i powoduje zablokowanie procesu w funkcji `msgrcv`. Po odebraniu pustego komunikatu przygotowywany jest komunikat pełny (linie 40–41) i umieszczany jest w buforze, czyli wysyłany do kolejki (linia 42). Podobnie jak suma wartości zmiennych semaforowych do kontroli zapełnienia bufora w przykładach 4–5 i 6–7 jest równa `MAX` liczba komunikatów jest również równa `MAX`, z wyjątkiem momentu, gdy jakiś proces wykonuje operację przekazania lub pobrania elementu.

Listing 9: Impelmentacja odczytu ograniczonego bufora za pomocą kolejki komunikatów

```
1  #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/msg.h>
4
   #define MAX 10

7  struct buf_elem {
       long mtype;
       int mvalue;
10 };
   #define PUSTY 1
   #define PELNY 2
13
   main(){
       int msgid, i;
16     struct buf_elem elem;

       msgid = msgget(45281, IPC_CREAT|IPC_EXCL|0600);
19     if (msgid == -1){
           msgid = msgget(45281, IPC_CREAT|0600);
           if (msgid == -1){
22             perror("Utworzenie kolejki komunikatów");
               exit(1);
           }
25     }
       else{
           elem.mtype = PUSTY;
28         for (i=0; i<MAX; i++)
               if (msgsnd(msgid, &elem, sizeof(elem.mvalue), 0) == -1){
                   perror("Wyslanie pustego komunikatu");
31                 exit(1);
               }
       }
34
       for (i=0; i<10000; i++){
           if (msgrcv(msgid, &elem, sizeof(elem.mvalue), PELNY, 0) == -1){
37             perror("Odebranie elementu");
               exit(1);
           }
40         printf("Numer: %5d   Wartosc: %5d\n", i, elem.mvalue);
           elem.mtype = PUSTY;
           if (msgsnd(msgid, &elem, sizeof(elem.mvalue), 0) == -1){
43             perror("Wyslanie pustego komunikatu");
               exit(1);
```

```
        }
    }
}
```

5.5 Zadania

5.1. Listingi 10 i 11 przedstawiają zmodyfikowane fragmenty (zgodnie z numeracją linii) programów odpowiednio z listingów 6 (str. 44) i 7 (str 45). Wykaż, że w zmodyfikowanej wersji może dojść do zakleszczenia procesów. Uwaga: w niniejszej wersji semafor nr 3 nie jest używany.

Listing 10: Modyfikacja programu 6

```
for (i=0; i<10000; i++){
    opusc(semid, 2);
    opusc(semid, 0);
    buf[indexZ] = i;
    indexZ = (indexZ+1)%MAX;
    podnies(semid, 1);
    podnies(semid, 2);
}
```

Listing 11: Modyfikacja programu 7

```
for (i=0; i<10000; i++){
    opusc(semid, 2);
    opusc(semid, 1);
    printf("Numer: %5d   Wartosc: %5d\n", i, buf[indexO]);
    indexO = (indexO+1)%MAX;
    podnies(semid, 0);
    podnies(semid, 2);
}
```

5.2. Listingi 12 i 13 przedstawiają zmodyfikowane fragmenty (zgodnie z numeracją linii) programów odpowiednio z listingów 6 i 7. Wskaż, na czym polega błąd w zmodyfikowanej wersji.

Listing 12: Modyfikacja programu 6

```
for (i=0; i<10000; i++){
    int index;

    opusc(semid, 0);
    opusc(semid, 2);
    index = indexZ;
    indexZ = (indexZ+1)%MAX;
    podnies(semid, 2);
    buf[index] = i;
    podnies(semid, 1);
}
```

Listing 13: Modyfikacja programu 7

```
for (i=0; i<10000; i++){
    int index;

    opusc(semid, 1);
    opusc(semid, 3);
    index = indexZ;
    indexO = (indexO+1)%MAX;
```

```
        podnies(semid, 3);
9       printf("Numer: %5d   Wartosc: %5d\n", i, buf[index]);
        podnies(semid, 0);
    }
```

5.6 Pytania kontrolne

1. Co jest wartością zwrotną funkcji typu *get*?
2. Jakie są podobieństwa i różnice w semantyce parametrów funkcji `shmget` i `semget`?
3. Do czego służy i co zwraca funkcja systemowa `shmat`?
4. Jaka byłaby prawidłowa kolejność wykonania następujących operacji w celu skorzystania z pamięci współdzielonej:
   (a) wywołanie funkcji `shmat`,
   (b) wywołanie funkcji `shmdt`,
   (c) wywołanie funkcji `shmget`
   (d) zapis lub odczyt danych w segmencie pamięci współdzielonej?
5. W jaki sposób z punktu widzenia programisty wykonuję się operację semaforową?
6. Kiedy może nastąpić zablokowanie procesu w wykonaniu operacji semaforowej?
7. W jaki sposób może nastąpić odblokowanie procesu zablokowanego w trakcie wykonywania operacji semaforowej?
8. W jaki sposób można ustawić określoną wartość zmiennej semaforowej?
9. Na czym polega synchronizacja producenta i konsumenta w dostępie do wspólnego bufora?
10. Które funkcje systemowe i w jakich warunkach mogą spowodować zablokowanie procesu?

6 Sygnały

Sygnał wysyłany jest do procesu w celu zakomunikowania o fakcie wystąpienia pewnego zdarzenia istotnego dla tego procesu. Sygnał może być wysłany przez jakiś proces w systemie (w szczególności proces może wysłać sygnał sam do siebie) lub przez jądro systemu operacyjnego. Możliwość przesłanie sygnału przez proces do innego procesu uwarunkowana jest zgodnością odpowiednich identyfikatorów. W przypadku zwykłych użytkowników obowiązujący identyfikator użytkownika procesu wysyłającego musi być taki sam, jak rzeczywisty identyfikator użytkownika procesu, do którego adresowany jest sygnał. Procesy użytkownika uprzywilejowanego `root` mogą wysyłać sygnały do wszystkich procesów w systemie.

6.1 Rodzaje sygnałów

Procesy mogą otrzymywać różne rodzaje sygnałów w zależności od zdarzenia w systemie, które spowodowało konieczność wysłania takiego sygnału. Na każdy rodzaj sygnału proces może inaczej reagować. Inna może być też reakcja domyślna.

Tablica 6.1: Lista sygnałów w standardzie POSIX.1

nazwa	numer	reakcja domyślna	opis
SIGHUP	1	zakończenie procesu	zawieszenie terminala
SIGINT	2	zakończenie procesu	naciśnięcie klawisza przerwania
SIGQUIT	3	zakończenie procesu i zrzut obrazu pamięci	naciśniecie klawisza zakończenia
SIGILL	4	zakończenie procesu i zrzut obrazu pamięci	nieprawidłowa instrukcja
SIGABRT	6	zakończenie procesu i zrzut obrazu pamięci	wywołanie funkcji `abort`
SIGFPE	8	zakończenie procesu i zrzut obrazu pamięci	wyjątek zeminnopozycyjny
SIGKILL	9	zakończenie procesu	sygnał zabicia procesu
SIGSEGV	11	zakończenie procesu i zrzut obrazu pamięci	nieprowidłowe odwołanie do pamięci
SIGPIPE	13	zakończenie procesu	przerwanie potoku
SIGALRM	14	zakończenie procesu	sygnał po wywołaniu funkcji `alarm`
SIGTERM	15	zakończenie procesu	sygnał zakończenia
SIGUSR1	30, 10, 16	zakończenie procesu	sygnał 1 użytkownika
SIGUSR2	31, 12, 17	zakończenie procesu	sygnał 2 użytkownika
SIGCHLD	20, 17, 18	ignorowanie sygnału	zatrzymanie lub zakończenie potomka
SIGCONT	19, 18, 25		kontynuacja po zatrzymaniu
SIGSTOP	17, 19, 23	zatrzymanie procesu	zatrzymanie (zawieszenie) procesu
SIGTSTP	18, 20, 24	zatrzymanie procesu	sygnał zatrzymiania z terminala
SIGTTIN	21, 21, 26	zatrzymanie procesu	odczyt z terminala przez proces w tle
SIGTTOU	22, 22, 27	zatrzymanie procesu	zapis na terminalu przez proces w tle

6.2 Wysyłanie sygnałów

Podstawową funkcją do przesyłania sygnałów do procesów jest `kill`.

`int kill(pid_t` *pid*`, int` *signum*`)` — wysłanie sygnału do procesu. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

pid identyfikator procesu, do którego adresowany jest sygnał,
signum numer przesyłanego sygnału.

W celu wysłania sygnału do samego siebie proces może użyć funkcji systemowej `raise`. Wywołanie tej funkcji jest funkcjonalnie równoważne wywołaniu funkcji `kill` z wynikiem wywołania funkcji `getpid` jako pierwszym argumentem. W sposób szczególny proces może wysłać do samego siebie sygnał `SIGALRM`. Wysłanie tego sygnału jest następstwem wywołania funkcji `alarm`.

`unsigned int alarm(unsigned int sec)` — wysłanie sygnału `SIGALRM` do procesu wywołującego funkcję po podanej liczbie sekund od momentu wywołania funkcji `alarm`. Funkcja zwraca liczbę sekund, jaka pozostała do upłynięcia poprzednio ustawionego alarmu lub 0, jeśli nie było oczekującego alarmu.

Opis parametrów:

sec liczba sekund od momentu wywołania funkcji do momentu wysłania sygnału (w przypadku wartości 0 alarm nie jest ustawiany).

Niezależnie od wartości kasowany jest poprzednio ustawiony alarm.

6.3 Obsługa sygnałów

Rozróżnia się 3 sposoby reakcji procesu na sygnał:

- reakcja domyślna,
- ignorowanie sygnału,
- przechwytywanie sygnału.

Reakcję na sygnał można zdefiniować, korzystając z funkcji `signal` lub `sigaction`.

`sighandler_t signal( int` *signum*`, sighandler_t *`*handler*`)` — zdefiniowanie reakcji procesu na sygnał. Funkcja zwraca wartość, oznaczającą dotychczasową reakcję procesu na sygnał lub `SIG_ERR` w przypadku błędu.

Opis parametrów:

signum numer przesyłanego sygnału,
handler wskaźnik na funkcję do obsługi sygnału lub jedna ze stałych:
`SIG_DFL` — ustawienie reakcji domyślnej,
`SIG_IGN` — ignorowanie sygnału.

Podczas przechwytywania sygnału do funkcji obsługującej przekazywany jest parametr typu `int`, którego wartość rzeczywista jest numerem sygnału. Dzięki temu można użyć tej samej funkcji do obsługi wielu sygnałów, a w ramach jej wykonania rozpoznać numer sygnału i podjąć stosowne działania. Przykład użycia funkcji `signal` pokazano na listingu 1.

Listing 1: Definiowanie reakcji procesu na sygnał

```
#include <signal.h>

3   void f(int sig_num){
       printf("Przechwycenie sygnalu nr %d\n", sig_num);
    }
6
    main(){

9      printf("Domyslna obsluga sygnalu\n");
       signal(SIGINT, SIG_DFL);
```

```
        sleep(5);
12      printf("Ignorowanie sygnalu\n");
        signal(SIGINT, SIG_IGN);
        sleep(5);
15      printf("Przechwytywanie sygnalu\n");
        signal(SIGINT, f);
        sleep(5);
18
    }
```

Opis programu: Kolejne wywołania funkcji `signal` ustawiają domyślna reakcję na sygnał `SIGINT`, ignorowanie sygnału `SIGINT`, a następnie jego przechwytywanie. Przechwycenie sygnału oznacza wywołanie funkcji `f`, która wypisuje komunikat z numerem przechwyconego sygnału.

Większe niż funkcja `signal` możliwości w zakresie obsługi sygnałów udostępnia funkcja `sigaction`.

Jeśli sygnał nie jest ignorowany, reakcja następuje w stanie aktywności procesu, do którego sygnał został przekazany. Aktualny stan przetwarzania może jednak wymagać zaniechania podejmowania akcji związanej z obsługą sygnału lub natychmiastowego zakończenia procesu. Na czas wykonywania fragmentu kodu, wykluczającego możliwość reakcji na sygnał, można by go ignorować. Ignorowanie sygnału uniemożliwia z kolei stwierdzenie faktu, że sygnał został przekazany i tym samym uniemożliwia detekcję zasygnalizowanej sytuacji wyjątkowej. Ponadto, ignorowanie większej liczby sygnałów jest dość uciążliwe dla programisty, gdyż wymaga wielokrotnego wywołania funkcji `signal`.

Uciążliwości i ograniczenia funkcji `signal` w tym zakresie zostały usunięte w mechanizmie blokowania sygnałów. Mechanizm ten pozwala zablokować reakcję na sygnał, przy czym sygnały przekazywane do procesu są rejestrowane w odpowiedniej strukturze w bloku kontrolnym procesu, co umożliwia podjęcie stosownych akcji po odblokowaniu. W stosunku do funkcji `signal` poprawiona została też wygoda posługiwania się mechanizmem blokowania, gdyż określa się zbiór sygnałów reprezentowanych przez zmienną typu `sigset_t`.

Zablokowanie sygnałów umożliwia funkcja `sigprocmask`.

`int sigprocmask( int` *how*`, const sigset_t *`*set*`, const sigset_t *`*oldset*` )` — modyfikacja zbioru blokowanych sygnałów. Funkcja zwraca 0 w przypadku poprawnego zakończeniu lub –1 w przypadku błędu.

Opis parametrów:

how	sposób interpretacji parametru *set* `SIG_BLOCK` — dodanie do zbioru blokowanych sygnałów, `SIG_UNBLOCK` — usunięcie ze zbioru blokowanych sygnałów, `SIG_SETMASK` — zdefiniowanie zbioru blokowanych sygnałów,
set	zbiór sygnałów, które mają zostać zablokowane lub odblokowane (w zależności od parametru *set*),
oldset	jeśli wskaźnik nie jest pusty, to wskazuje na zbiór sygnałów, w którym umieszczone zostaną sygnały dotychczas blokowane.

Do obsługi zbioru sygnałów zdefiniowane zostały specjalne funkcje: `sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`, `sigismember`.

`int sigemptyset( sigset_t *`*set*` )` — inicjalizacja pustego zbioru sygnałów. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

`int sigfillset( sigset_t *`*set*` )` — inicjalizacja zbioru zawierającego wszystkie możliwe sygnały. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

`int sigaddset( sigset_t *`*set*`, int` *signum*` )` — dodanie sygnału do zbioru. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

`int sigdelset( sigset_t *set, int signum )` — usunięcie sygnału ze zbioru. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

`int sigismember( const sigset_t *set, int signum )` — sprawdzenie przynależności sygnału do zbioru. Funkcja zwraca 1 lub 0, odpowiednio gdy sygnał należy lub nie należy do zbioru. Wartością zwrotną w przypadku błędu jest –1.

Opis parametrów:

set wskazanie zbioru sygnałów,

signum numer sygnału.

Odblokowanie sygnałów (również przy użyciu funkcji `sigprocmask`) oznacza podjęcie obsługi sygnałów zgłoszonych podczas zablokowania, co pokazuje przykład na listingu 2.

Listing 2: Blokowanie sygnałów

```
#include <signal.h>
#include <stdio.h>

   void f(int sig_num){
      printf("Przechwycenie sygnalu nr %d\n", sig_num);
   }

   main(){
      sigset_t zbior;

      sigfillset(&zbior);
      sigprocmask(SIG_BLOCK, &zbior, NULL);
      signal(SIGINT, f);
      printf("Oczekiwanie na sygnaly\n");
      sleep(10);
      sigprocmask(SIG_UNBLOCK, &zbior, NULL);
      printf("Koniec\n");
   }
```

Opis programu: Na początku programu w wyniku wywołania funkcji `sigfillset` oraz `sigprocmask` następuje zablokowanie wszystkich sygnałów, a następnie ustawienie przechwytywania sygnału `SIGINT`. W trakcie 10-sekundowego uśpienia procesu sygnały są rejestrowane, ale reakcja na nie nastąpi dopiero po odblokowanie przez ponowne wywołanie funkcji `sigprocmask`.

Funkcja `sigpending` umożliwia z kolei stwierdzenie, które sygnały zostały zarejestrowane w czasie ich zablokowania.

`int sigpending( sigset_t *set )` — odczytanie zbioru sygnałów, zgłoszonych podczas zablokowania. Funkcja zwraca 0 w przypadku poprawnego zakończeniu lub –1 w przypadku błędu.

Opis parametrów:

set wskaźnik do zmiennej, w której umieszczona zostanie informacja o odebranych sygnałach.

Przykład użycia funkcji `sigpending` pokazano na listingu 3.

Listing 3: Odczyt zgłoszonych sygnałów

```
#include <signal.h>
#include <stdio.h>

   main(){
      sigset_t zbior, zbior2;

```

```
        sigfillset(&zbior);
        sigprocmask(SIG_BLOCK, &zbior, NULL);

        printf("Oczekiwanie na sygnaly\n");
        sleep(10);

        sigpending(&zbior2);
        switch ( sigismember(&zbior2, SIGINT) ){
           case -1:
              perror("sigismember");
              break;
           case  0:
              printf("Nie odebrano sygnalu SIGINT\n");
              break;
           case  1:
              printf("Odebrano sygnal SIGINT\n");
        }
        sigprocmask(SIG_UNBLOCK, &zbior, NULL);
        printf("Koniec\n");
     }
```

Opis programu: W programie po 10 sekundach oczekiwania przy zablokowanych sygnałach następuje sprawdzenie poprzez wywołanie funkcji `sigpending`, jakie sygnały zostały dotychczas odebrane.

W celu anulowania zablokowanego sygnału i tym samym uniknięcia reakcji na ten sygnał po odblokowaniu wystarczy za pomocą funkcji `signal` ustawić ignorowanie sygnału.

Proces można zawiesić w oczekiwaniu na sygnał za pomocą funkcji `sigsuspend`.

`int sigsuspend( const sigset_t *`*maska*` )` — oczekiwanie na sygnał. Funkcja zwraca zawsze –1.

Opis parametrów:

maska wskazanie na zbiór sygnałów, które mają być ignorowane podczas oczekiwania. Wznowienie procesu może więc nastąpić w wyniku otrzymania sygnału nie ujętego we wskazanym zbiorze. Po otrzymaniu sygnału nastąpi normalne reakcja, czyli w zależności od wcześniejszych ustawień: ignorowanie, przechwytywanie lub reakcja domyślna.

Listing 4: Zawieszenie procesu w oczekiwaniu na sygnał

```
#include <signal.h>

   main(){
      sigset_t maska;

      sigemptyset(&maska);
      sigaddset(&maska, SIGINT);
      sigaddset(&maska, SIGTERM);

      printf("Oczekiwanie na sygnal\n");
      sigsuspend(&maska);
   }
```

Opis programu: W wyniku wywołania funkcji `sigsuspend` proces wchodzi w stan oczekiwania na dowolny sygnał w wyjątkiem `SIGINT` i `SIGTERM`, które zostały wcześniej ustawione w masce sygnałów ignorowanych.

Literatura

[1] Gray J. S.: *Komunikacja między procesami w Unixie.* Oficyna Wydawnicza ReadMe, Warszawa, 1998.

[2] Rochkind M. J.: *Programowanie w systemie Unix dla zaawansowanych.* Wydawnictwa Naukowo-Techniczne, Warszawa, wydanie 3, 2007.

[3] Silberschatz A., Galvin P. B., Gagne G.: *Podstawy systemów operacyjnych.* Wydawnictwa Naukowo-Techniczne, Warszawa, wydanie 7, 2006.

[4] Stevens W. R.: *Programowania zastosowań sieciowych w systemie Unix.* Wydawnictwa Naukowo-Techniczne, Warszawa, 1995.

[5] Stevens W. R.: *Programowania w środowisku systemu UNIX.* Wydawnictwa Naukowo-Techniczne, Warszawa, 2002.

[6] Weiss Z., Gruźlewski T.: *Programowanie współbieżne i rozproszone w przykładach i zadaniach.* Wydawnictwa Naukowo-Techniczne, Warszawa, 1993.

Indeks