Systemy operacyjne II

Skrypt do ćwiczeń laboratoryjnych

Cezary Sobaniec

v1.4
2012-02-16

Politechnika Poznańska
Instytut Informatyki

Spis treści

1 Wprowadzenie

1.1 Organizacja zajęć

Aktualna wersja niniejszego dokumentu jest dostępna w postaci elektronicznej pod adresem:

http://www.cs.put.poznan.pl/csobaniec/edu/sop/sop2.pdf

Dodatkowe informacje na temat przedmiotu Systemy operacyjne można znaleźć na stronach:

http://www.cs.put.poznan.pl/sop/

Uwagi ogólne

1. Do przeprowadzenia ćwiczeń potrzebny jest dostęp do dowolnego systemu Unix (np. Linux). Dostęp ten może być realizowany poprzez uruchomienie systemu operacyjnego Unix bezpośrednio na komputerze użytkownika, bądź poprzez dostęp zdalny do serwera dydaktycznego. W drugim przypadku można do tego celu zastosować darmowy program PuTTY. Połączenie z serwerem wymaga podania w polu *Host Name* pełnej nazwy unixlab.cs.put.poznan.pl. Należy ustawić translację znaków na UTF-8 (opcja *Window* ▷ *Translation* ▷ *Character set translation on received data*). Konfigurację można zapisać w zakładce *Session*, wpisując w polu *Saved Sessions* nazwę mars i wciskając przycisk *Save*.
2. Kopiowanie plików do i z serwera dydaktycznego można przeprowadzić z wykorzystaniem darmowego programu WinSCP.
3. Przykłady zamieszczone w skrypcie mogą być kopiowane do edytorów tekstowych. Podczas kopiowania niektóre znaki zostają wklejone w niepoprawny sposób. Dotyczy to m.in. apostrofów i myślników. Należy je po skopiowaniu usunąć i wstawić ponownie wpisując z klawiatury.
4. Do zapisu poleceń wydawanych w środowisku Unix przyjęto następującą konwencję. Znak „#” na początku linii symbolizuje znak zachęty interpretera poleceń. Pogrubiona czcionka oznacza komendy/tekst wprowadzany przez użytkownika. Poniższy przykład:

   ```
   # echo przykład
   przykład
   ```

   oznacza wpisanie komendy „`echo przykład`”, która powoduje wypisanie napisu „przykład” na ekranie.

5. Kod źródłowy programów należy formatować w sposób czytelny poprzez stosowanie wcięć odwzorowujących poziom zagnieżdżenia poszczególnych bloków kodu. Wcięcia powinny mieć od 2 do 8 spacji i powinny być takie same w całym pliku. Przykłady zawarte w skrypcie pokazują jak praktycznie formatować kod źródłowy.
6. *Last but not least*. Przerywanie programów realizowane jest w systemach uniksowych kombinacją Ctrl-c. Wiele osób wykorzystuje w tym celu kombinację Ctrl-z co jest oczywistym błędem i nadużyciem. Kombinacja Ctrl-z służy bowiem do *wstrzymywania* procesów, a nie ich *zatrzymywania* (usuwania z systemu, przerywania, zabijania). Sterowanie co prawda wraca do interpretera poleceń w oknie emulatora terminala, ale wstrzymany proces pozostaje w systemie nadal blokując zajęte przez siebie zasoby (np. pamięć). Wznowienie wstrzymanego procesu umożliwia komenda `fg`, a `jobs` wyświetla listę procesów zarządzanych przez interpreter poleceń . Obrazuje to następująca sekwencja zleceń:

```
# sleep 10
# <Ctrl-Z>
# ls
...
# jobs
[1]+  Stopped                 sleep 10
# fg
```

Proszę więc *nie używać* kombinacji Ctrl-z, chyba że z pełną świadomością tego, co się robi.

Zaliczenie przedmiotu

Podczas zajęć przeprowadzone zostaną 3 testy wielokrotnego wyboru. Każdy test zawiera 10 pytań po 1 pkt., co daje w sumie 30 pkt. Dodatkowo realizowany będzie projekt za 15 pkt. W efekcie możliwe jest uzbieranie 45 pkt. Ocena końcowa wyliczana jest wg. poniższej tabeli:

Liczba punktów	Ocena końcowa
$\geq 22,0$	3.0
$\geq 26,5$	3.5
$\geq 31,0$	4.0
$\geq 35,5$	4.5
$\geq 40,0$	5.0

Przedkładając projekt do oceny należy być przygotowanym do udzielenia wszelkich wyjaśnień dotyczących jego realizacji. Nieznajomość implementacji lub niezrozumienie wykorzystanych technologii są podstawą do niezaliczenia projektu.

1.2 Środowisko pracy

1. Utwórz plik tekstowy zawierający poniższy kod w języku C:

```c
#include <stdio.h>

int main()
{
  printf("Hello world!\n");
}
```

Do edycji pliku użyj edytora `vi`, `mcedit`, `pico`, `jed`, `emacs` lub innego.

2. Skompiluj program:

```
# cc -o hello hello.c
```

3. Uruchom program:

```
# ./hello
Hello world!
```

4. Włącz wyświetlanie wszystkich ostrzeżeń podczas kompilacji:

```
# cc -Wall -o hello hello.c
hello.c: In function 'main':
hello.c:6: warning: control reaches end of non-void function
```

Usunięcie powyższego ostrzeżenia (brak jawnego zwrócenia wartości przez funkcję, która zwraca typ `int`) wymaga dodania na końcu funkcji `main()` instrukcji:

```
return 0;
```

Programy powinny być tak pisane, aby ich kompilacja z przełącznikiem `-Wall` nie powodowała wyświetlania żadnych komunikatów.

1.3 Pomoc systemowa

1. Sprawdź pomoc systemową dla funkcji `printf(3)`:

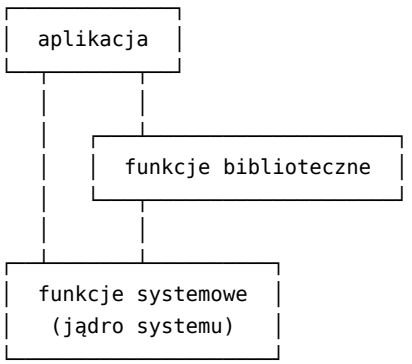
```
# man 3 printf
```

Sekcja SYNOPSIS zawiera informacje o wymaganych plikach nagłówkowych.

2. Uzyskanie pomocy systemowej w języku polskim wymaga ustawienia zmiennej środowiskowej LANG:

```
# export LANG=pl_PL.UTF-8
# man 2 open
```

3. Zagadnienia omawiane prezentowane w tym skrypcie i omawiane na zajęciach udokumentowane są w sekcjach 2 i 3 pomocy systemowej. Sekcja 2 zawiera opisy funkcji systemowych systemu operacyjnego, a sekcja 3 opisy funkcji bibliotecznych. Funkcje biblioteczne nadbudowują nad funkcjami systemowymi dostarczając wygodniejszego interfejsu programisty. Obrazuje to poniższy rysunek:



Przykładem może być funkcja systemowa `write` i funkcja biblioteczna `printf`. Obie mogą posłużyć do wypisania komunikatu na ekranie, ale funkcja `printf` jest bogatsza, bo oferuje m.in. buforowanie oraz formatowanie danych. Wewnętrznie funkcja `printf` do swojej implementacji wykorzystuje funkcję `write`.

Przywołanie pomocy systemowej z określonej sekcji wymaga wskazania numeru sekcji jako pierwszego argumentu komendy `man`:

```
# man 2 open
# man 3 printf
```

Odwołania do stron pomocy systemowej zapisywane są często razem ze wskazaniem na numer sekcji, np. strona printf(3) jest dokumentacją funkcji bibliotecznej `printf` opisywanej w sekcji 3.

1.4 Przekazywanie argumentów

Programy przygotowywane w ramach ćwiczeń będą prostymi przykładami zastosowań różnych mechanizmów systemu operacyjnego. Programy te będą niewielkie i pozbawione rozbudowanego interfejsu użytkownika. Jedynym sposobem komunikacji z takimi programami będą ich standardowe strumienie (wejściowy i wyjściowe) oraz argumenty przekazywane w linii poleceń. W języku C argumenty z linii poleceń są widoczne w programie jako argumenty funkcji `main()`. Pierwszy argumenty typu `int` (nazwa jest dowolna) określa liczbę argumentów przekazanych w linii poleceń (włączając w to również nazwę samego programu). Drugi argumenty typu `char**` jest tablicą łańcuchów tekstowych przechowujących wartości argumentów. Poniższy przykład pokazuje w jaki sposób można odczytać argumenty z linii poleceń:

```
int main(int argc, char* argv[])
{
  int i;
  for(i=0; i<argc; i++)
  {
    printf("argument %d: %s\n", i, argv[i]);
  }
}
```

2 Pliki

2.1 Podstawowe funkcje

1. Otworzenie pliku umożliwia funkcja systemowa open. Jej pierwszym argumentem jest ścieżka do pliku, który ma być otworzony. Drugim argumentem są flagi określające tryb otwarcia pliku. Trzeci (opcjonalny) argument określa prawa dostępu do nowego pliku, jeżeli ma on powstać w wyniku otwierania pliku. Najczęściej stosowane flagi dla funkcji open to:

   O_RDONLY otwarcie pliku tylko do odczytu,
   O_WRONLY otwarcie pliku tylko do zapisu,
   O_RDWR otwarcie pliku do odczytu i zapisu,
   O_CREAT flaga powodująca utworzenie otwieranego pliku w przypadku stwierdzenia jego braku.

   Oto przykładowe wywołanie funkcji open:

   ```
   fd = open("przyklad.txt", O_WRONLY | O_CREAT, 0644);
   ```

   Funkcja systemowa open zwraca *deskryptor pliku*. Deskryptor pliku jest liczbą, która powinna być interpretowana jako indeks do tablicy otwartych plików utrzymywanej przez system dla każdego procesu. Tablica ta jest niedostępna bezpośrednio dla programisty i zmiany do niej można wprowadzać jedynie poprzez wywołania odpowiednich funkcji systemowych. Poniższy przedstawiono przykładową zawartość tablicy otwartych plików:

0	stdin	standardowe wejście
1	stdout	standardowe wyjście
2	stderr	standardowe wyjście diagnostyczne
3	przykład.txt	otwarty plik
4		
…	…	

   Jak widać pierwsze 3 pozycje w tej tabeli są już wstępnie zainicjowane i reprezentują standardowe strumienie procesu. Funkcja open zajmuje zawsze pierwszą wolną pozycję w tablicy otwartych plików. Dla nowego procesu jest to więc pozycja o indeksie 3.

2. Zamknięcie pliku:

   ```
   close(fd);
   ```

Funkcja systemowa `close` wymaga podania deskryptora otwartego pliku jako argumentu. Niezamknięte pliki są automatycznie zamykane przez system w momencie kończenia procesu.

3. Zapis do pliku:

```
write(fd, "Czesc", 5);
```

Drugi argument funkcji systemowej `write` jest adresem miejsca w pamięci, z którego będą pobierane dane do zapisu. W powyższym przypadku drugi argument reprezentuje adres statycznego napisu. Ostatni argument określa liczbę bajtów do zapisu. Funkcja zwraca liczbę faktycznie zapisanych bajtów.

4. Odczyt z pliku:

```
char buf[20];
...
n = read(fd, buf, 20);
```

Funkcja systemowa `read` ma te same argumenty co funkcja `write`. W tym przypadku jednak użyto tablicy znaków jako drugiego argumentu (bufor danych). Ostatni argument wskazuje na liczbę bajtów jakie mają być odczytane. Funkcja `read` zwraca liczbę bajtów jakie faktycznie udało się przeczytać (wartość większa od zera) lub 0 jeżeli wskaźnik plikowy dotarł do końca pliku.

5. Odczyt całego pliku wymaga wielokrotnego wywołania funkcji `read`, ponieważ rozmiar pliku z reguły nie jest znany z góry. Poniższy przykład pokazuje przykładową pętlę odczytującą zawartość pliku i wyświetlającą odczytane dane na standardowym wyjściu, a więc kierując te dane do deskryptora o numerze 1. Wypisywanie na ekranie dotyczy takiej liczby bajtów jak faktycznie udało się przeczytać.

```
while((n=read(fd, buf, 20)) > 0)
{
  write(1, buf, n);
}
```

6. Funkcje systemowe zwracają wartość liczbową wskazującą na status wykonania zleconej operacji. Z reguły wartość -1 sygnalizuje wystąpienie błędu. Szczegółowy kod błędu można odczytać badając wartość globalnej zmiennej `errno` typu `int`. Do obsługi błędów przydatna będzie funkcja biblioteczna `perror`, która bada wartość zmiennej `errno` i wyświetla tekstowy opis błędu, który wystąpił. Typowo więc obsługa błędów wygląda więc następująco:

```
fd = open("przyklad.txt", O_RDONLY);
if (fd == -1)
{
  printf("Kod: %d\n", errno);
  perror("Otwarcie pliku");
  exit(1);
}
```

Odwołanie do zmiennej `errno` wymaga załączenia pliku nagłówkowego `errno.h`.

Przetestuj obsługę błędów na nieistniejącym pliku oraz na pliku, do którego nie ma prawa do odczytu.

7. Przetestuj wyświetlanie systemowych komunikatów w języku polskim. W tym celu należy ustawić odpowiednie *locale* na początku programu:

```
setlocale(LC_ALL, "pl_PL.UTF-8");
```

8. Napisz program dopisujący dane na końcu istniejącego pliku. Można w tym celu wykorzystać otwieranie pliku w trybie „dopisywania”:

```
fd = open("przyklad.txt", O_WRONLY | O_APPEND | O_CREAT, 0644);
```

9. Usuń istniejący plik:

```
fd = unlink("przyklad.txt");
```

   Usuwanie pliku nie wymaga jego otwierania.

10. Skasuj zawartość pliku funkcją systemową `ftruncate`:

```
fd = open("przyklad.txt", O_WRONLY);
ftruncate(fd, 0);
```

11. Maska dla nowotworzonych plików:

```
# umask
022
# umask 077
```

   (zobacz również opis funkcji systemowej `umask`)

12. Przećwicz przemieszczanie się wewnątrz pliku funkcją `lseek`, której nagłówek jest następujący:

```
int lseek(int fd, int offset, int whence);
```

   Parametr `offset` określa przesunięcie a parametr `whence` określa punkt odniesienia: `SEEK_SET` — względem początku, `SEEK_CUR` — względem bieżącej pozycji, `SEEK_END` — względem końca pliku. Funkcja — w przypadku poprawnego wykonania — zwraca przesunięcie w bajtach względem początku pliku.
   Przykładowe zlecenie:

```
lseek(fd, -1, SEEK_END);
```

   spowoduje przesunięcie wskaźnika plikowego na pozycję przed ostatnim bajtem w pliku.

2.2 Implementacja prostych narzędzi systemu Unix

1. Zaproponuj implementację programu `cat` do wyświetlania danych ze wskazanego pliku na standardowym wyjściu. Dodaj możliwość czytania z wielu plików. Przykład użycia programu:

```
# mycat a.txt b.txt c.txt
```

2. Napisz program kopiujący zawartość pliku wskazanego pierwszym argumentem do pliku wskazanego drugim argumentem — analogicznie do standardowej komendy `cp`.
3. Rozważ implementację programu `tee` powielającego dane ze standardowego wejścia na standardowe wyjście i do pliku. Przykład użycia programu:

```
# ps ax | mytee wynik.log
```

4. Napisz program obliczający liczbę znaków i linii w pliku — analogicznie do standardowej komendy `wc`.
5. Napisz program rozpoznawający czy plik dany argumentem jest plikiem tekstowym, a więc zawierającym tylko znaki o kodach od 32 do 127.

6. Napisz program wyświetlający zawartość wskazanego pliku wielkimi literami. Zastosuj funkcję biblioteczną `toupper`. Program powinien więc działać analogicznie do zlecenia:

   ```
   # cat plik.txt | tr a-z A-Z
   ```

7. Napisz program sprawdzający czy zawartość dwóch plików wskazanych argumentami jest identyczna — analogicznie do standardowej komendy `cmp`.
8. Napisz program wyświetlający rozmiar pliku wskazanego argumentem. Do badania rozmiaru wykorzystaj funkcję `lseek`.
9. Napisz program wypisujący od końca (znak po znaku) zawartość wskazanego pliku. Zaproponuj rozwiązanie, które będzie gwarantowało wysoką wydajność.

2.3 Blokowanie dostępu do plików

1. Współbieżny dostęp do plików wymaga blokowania dostępu do nich w celu zabezpieczenia się przed niepoprawnymi wynikami. W najprostszym przypadku blokowanie można zrealizować na poziomie całego pliku przy użyciu funkcji `flock`:

   ```
   int fd = open(...);
   flock(fd, LOCK_EX);
   ...
   flock(fd, LOCK_UN);
   ```

   Blokada typu `LOCK_EX` jest blokadą wyłączną (do zapisu). Blokada dzielona (do odczytu) jest zakładana flagą `LOCK_SH`.

2. Blokowanie dostępu do pliku na poziomie poszczególnych bajtów można zrealizować funkcją `fcntl`. Poniższy fragment kodu zakłada i zdejmuje blokadę na pliku:

   ```
   struct flock lck;
   int fd = open(...);
   lck.l_type = F_WRLCK;
   lck.l_start = 10;
   lck.l_len = 20;
   lck.l_whence = SEEK_SET;
   fcntl(fd, F_SETLKW, &lock);
   ...
   lck.l_type = F_UNLCK;
   fcntl(fd, F_SETLKW, &lock);
   ```

   Blokada w powyższym przykładzie jest wyłączna (typ `F_WRLCK`). Blokada dzielona jest uzyskiwana poprzez ustawienie pola `l_type` na `F_RDLCK`.

3 Procesy

3.1 Funkcja `fork`

1. Wykonaj program wywołujący następujące funkcje:

```
printf("Poczatek\n");
fork();
printf("Koniec\n");
```

2. Dodaj wywołanie funkcji bibliotecznej `sleep` za wywołaniem funkcji `fork`. Po uruchomieniu programu w tle, sprawdź komendą `ps -l` listę procesów w systemie. Zwróć uwagę na identyfikatory procesów macierzystych (kolumna PPID).

```
# ./a.out &
# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R  1011  2567  2566  0  75   0 -  1233 -      pts/2    00:00:00 bash
0 S  1011  2589  2567  0  75   0 -   337 schedu pts/2    00:00:00 a.out
0 S  1011  2590  2589  0  75   0 -   337 schedu pts/2    00:00:00 a.out
1 R  1011  2591  2567  0  75   0 -   905 -      pts/2    00:00:00 ps
```

3. Sprawdź wartość zwracaną przez funkcję `fork`. Wyświetl wartości PID i PPID korzystając z funkcji systemowych `getpid` i `getppid`. Przykładowa implementacja:

```
int x;
x = fork();
printf("wynik fork=%d  PID=%d  PPID=%d\n", x, getpid(), getppid());
```

4. Ile procesów powstanie po wykonaniu następującego fragmentu kodu:

```
fork();
fork();
```

   A ile dla tego fragmentu:

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

3.2 Funkcja exec

1. Wykonaj program wywołujący następujące funkcje:

```
printf("Poczatek\n");
execl("/bin/ls", "ls", "-l", NULL);
printf("Koniec\n");
```

2. Wykonaj polecenie `ls -l` w procesie potomnym:

```
printf("Poczatek\n");
if (fork()==0)
{
  /* proces potomny */
  execlp("ls", "ls", "-l", NULL);
  exit(1);
}
printf("Koniec\n");
```

3. Wprowadź oczekiwanie na zakończenie procesu potomnego w procesie macierzystym:

```
if (fork()==0)
{
  /* proces potomny */
  ...
}
else
{
  /* proces macierzysty */
  wait(NULL);
  printf("Koniec\n");
}
```

4. Odczytaj status zakończenia procesu potomnego:

```
if (fork()==0)
{
  /* proces potomny */
  exit(5);
}
else
{
  /* proces macierzysty */
  int stat;
  wait(&stat);
  printf("status=%d\n", stat>>8);
}
```

5. Zaobserwuj powstawanie procesów *zombie* dodając oczekiwanie funkcją `sleep` w procesie macierzystym.
6. Zapoznaj się z innymi wersjami funkcji `exec`, np. `execvp`. Napisz program, który wykona następujące polecenie korzystając z funkcji `execvp`:

```
sort -n -k 3,3 -t: /etc/passwd
```

3.3 Przekierowania strumieni standardowych

1. Wykonaj polecenie `ps ax` zapisując jego wynik do pliku `out.txt`. Oto przykładowy fragment implementacji:

```
int fd;
close(1);
fd = open("out.txt", O_WRONLY | O_CREAT, 0644);
execlp("ps", "ps", "ax", NULL);
```

2. Ustaw deskryptory plików na właściwych pozycjach w tablicy otwartych plików korzystając z funkcji systemowej `dup` lub `dup2`:

```
int fd;
fd = open("out.txt", O_WRONLY | O_CREAT, 0644);
dup2(fd, 1);
```

3. Zrealizuj programowo następujące zlecenie:

```
# ls -l -a -i >> wynik.txt
```

4. Zrealizuj programowo następujące zlecenie:

```
# grep xy < dane.txt > wynik1.txt 2> wynik2.txt
```

5. Napisz program który wykona zlecenie powłoki takie jak w poprzednim zadaniu, ale dodatkowo wypisze na ekranie komunikat przed i po wykonaniu zlecenia.

4 Sygnały

4.1 Obsługa sygnałów

1. Przećwicz obsługę sygnału SIGINT następującym programem:

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

void obsluga(int signo)
{
  printf("Odebrano sygnał %d\n", signo);
}

int main()
{
  signal(SIGINT, obsluga);
  while(1) ;          /* p tla niesko czona */
}
```

   Uruchomienie programu:

```
# ./a.out
^C
Odebrano sygnał 2
^\
Quit
```

2. Zapoznaj się z listą dostępnych sygnałów na stronie pomocy systemowej signal(7).
3. Sprawdź jakie sygnały są generowane po naciśnięciu kombinacji: Ctrl-c, Ctrl-\, Ctrl-z. Wyślij do programu testowego komendą `kill` sygnały: SIGHUP, SIGTERM, SIGQUIT, SIGSTOP, SIGCONT, SIGKILL.
4. Wyślij sygnał do procesu potomnego:

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

void obsluga(int signo)
```

```
{
  printf("Odebrano sygnał %d\n", signo);
  exit(0);
}

int main()
{
  int pid;
  if ((pid=fork())==0)
  {
    signal(SIGINT, obsluga);
    while(1);
  }
  sleep(2);
  printf("Wysyłam sygnał do procesu potomnego %d\n", pid);
  kill(pid, 2);
  return 0;
}
```

5. Korzystając z funkcji systemowej `alarm` przerwij działanie pętli nieskończonej z punktu 1 po upływie 5 sekund.
6. Napisz program, który będzie czekał na odbiór sygnału SIGINT poprzez wywołanie funkcji `pause`. Program powinien wypisywać na ekranie komunikat przed i po wywołaniu funkcji `pause`, np.:

```
printf("Press Ctrl-C\n");
pause();
printf("Thanks!\n");
```

7. Napisz własną implementację funkcji `sleep`.
8. Zmodyfikuj przykład z punktu 1 tak, aby pętla główna badała wartość zmiennej globalnej:

```
int x = 0;
...
while(x==0) ;
```

Procedura obsługi sygnału powinna ustawić zmienną x na 1, powodując w ten sposób zakończenie wykonywania pętli. Sprawdź działanie programu po skompilowaniu go z optymalizacją:

```
# cc -O2 czekaj.c -o czekaj
```

4.2 Asynchroniczna obsługa procesów zombie

1. Sprawdź numer i nazwę sygnału przesyłanego przez proces potomny do procesu macierzystego w momencie zakończenia procesu potomnego.
2. Napisz procedurę obsługi sygnału SIGCHLD, w której proces macierzysty będzie odczytywał status zakończenia procesu potomnego:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
void obsluga(int signo)
{
  int s;
  wait(&s);
  printf("Status zakończenia procesu potomnego: %d\n", s>>8);
}

int main()
{
  signal(SIGCHLD, obsluga);
  if (fork()==0)
  {
    sleep(3);
    exit(5);
  }
  while(1)
  {
    sleep(1);
    printf("Proces macierzysty\n");
  }
}
```

5 Potoki

5.1 Potoki nienazwane

1. Utwórz potok funkcją systemową `pipe`, zapisz do niego przykładowy napis, a następnie odczytaj:

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

int main()
{
  int fd[2];
  char buf[100];

  pipe(fd);
  write(fd[1], "Hello", 6);
  read(fd[0], buf, 6);
  printf("%s\n", buf);
  return 0;
}
```

2. Przenieś proces zapisu do potoku do procesu potomnego. Proces macierzysty powinien utworzyć potok, a następnie uruchomić proces potomny:

```
pipe(fd);
if (fork()==0)
{
  write(...);
  exit(0);
}
...
```

   Dodając oczekiwanie funkcją `sleep` w procesie potomnym przed zapisem do potoku można zaobserwować pracę dwóch procesów w systemie. Proces macierzysty (czytający) czeka na dane z potoku.

3. Napisz program, który zrealizuje przetwarzanie potokowe: `ls | tr a-z A-Z`.

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

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

  pipe(fd);
  if (fork()==0)
  {
    dup2(fd[1], 1);
    execlp("ls", "ls", NULL);
  }
  else {
    dup2(fd[0], 0);
    execlp("tr", "tr", "a-z", "A-Z", NULL);
  }
  return 0;
}
```

Uzupełnij kod programu tak, aby się poprawnie kończył.

4. Uzupełnij program z poprzedniego punktu tak, aby wynik przetwarzania trafiał do pliku `out.txt`, jak w poniższym zleceniu:

```
# ls | tr a-z A-Z > out.txt
```

5. Napisz program, który wykona następujące zlecenie:

```
# ls -l /tmp | sort -n +4 | tail -5
```

6. Napisz funkcję, która zwraca pełną domenową nazwę komputera. Nazwa powinna być pobierana z polecenia:

```
# hostname -f
```

5.2 Potoki nazwane

1. Napisz dwa programy, które skomunikują się za pośrednictwem łącza nazwanego. Pierwszy program powinien utworzyć takie łącze i zapisać do niego wybrany łańcuch tekstowy, a drugi powinien z tego łącza odczytać dane. Przykładowa implementacja programu piszącego:

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

int main()
{
  int fd;

  mkfifo("fifo", 0600);
  fd = open("fifo", O_WRONLY);
  write(fd, "Hello", 6);
  close(fd);
  return 0;
}
```

2. Zrealizuj zlecenie `ps ax | tr a-z A-Z`, wykorzystując do komunikacji potok nazwany.

5.3 Komunikacja dwukierunkowa

1. Napisz program komunikujący się dwukierunkowo ze swoim procesem potomnym. Proces potomny powinien dopisać do przekazanego łańcucha tekstowego napis „OK" i przesłać go do procesu macierzystego. Należy skorzystać z pary lokalnych gniazdek komunikacyjnych tworzonych funkcją systemową `socketpair`:

```
int fd[2];

if ((socketpair(PF_UNIX, SOCK_STREAM, 0, fd))==-1)
{
  perror("socketpair");
  exit(1);
}
```

6
Pamięć współdzielona

6.1 Dostęp do pamięci współdzielonej

1. Utwórz blok pamięci współdzielonej korzystając z poniższego kodu:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
  int id;
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  printf("Blok utworzony: %d\n", id);
  return 0;
}
```

2. Po utworzeniu bloku sprawdź jego istnienie komendą `ipcs`:

```
# ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000111 3342338    sobaniec   700        1024       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

3. Usuń blok pamięci komendą `ipcrm`:

```
# ipcrm -M 0x111
```

lub:

```
# ipcrm -m 3342338
```

4. Dołącz blok pamięci współdzielonej do przestrzeni adresowej procesu i wpisz dowolny napis do tego bloku:

```
char* ptr;
...
ptr = shmat(id, NULL, 0);
strcpy(ptr, "Ala ma kota");
```

5. Napisz program, który będzie odczytywał zawartość łańcucha tekstowego z pamięci współdzielonej.
6. Odłącz blok pamięci współdzielonej od przestrzeni adresowej procesu:

```
char* ptr;
...
shmdt(ptr);
```

7. Usuń blok pamięci współdzielonej:

```
id = shmget(0x111, 1024, 0);
...
shmctl(id, IPC_RMID, NULL);
```

6.2 Synchronizacja procesów

1. Napisz dwa programy, zapisujący i odczytujący z bloku pamięci współdzielonej. Program piszący powinien zapisywać do pamięci naprzemiennie, w to samo miejsce napisy: „xxxxxx" i „oooooo". Program czytający odczytuje zawartość bufora i sprawdza czy jest to jeden z wpisywanych napisów. W przypadku błędu, wypisuje niepoprawny łańcuch na ekranie.
2. Przeanalizuj działanie programu z przykładu 6.1 przesyłającego napisy pomiędzy procesami. Czy zaproponowana synchronizacja jest wystarczająca? Porównaj przedstawione rozwiązanie z przykładami 6.2 i 6.3. Przetestuj działanie programu z przykładu 6.2 po dodaniu dodatkowego wywołania `sleep(1)` zaraz za wywołaniem funkcji `printf` w procesie macierzystym.

Przykład 6.1: Synchronizacja procesow — wersja 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

void obsluga(int sig)
{
  printf("sygnał w procesie %d\n", getpid());
}

int main()
{
  int id;
  char *ptr;
  int pid;
  signal(SIGHUP, obsluga);
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  ptr = (char*)shmat(id, NULL, 0);
  if ((pid=fork())==0)
  {
    int ppid = getppid();
    sleep(1);
    while(1)
    {
      strcpy(ptr, "xxxxxx");
      sleep(1);
      kill(ppid, 1);
      pause();
      strcpy(ptr, "oooooo");
      sleep(1);
      kill(ppid, 1);
      pause();
    }
  }
  else
  {
    while(1)
    {
      pause();
      printf("%s\n", ptr);
      sleep(1);
      kill(pid, 1);
    }
  }
}
```

Przykład 6.2: Synchronizacja procesow — wersja 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int ok = 0;

void obsluga(int sig) {
  ok = 1;
}

int main()
{
  int id;
  char *ptr;
  int pid;
  signal(SIGHUP, obsluga);
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  ptr = (char*)shmat(id, NULL, 0);
  if ((pid=fork())==0)
  {
    int ppid = getppid();
    while(1)
    {
      strcpy(ptr, "xxxxxx");
      ok = 0;
      kill(ppid, 1);
      while(!ok);
      strcpy(ptr, "oooooo");
      ok = 0;
      kill(ppid, 1);
      while(!ok);
    }
  }
  else
  {
    while(1)
    {
      while(!ok);
      printf("%s\n", ptr);
      ok = 0;
      kill(pid, 1);
    }
  }
}
```

Przykład 6.3: Synchronizacja procesow — wersja 3

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

void obsluga(int sig)
{
}

int main()
{
  int id;
  char *ptr;
  int pid;
  sigset_t mask, mask_old;
  signal(SIGHUP, obsluga);
  sigemptyset(&mask);
  sigaddset(&mask, SIGHUP);
  sigprocmask(SIG_BLOCK, &mask, &mask_old);
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  ptr = (char*)shmat(id, NULL, 0);
  if ((pid=fork())==0)
  {
    int ppid = getppid();
    while(1)
    {
      strcpy(ptr, "xxxxxx");
      kill(ppid, 1);
      sigsuspend(&mask_old);
      strcpy(ptr, "oooooo");
      kill(ppid, 1);
      sigsuspend(&mask_old);
    }
  }
  else
  {
    while(1)
    {
      sigsuspend(&mask_old);
      printf("%s\n", ptr);
      kill(pid, 1);
    }
  }
}
```

7 Semafory

7.1 Obsługa semaforów

1. Poniższy program tworzy semafor i wykonuje na nim podstawowe operacje.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>

int main()
{
  int sid;
  struct sembuf op;

  sid = semget(0x123, 1, 0600 | IPC_CREAT);
  if (sid == -1)
  {
    perror("semget");
    exit(1);
  }

  op.sem_num = 0;
  op.sem_flg = 0;

  /* zwi kszenie warto ci semafora o 1 */
  op.sem_op = 1;
  semop(sid, &op, 1);

  /* zmniejszenie warto ci semafora o 1 */
  op.sem_op = -1;
  semop(sid, &op, 1);

  return 0;
}
```

2. Po uruchomieniu sprawdź istnienie semafora w systemie komendą `ipcs`:

```
# ipcs
------ Shared Memory Segments --------
```

```
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000123 32768      sobaniec   700        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

3. Usuń semafor z systemu komendą `ipcrm`:

```
# ipcrm -S 0x123
```

lub:

```
# ipcrm -s 32768
```

4. Odczytaj bieżącą wartość semafora i ustaw ją na wybraną wartość:

```
int num;
...
num = semctl(sid, 0, GETVAL, 0);
printf("semaphore value: %d\n", num);
num = 1;
semctl(sid, 0, SETVAL, num);
```

5. Poniższy program umożliwia wykonywanie operacji *P* i *V* na semaforze. Sprawdź zachowanie programu w przypadku zmniejszania wartości semafora poniżej zera.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>

int main(int argc, char* argv[])
{
  int sid;
  struct sembuf op;

  sid = semget(0x123, 1, 0600 | IPC_CREAT);
  if (sid == -1)
  {
    perror("semget");
    exit(1);
  }

  printf("przed: %d\n", semctl(sid, 0, GETVAL, NULL));
  op.sem_num = 0;
  op.sem_flg = 0;
  op.sem_op = atoi(argv[1]);
  semop(sid, &op, 1);
  printf("po:    %d\n", semctl(sid, 0, GETVAL, NULL));

  return 0;
}
```

6. Uruchom kilkukrotnie poniższy program, który wykonuje operacje w sekcji krytycznej. Pierwsza kopia powinna być uruchomiona z dodatkowym argumentem powodując w ten sposób zainicjowanie semafora.

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

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/sem.h>

int main(int argc, char* argv[])
{
  int sid;
  int num;
  struct sembuf op;

  sid = semget(0x123, 1, 0600 | IPC_CREAT);
  if (sid == -1)
  {
    perror("semget");
    exit(1);
  }

  if (argc>1)
  {
    printf("Inicjacja semafora...\n\n");
    num = 1;
    semctl(sid, 0, SETVAL, num);
  }

  op.sem_num = 0;
  op.sem_flg = 0;
  while(1)
  {
    printf("Wchodzę do sekcji krytycznej...\n");
    op.sem_op = -1;
    semop(sid, &op, 1);
    printf("Jestem w sekcji krytycznej ...\n");
    sleep(5);
    op.sem_op = 1;
    semop(sid, &op, 1);
    printf("Wyszedłem z sekcji krytycznej ...\n");
    sleep(1);
  }
  return 0;
}
```

7.2 Problem producenta-konsumenta

1. Zrealizuj wymianę danych pomiędzy dwoma procesami (producentem i konsumentem) synchronizując ich pracę semaforami. Dane powinny być przechowywane w pamięci współdzielonej w postaci wieloelementowej tablicy. Praca procesów może się odbywać współbieżnie o ile dotyczy rozłącznych bloków pamięci.

   Przykład 7.1 przedstawia przykładową implementację procesu producenta. Na podstawie kodu opracuj implementację procesu konsumenta.

2. Czy implementacja procesu producenta z przykładu 7.1 jest wystarczająca w przypadku współbieżnej pracy wielu procesów produkujących dane? Jeżeli nie, to jakie zmiany należałoby wprowadzić?

Przykład 7.1: Proces producenta

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define N 10         /* rozmiar wspó dzielonej tablicy */

int main()
{
  int shmid;         /* systemowy identyfikator bloku pami ci dzielonej */
  int semid;         /* systemowy identyfikator semafora */
  int *tab;          /* wska nik na tablic  z danymi */
  int pos;           /* pozycja w tablicy */
  int num;           /* aktualnie zapisywana warto   */
  struct sembuf op;  /* struktura dla operacji na semaforze */

  shmid = shmget(0x123, sizeof(int)*N, 0600 | IPC_CREAT);
  if (shmid == -1)
  {
    perror("shmget");
    exit(1);
  }
  tab = (int*)shmat(shmid, NULL, 0);
  if (tab == NULL)
  {
    perror("shmat");
    exit(1);
  }

  /* utworzenie nowego dwuelementowego semafora
   * 0 - semafor producenta
   * 1 - semafor konsumenta
   */
  semid = semget(0x234, 2, 0600 | IPC_CREAT);
  if (semid == -1)
  {
    perror("semget");
    exit(1);
  }
  /* ustawienie warto ci pocz tkowych semaforów */
  num = N;
  semctl(semid, 0, SETVAL, num);
  num = 0;
  semctl(semid, 1, SETVAL, num);

  op.sem_flg = 0;
  pos = 0;
  num = 1;
  while(1)
  {
    /* opuszczenie semafora producenta */
    op.sem_num = 0;
    op.sem_op = -1;
    semop(semid, &op, 1);

    /* zapis elementu do tablicy */
    tab[pos] = num;
    printf("Na pozycji %d zapisałem %d\n", pos, num);
    pos = (pos + 1) % N;
    num++;
    sleep(1);

    /* podniesienie semafora konsumenta */
    op.sem_num = 1;
    op.sem_op = 1;
    semop(semid, &op, 1);
  }
}
```

8
Kolejki komunikatów

1. Utwórz kolejkę komunikatów wywołując funkcję `msgget(2)`:

   ```
   mid = msgget(0x123, 0600 | IPC_CREAT);
   ```

   Sprawdź komendą `ipcs` czy kolejka komunikatów faktycznie została utworzona. Zadbaj o obsługę ewentualnych błędów poprzez sprawdzenie zwracanej przez funkcję wartości.

2. Zadeklaruj w programie strukturę do reprezentowania wiadomości. Pierwsze pole w strukturze służy do przechowywania informacji o typie komunikatu. Pole to musi być typu `long`. Pozostałe pola mogą być dowolnego typu. Poniżej przedstawiono przykładową strukturę do przesyłania wiadomości tekstowych:

   ```
   typedef struct
   {
     long type;
     char text[1024];
   } MSG;
   ```

3. Wyślij wiadomość korzystając z funkcji `msgsnd(2)`:

   ```
   MSG msg;
   ...
   msgsnd(mid, &msg, strlen(msg.text)+1, 0);
   ```

   Po wysłaniu wiadomości sprawdź komendą `ipcs` czy kolejka faktycznie zawiera nową wiadomość.

4. Odbierz wiadomość z kolejki korzystając z funkcji `msgrcv(2)`:

   ```
   MSG msg;
   ...
   msgrcv(mid, &msg, 1024, 0, 0);
   ```

   Trzeci argument funkcji `msgrcv` reprezentuje rozmiar przekazanej struktury reprezentującej komunikat z pominięciem początkowego pola typu `long`. Czwarty argument wskazuje na typ wiadomości, która powinna zostać pobrana z kolejki:

   - wartość 0 oznacza, że ma być pobrana pierwsza wiadomość z kolejki,
   - wartość dodatnia oznacza, że ma być pobrana pierwsza wiadomość z kolejki o typie równym argumentowi,

- wartość ujemna oznacza, że ma być pobrana pierwsza wiadomość z kolejki o typie mniejszym lub równym wartości bezwzględnej argumentu.

Ostatni argument to dodatkowe flagi doprecyzowujące tryb pracy funkcji `msgrcv`.

5. Usuń kolejkę komunikatów korzystając z funkcji `msgctl`:

```
msgctl(mid, IPC_RMID, NULL);
```

Usuwanie można również przeprowadzić poleceniem `ipcrm`.

6. Napisz program serwera, który w pętli nieskończonej odbiera komunikaty z kolejki i wyświetla je na ekranie. Z programem tym komunikuje się program klienta, który wysyła wiadomość przekazaną jako argument w linii poleceń.
7. Zmodyfikuj implementację serwera z poprzedniego przykładu tak, aby serwer odpowiadał na żądanie klienta odsyłając tą samą wiadomość tekstową, ale z zamienionymi wszystkimi literami na wielkie.

   Podpowiedź: komunikację dwustronną można przeprowadzić poprzez wykorzystanie różnych typów komunikatów. Pierwszy argument dla programu klienta powinien więc określać typ komunikatów odbieranych przez tego klienta.
8. Przetestuj odbiór wiadomości w trybie nieblokującym.
9. Sprawdź czy są jakieś wiadomości w kolejce bez ich odbierania.

9
Wątki

1. Uruchom poniższy program tworzący pojedynczy wątek:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void* worker(void* info)
{
  int i;
  for(i=0; i<10; i++)
  {
    sleep(1);
    printf("thread\n");
  }
  return NULL;
}

int main()
{
  pthread_t th;
  int i;

  pthread_create(&th, NULL, worker, NULL);
  for(i=0; i<10; i++)
  {
    sleep(1);
    printf("main program\n");
  }
  pthread_join(th, NULL);
  return 0;
}
```

   Kompilację należy przeprowadzić zleceniem:

```
# cc -o thtest thtest.c -lpthread
```

2. Wyświetl listę procesów i sprawdź obecność wątków:

```
# ps x
# ps -L x
```

```
# ps -T x
```

Zwróć uwagę na kolumny PID, LWP (*light weight process*) i SPID.

3. Zmodyfikuj powyższy program tak, aby tworzył 2 wątki. Każdy wątek powinien odebrać liczbę jako swój parametr:

```
int d = 1;
pthread_create(&th, NULL, worker, &d);
```

4. Zweryfikuj działanie sekcji krytycznej:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t mx;

void* worker(void* info)
{
  int i;
  int tid = *(int*)info;
  for(i=0; i<10; i++)
  {
    printf("Thread %d is entering critical section...\n", tid);
    pthread_mutex_lock(&mx);
    printf("Thread %d is in critical section...\n", tid);
    sleep(5);
    pthread_mutex_unlock(&mx);
    printf("Thread %d is leaving critical section...\n", tid);
    sleep(2);
  }
  return NULL;
}

int main()
{
  pthread_t th1, th2, th3;
  int w1=1, w2=2, w3=3;

  pthread_mutex_init(&mx, NULL);

  pthread_create(&th1, NULL, worker, &w1);
  pthread_create(&th2, NULL, worker, &w2);
  pthread_create(&th3, NULL, worker, &w3);

  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  pthread_join(th3, NULL);

  return 0;
}
```

5. Odbierz dane wyjściowe z wątku podczas wywołania funkcji `pthread_join(3)`.
6. Zaimplementuj program przesyłający dane pomiędzy wątkiem-producentem a wątkiem-konsumentem.

9.1 Zarządzanie wątkami

Wątek może oczekiwać na zakończenie innego wątku funkcją:

```
int pthread_join(pthread_t th, void **retval);
```

Jest to funkcja analogiczna do funkcji `wait` oczekującej na zakończenie procesu potomnego. Wskaźnik `retval` zostanie zainicjowany wartością zwróconą przez wątek.

Wykonanie wątku można zakończyć w dowolnym momencie wywołaniem funkcji:

```
void pthread_exit(void *retval);
```

Wartość `retval` może być odczytana przez inny wątek, który zsynchronizuje się z nim wywołaniem `pthread_join`.

Wątek można zatrzymać z poziomu innego wątku funkcją `pthread_cancel`:

```
int pthread_cancel(pthread_t thread);
```

Zatrzymywanie wątku można blokować funkcją:

```
int pthread_setcancelstate(int state, int *oldstate);
```

Wątek może „odłączyć" się od procesu i kontynuować pracę niezależnie od wątku głównego. Służy do tego funkcja:

```
int pthread_detach(pthread_t th);
```

Uniezależnienie wątku oznacza, że jego zasoby będą zwolnione po jego zakończeniu. Z drugiej jednak strony nie będzie możliwe zsynchronizowanie z innym wątkiem poprzez wywołanie `pthread_join`.

9.2 Obsługa sygnałów

Wszystkie wątki współdzielą między sobą jedną tablicę z adresami procedur obsługi. Każdy wątek może zmieniać przypisaną wcześniej do sygnału procedurę standardową funkcją `signal`. Istnieje możliwość wysyłania sygnałów do poszczególnych wątków:

```
int pthread_kill(pthread_t thread, int signo);
```

Wątki mogą blokować odbiór określonych sygnałów używając funkcji:

```
int pthread_sigmask(int how, const sigset_t *newmask,
                                    sigset_t *oldmask);
```

gdzie `newmask` jest maską sygnałów (4 liczby typu `unsigned long`) tworzoną z użyciem funkcji:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Wywołanie funkcji `pthread_sigmask` z argumentem drugim ustawionym na NULL powoduje odczytanie aktualnej maski sygnałów.

W ramach synchronizacji wątków można wymusić oczekiwanie na określony sygnał:

```
int sigwait(const sigset_t *set, int *sig);
```

9.3 Synchronizacja wątków

Do synchronizacji wątków wykorzystywane są następujące mechanizmy: zamki (ang. *mutex*), zmienne warunkowe (ang. *conditional variable*) i semafory POSIX.

9.3.1 Zamki

Zamki są binarnymi semaforami, a więc mogą znajdować się w jednym z dwóch stanów: podniesiony lub opuszczony. Do inicjowania zamka wykorzystywana jest funkcja:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

Poniższy fragment kodu tworzy zamek z domyślnymi atrybutami:

```
pthread_mutex_t  m;
...
pthread_mutex_init(&m, NULL);
```

Do wykonywania operacji na zamkach służą następujące funkcje:

`pthread_mutex_lock()`
: Zajęcie zamka. Funkcja jest blokująca do czasu aż operacja może zostać wykonana.

`pthread_mutex_trylock()`
: Zajęcie wolnego zamka. Próba zajęcia już zajętego zamka kończy się zasygnalizowaniem błędu (`EBUSY`).

`pthread_mutex_unlock()`
: Zwolnienie zamka. Zwolnienia powinien dokonać wątek, który zajął zamek.

`pthread_mutex_destroy()`
: Usunięcie zamka.

Wszystkie wymienione funkcje pobierają wskaźnik do struktury `pthread_mutex_t` jako jedyny argument i zwracają wartość typu `int`.

9.3.2 Zmienne warunkowe

Zmienne warunkowe pozwalają na kontrolowane zasypianie i budzenie procesów w momencie zajścia określonego zdarzenia.

Poniższy fragment kodu tworzy zmienną warunkową z domyślnymi atrybutami:

```
pthread_cond_t  c;
...
pthread_cond_init(&c, NULL);
```

Budzenie wątków realizowane jest z wykorzystaniem jednej z dwóch poniższych funkcji:

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Funkcja `pthread_cond_signal()` budzi co najwyżej jeden wątek oczekujący na wskazanej zmiennej warunkowej. Funkcja `pthread_cond_broadcast()` budzi wszystkie wątki uśpione na wskazanej zmiennej warunkowej.

Kolejne dwie funkcje służą do usypiania wątku na zmiennej warunkowej:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t  *mutex,
                           const struct timespec *abstime);
```

Funkcja `pthread_cond_wait()` oczekuje bezwarunkowo do czasu odebrania sygnału budzącego, podczas gdy funkcja `pthread_cond_timedwait()` ogranicza maksymalny czas oczekiwania. Zaśnięcie wątku wymaga użycia jednocześnie zmiennej warunkowej i zamka. Przed zaśnięciem zamek musi być już zajęty. Zaśnięcie oznacza atomowe zwolnienie zamka i rozpoczęcie oczekiwania na sygnał budzący. Obudzenie wątku powoduje ponowne zajęcie zamka. Poniższy przykład pokazuje zastosowanie zmiennej warunkowej do synchronizacji wątków:

- wątek oczekujący

```
pthread_cond_t  c;
pthread_mutex_t  m;
...
pthread_mutex_lock(&m);       /* zajęcie zamka */
pthread_cond_wait(&c, &m);    /* oczekiwanie na zmiennej warunkowej */
pthread_mutex_unlock(&m);     /* zwolnienie zamka */
```

- wątek budzący

```
pthread_cond_signal(&c);      /* sygnalizacja zmiennej warunkowej */
```

Pomimo, że nie jest to wymagane, często do poprawnej synchronizacji wywołuje się funkcję budzącą podczas przetrzymywania zamka:

```
pthread_mutex_lock(&m);
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

9.3.3 Semafory

Do synchronizacji wątków można wykorzystać semafory standardu POSIX. Jest to rozwiązanie całkowicie niezależne od semaforów Systemu V będących częścią zestawu mechanizmów komunikacji międzyprocesowej IPC. Semafor POSIX jest licznikiem przyjmującym wartości od zera wzwyż. Do obsługi semaforów POSIX przewidziano następujące funkcje:

`sem_init()`
: Funkcja tworząca i inicjująca nowy semafor. Semafor może być strukturą wewnętrzną procesu lub może służyć do synchronizacji niezależnych procesów, podobnie jak semafory IPC[1]. Argumentem funkcji `sem_init()` jest początkowa wartość semafora.

`sem_wait()`
: Oczekiwanie na wartość semafora większą od zera i obniżenie jej o 1.

`sem_trywait()`
: Nieblokująca próba zmniejszenia wartości semafora o 1.

`sem_post()`
: Zwiększenie wartości semafora o 1. Operacja zawsze wykonuje się poprawnie i nie jest blokująca.

`sem_getvalue()`
: Pobranie aktualnej wartości semafora.

[1] Aktualna implementacja LinuxThreads nie wspiera semaforów synchronizujących niezależne procesy.

`sem_destroy()`
: Usunięcie semafora.

Przykład 9.1 prezentuje proste zastosowanie semaforów standardu POSIX. Jest to oczywiście zmodyfikowana wersja przykładu **??**.

Zadania

1. Przećwicz przekazywanie argumentów do wątku i odbiór wartości zwrotnych. Uruchom w tym celu 3 nowe wątki, które będą zwracały podwojoną wartość typu `int`.
2. Utwórz 2 wątki i wstrzymaj ich wykonanie zmienną warunkową. Wątek główny powinien wznowić ich pracę sygnalizując to odpowiedniej zmiennej warunkowej. Sprawdź różnicę pomiędzy funkcją `pthread_cond_signal()` a `pthread_cond_broadcast()`.
3. Zaproponuj implementację operacji `bariera()`, której działanie polegałoby na zsynchronizowaniu wskazanej liczby wątków. Operacja kończy się w momencie jej wywołania przez wszystkie wątki.
4. Zaproponuj synchronizację 2 wątków w problemie producenta-konsumenta. Przeanalizuj czy rozwiązanie to działa poprawnie również w przypadku większej liczby producentów i konsumentów.

Przykład 9.1: Proste zastosowanie semaforów POSIX

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

sem_t s;

void* work(void* arg)
{
  int id = pthread_self();
  int i;
  for(i=0; i<10; i++)
  {
    printf("[%d] Czekam...\n", id);
    sem_wait(&s);
    printf("[%d] Sekcja krytyczna...\n", id);
    sleep(1);
    printf("[%d] Wyjście...\n", id);
    sem_post(&s);
    usleep(100000);
  }
  return NULL;
}

int main()
{
  pthread_t th1, th2;

  sem_init(&s, 0, 1);
  pthread_create(&th1, NULL, work, NULL);
  pthread_create(&th2, NULL, work, NULL);
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  return 0;
}
```

10 Tworzenie bibliotek

10.1 Kompilacja projektu złożonego

Jeżeli projekt składa się z kilku plików źródłowych, to można go kompilować pojedynczym zleceniem, np.:

```
# cc -o prg modul_1.c modul_2.c prg.c
```

gdzie pliki `modul\_1.c`, `modul\_2.c` i `prg.c` zawierają różne fragmenty aplikacji. Oczywiście tylko jeden z tych plików powinien mieć zdefiniowaną funkcję `main`. Kompilację można przeprowadzić również krokowo:

```
# cc -c modul_1.c
# cc -c modul_2.c
# cc -o prg modul_1.o modul_2.o prg.c
```

10.2 Biblioteki statyczne

Biblioteki statyczne składają się z kilku plików obiektowych `.o` połączonych w jeden plik z rozszerzeniem `.a`. Tworzenie bibliotek przebiega podobnie do tworzenia archiwum programem `tar`. Oto przykład utworzenia biblioteki z modułów `modul\_1.c` i `modul\_2.c`:

```
# cc -c modul_1.c
# cc -c modul_2.c
# ar rv libmodul.a modul_1.o modul_2.o
```

Kompilacja głównego programu z wykorzystaniem biblioteki statycznej może przebiegać następująco:

```
# cc -o prg prg.c libmodul.a
```

10.3 Biblioteki dynamiczne

Biblioteki dynamiczne są ładowane dynamicznie podczas uruchamiania aplikacji i są współdzielone pomiędzy aplikacjami. Oto przykład kompilacji:

```
# cc -shared -o libmodul.so modul_1.c modul_2.c
```

I kompilacja aplikacji:

```
# cc -o prg prg.c -lmodul -L .
```

Przełącznik `-L` wskazuje na katalog, który ma być przeszukiwany podczas wyszukiwania biblioteki `modul`. Listę dynamicznie dołączanych bibliotek aplikacji można wyświetlić komendą `ldd`:

```
# ldd prg
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0x40027000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
        modul.so => ./modul.so (0x40027000)
```

Dynamiczne wywoływanie metod

Metody zapisane w bibliotekach dzielonych można wywoływać dynamicznie, poprzez podanie odpowiednich nazw funkcji. Wymaga to użycia dynamicznego linkera. Oto przykładowy programu:

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main()
{
  void* handle = dlopen("libhello.so", RTLD_LAZY);
  void (*fun)();
  if (handle == NULL)
  {
    printf("%s\n", dlerror());
    exit(1);
  }
  else
  {
    fun = dlsym(handle, "fun");
    (*fun)();
    dlclose(handle);
  }
  return 0;
}
```

Implementacja biblioteki dzielonej:

```
#include <stdio.h>

void fun()
{
  printf("Hello world\n");
}
```

Więcej informacji na temat dynamicznego ładowania kodu można znaleźć na stronie pomocy systemowej `dlopen(3)`.

11 Inne biblioteki

11.1 Liczby pseudolosowe

Generowanie liczb pseudolosowych może być zrealizowane z wykorzystaniem funkcji `rand(3)`, która przy każdym wywołaniu zwraca liczbę z przedziału od 0 do `RAND_MAX` (predefiniowana stała). Generator liczb losowych inicjowany jest funkcją `srand(3)`, której argumentem jest liczba. Losowanie liczb z ograniczonego przedziału można wykonać w oparciu o losową wartość z przedziału $[0, 1]$, którą uzyskujemy następująco:

```
float f = rand()/(RAND_MAX+1.0);
```

Poniższy fragment kodu przypisuje do zmiennej x losową wartość z przedziału od 1 do 10:

```
int x;
srand(getpid());
x = 1 + 10.0*(rand()/(RAND_MAX+1.0));
```

Funkcja `getpid` zwraca identyfikator procesu, dzięki czemu generator będzie zwracał inne wartości przy każdym uruchomieniu programu, bo kolejne wykonania będą realizowane w procesach o innych identyfikatorach.

11.2 getopt

11.3 readline

11.4 curses

11.5 locale

12 Zadania projektowe

Należy wybrać *jedno* z poniższych zadań i napisać program w C rozwiązujący to zadanie.

Zadanie 1

Napisz program, w którym proces macierzysty uruchomi zadaną jako pierwszy parametr wywołania liczbę procesów potomnych. Każdy proces potomny zapisuje swój identyfikator (PID) do pliku. Nazwa tego pliku to `proces.txt` jeśli nie podano trzeciego parametru wywołania; jeśli taki parametr zostanie podany to jest on właściwą nazwą pliku identyfikatorów procesów. Dalej, każdy proces potomny odczekuje losową[1] liczbę sekund i się kończy z losowym statusem zakończenia. Proces macierzysty oczekuje na zakończenie pierwszych N procesów potomnych (gdzie N jest podane jako drugi parametr wywołania) i zapisuje ich identyfikatory do pliku łącznie z ich statusami zakończenia. Następnie proces macierzysty kończy pozostałe procesy potomne wysyłając sygnał TERM. Procesy powinny także wyświetlać komunikaty kontrolne na standardowym wyjściu, informujące o rozpoczęciu i zakończeniu pracy (również po odebraniu sygnału). Współbieżny dostęp do pliku `proces.txt` powinien być zabezpieczony poprzez blokowanie dostępu[2].

Zadanie 2

Napisz programy serwera i klienta. Proces klienta przesyła do procesu serwera informacje o poleceniu, którego wynik chce poznać; 1: `ps -ax`.; 2: `iostat`; 3: `netstat -tn`. Serwer po odebraniu zgłoszenia klienta wykonuje właściwe polecenie i jego wynik przesyła do procesu klienta. Serwer odnotowuje także każde zgłoszenie procesu klienta w pliku `log.txt`, a proces klienta wyświetla otrzymany wynik na standardowym wyjściu. Wybór polecenia, którego wynik pozyska klient jest wskazywany pierwszym parametrem wywołania programu klienta. Procesy powinny także wyświetlać odpowiednie komunikaty kontrolne na standardowym wyjściu. Zakończenie pracy procesu serwera następuje po wciśnięciu kombinacji klawiszy Ctrl-c. Komunikacja pomiędzy serwerem a klientem powinna być zrealizowana z wykorzystaniem potoków nazwanych lub kolejek komunikatów.

[1] Zobacz punkt 11.1.
[2] Zobacz punkt 2.3.

Zadanie 3

Napisz programy klienta i serwera, które realizują następujący scenariusz: proces serwera wyznacza średnią arytmetyczną z liczb, które przesyła do niego proces klienta. Klient przesyła liczby, które są podane jako parametry wywołania. Jeśli podana jest tylko jedna liczba, to oznacza ona liczbę liczb jakie proces klienta ma wylosować i przesłać do procesu serwera. Losowane liczby powinny być z przedziału od 1 do 100. Serwer po odebraniu ostatniej liczby zapisuje je i ich średnią w pliku o nazwie `numbers.txt` (na końcu). Średnia jest także przesyłana do procesu klienta. Klient powinien wyświetlić na standardowym wyjściu otrzymany wynik. Procesy powinny także wyświetlać odpowiednie komunikaty kontrolne na standardowym wyjściu. Zakończenie pracy procesu serwera następuje po wciśnięciu kombinacji klawiszy Ctrl-c. Komunikacja pomiędzy serwerem a klientem powinna być zrealizowana z wykorzystaniem potoków nazwanych lub kolejek komunikatów.

Zadanie 4

Napisz program do prowadzenia rozmów przez dwie osoby. Program powinien wewnętrznie wykorzystywać do pracy dwa wątki. Jeden wątek powinien oczekiwać na nowe dane z klawiatury i wprowadzone wiadomości przesyłać do rozmówcy. Drugi wątek powinien w tym samym czasie nasłuchiwać i odbierać wiadomości pisane przez rozmówcę. Wymiana danych może następować z wykorzystaniem potoków nazwanych, kolejek komunikatów lub pamięci dzielonej.

A Kompilator języka C

A.1 Wywołanie kompilatora

- `gcc/cc`
- błędy i ostrzeżenia
- numery linii
- opcja `-Wall`
- odświeżanie ekranu kombinacją Ctrl-l

A.2 Komunikaty o błędach

`error: 'x' undeclared (first use in this function)`
: Niezadeklarowana zmienna. W programie użyto zmiennej „x", która nie została jawnie zadeklarowana.

`warning: implicit declaration of function 'sleep'`
: Niejawna deklaracja funkcji `sleep`. W linii 5 odwołano się do funkcji `sleep`, która nie została dotąd zadeklarowana. Usunięcie tego ostrzeżenia wymaga dodania dyrektywy załączającej odpowiedni plik nagłówkowy. W powyższym przypadku należałoby zajrzeć do strony pomocy systemowej `sleep(3)` i sprawdzić sekcję SYNOPSIS, w której znaleźlibyśmy informację o wymaganym pliku nagłówkowym `unistd.h`. Do programu należałoby więc dodać dyrektywę: `#include <unistd.h>`.

`warning: no newline at end of file`
: Brak znaku przejścia do nowej linii na końcu pliku. Ostrzeżenie to pojawia się jeżeli stosowany jest edytor tekstowy, który nie dodaje znaku przejścia do nowej linii na końcu pliku (np. `mcedit`). W celu uniknięcia tego ostrzeżenia ostatnia linia w pliku musi być zupełnie pusta — bez spacji i znaków tabulacji.

Bibliografia

[Gra98] J. S. Gray. *Arkana: Komunikacja między procesami w Unixie*. READ ME, 1998.

[HGS99] K. Havilland, D. Gray, and B. Salama. *Unix — programowanie systemowe*. READ ME, 1999.

[JT00] M. K. Johnson and E. W. Troan. *Oprogramowanie użytkowe w systemie Linux*. WNT, 2000.

[NS99] M. Neil and R. Stones. *Linux — Programowanie*. READ ME, Warszawa, 1999.

[Roc93] M. J. Rochkind. *Programowanie w systemie UNIX dla zaawansowanych*. WNT, Warszawa, 1993.

Skorowidz