Wojskowa Akademia Techniczna

Wydział Elektroniki

Programowanie w Unix/Linux

Zadania laboratoryjne

Opracował:
mjr. dr inż. Krzysztof Maślanka

Warszawa 2019

Spis treści

Przygotowanie do ćwiczeń laboratoryjnych:

1. Pobrać, zainstalować i zapoznać się z oprogramowaniem Eclipse.
2. Zapoznać się z materiałami z wykładów.
3. Zapoznać się z podstawowymi poleceniami systemu Linux.

Sprawozdanie z ćwiczeń laboratoryjnych

1. Podczas wykonywania programów należy kod każdego nowego programu i jego modyfikację zapisywać w oddzielnych plikach. W tym celu najlepiej założyć nowy katalog dla każdego zadania z numerem zadania (np. zadanie1, zadanie2 itd.) a poszczególne modyfikacje zapisywać w plikach (katalogach jeśli wymagana jest większa liczba plików) o różnych nazwach (np. zadanie1_2.c).
2. Poszczególne wykonane zadania będą podlegały ocenie w trakcie zajęć.
3. Wszystkie kody źródłowe należy **zarchiwizować do pojedynczego pliku** z kompresją (zip) i **po rozliczeniu się z wszystkich punktów** (otrzymaniu pozytywnej oceny) umieścić w DMS.
4. Przykładowe kody na podstawie książki: M.Mitchell, J.Oldham, A.Samuel, Advanced Linux Programming.

1. Ćwiczenie 1

Celem ćwiczenia jest utrwalenie podstawowych elementów programowania w języku skryptowym Unix/Linux.

Zadanie 1. Uruchomienie skryptu

Zadanie ma na celu przygotowanie uruchomienie najprostszego możliwego skryptu z wykorzystaniem powłoki *bash*.

1. Utworzyć nowy plik we własnym katalogu założonym w katalogu domowym użytkownika.
2. Zmienić uprawnienia pliku, tak, aby użytkownik mógł go uruchamiać.
3. Umieścić kod pozwalający na wyświetlenie na ekranie komunikatu "*Witaj mój drogi stwórco!*", np.

```
#!/bin/sh

echo „Witaj moj drogi stworco”
exit 0
```

4. Uruchomić przygotowany skrypt.
5. Skopiować przygotowany skrypt. Dokonać modyfikacji tak, aby wyświetlany tekst pojawił się 10 razy .
6. Skopiować przygotowany skrypt. Zmodyfikować kod tak, aby możliwe było podanie tekstu podczas uruchamiania skryptu (jako argument) i wyświetlenie go na standardowym wyjściu.
7. Skopiować przygotowany skrypt. Zmodyfikować skrypt tak, aby w przypadku kiedy jako argument podany jest tekst: „Powtarzaj 10 razy” - wyświetlić go 10 krotnie, dodając numery kolejnych

Zadanie 2. Przetwarzanie plików

Zadanie ma na celu utrwalenie sposobu przetwarzania plików w skryptach Linux/Unix.

1. Utworzyć nowy plik. Przygotować go do uruchamiania skryptów powłoki *bash*.
2. W przygotowanym pliku zaprogramować skrypt, który zlicza linie we wszystkich plikach tekstowych znajdujących się w bieżącym lub wybranym katalogu (podanym jako argument), zlicza liczbę znaków wpisanych do pliku (przydatne polecenie: *wc*).
3. Wynik ma zostać zapisany w postaci nazwy pliku, jego właściciela oraz liczby wierszy i liczby znaków, na standardowym wyjściu oraz w pliku tekstowym o nazwie podanej jako argument.
4. Utworzyć nowy plik. Przygotować go do uruchamiania skryptów powłoki *bash*.
5. W pliku zaprogramować skrypt, który wyświetli na standardowym wyjściu wszystkie pliki z bieżącej lokalizacji, zawierające określony jako argument ciąg znaków.

Zadanie 3. Przetwarzanie tekstu

Zadanie ma na celu utrwalenie mechanizmów przetwarzania tekstu w skryptach Linux/Unix. Program ma za zadanie proste sprawdzenie wprowadzonego hasła.

1. Utworzyć nowy plik. Przygotować go do uruchamiania skryptów powłoki *bash*.
2. Napisać program, który wyświetli komunikat zachęcający do wprowadzenia hasła i odczyta wprowadzony przez użytkownika ciąg znaków.

3. Po odczytaniu sprawdzi wprowadzone hasło w następujący sposób (wykorzystać instrukcję *case*):
   - zliczy liczbę małych liter,
   - zliczy liczbę wielkich liter,
   - zliczy liczbę cyfr,
   - zliczy liczbę znaków specjalnych
4. Jeśli liczba wszystkich sprawdzanych rodzajów znaków jest większa od 0 – wyświetla komunikat o bardzo dobrej jakości haśle, w przeciwnym przypadku – wyświetla komunikat o ponownym wprowadzeniu hasła.
5. Program kończy działanie po poprawnie wprowadzonym przez użytkownika haśle.

Zadanie 4. Zwracanie wartości

Zadanie ma na celu utrwalenie mechanizmów zwracania wartości przez programy i skrypty Linux/Unix.

1. Utworzyć nowy plik. Przygotować go do uruchamiania skryptów powłoki *bash*.
2. Napisać skrypt wyświetlający pytanie rodzaj zajęć (np. czy są to zajęcia laboratoryjne).
3. Odczytać odpowiedź użytkownika w postaci jednego znaku.
4. Jeśli udzielona została odpowiedź ‘t’ lub ‘T’ skrypt ma zwracać wartość 0. Jeśli ‘n’ lub ‘N’ skrypt ma zwracać wartość 1, w pozostałych przypadkach ma zwracać wartość 2.
5. Zweryfikować wartość zwracaną przez skrypt w systemie operacyjnym. .
6. Zweryfikować zwracaną wartość przez skrypt, jeśli w pozostałych przypadkach zwracaną wartością będzie 256.

2. Ćwiczenie 2

Celem ćwiczenia jest utrwalenie podstawowych elementów programowania w języku C/C++ oraz procesu kompilacji i konsolidacji kodu wynikowego.

Zadanie 1. Uruchomienie programu

1. Utworzyć nowy plik tekstowy (źródłowy) o dowolnej nazwie z „rozszerzeniem” *.c*.
2. Wpisać do niego kod programu wyświetlającego komunikat "*Witaj mój drogi C-stwórco!*"

```
#include <stdio.h>
int main () {
    printf („Witaj moj drogi C-stworco!\n”);
    return 0;
}
```

3. Skompilować przygotowany kod poleceniem:

```
$ gcc zadanie.c -o zadanie
```

4. Uruchomić skompilowany program i sprawdzić zwracaną przez niego wartość. Zmienić wartość zwracaną przez program na różną od 0. Ponownie skompilować, uruchomić program oraz zweryfikować zwracaną wartość.

Zadanie 2. Przekazywanie parametrów do programu

1. Skopiować plik utworzony w zadaniu 1 do pliku o innej nazwie.

2. Zmodyfikować program tak, aby wyświetlał komunikat zawierający zamiast „*C-stworco*” imię podane jako argument. Jeśli nie będzie podanego argumentu program ma zakończyć działanie informując o błędzie oraz kod błędu różny od 0.
3. Zweryfikować poprawność wykonywania programu.

Zadanie 3. Kompilacja złożonego programu

1. Założyć nowy katalog. W katalogu tym utworzyć 3 pliki:
   - zadanie2_3.c zawierający kod programu obliczającego odwrotność liczby podanej jako argument

```
#include <stdio.h>
#include <stdlib.h>
#include "reciprocal.hpp"

int main (int argc, char **argv) {
  int i;

  i = atoi (argv[1]);
  printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
  return 0;
}
```

   - reciprocal.cpp – zawierający kod w języku C++ funkcji wykonującej obliczenia odwrotności podanej wartości

```
#include <cassert>
#include "reciprocal.hpp"

double reciprocal (int i) {
  // I should be non-zero.
  assert (i != 0);
  return 1.0/i;
}
```

   - reciprocal.hpp – plik nagłówkowy dla funkcji

```
#ifdef __cplusplus
extern "C" {
#endif

extern  double reciprocal (int i);

#ifdef __cplusplus
}
#endif
```

2. Skompilować przygotowane pliki źródłowe do plików obiektowych:
   - kod w języku C poleceniem (plik wynikowy będzie miał taką samą nazwę jak plik z kodem źródłowym jednakże inne rozszerzenie „.o”):

```
$ gcc -c zadanie2_3.c
```

   - kod w języku C++ poleceniem (może również zostać użyty kompilator *gcc*):

```
$ g++ -c reciprocal.cpp
```

- Domyślnie pliki nagłówkowe wyszukiwane są w bieżącym katalogu i plikach nagłówkowych standardowych bibliotek. Jeżeli pliki nagłówkowe znajdują się w systemie plików w innym miejscu dodatkowe katalogi można dołączyć za pomocą opcji „`-I`”:

```
$ g++ -c -I ../include reciprocal.cpp
```

3. Połączyć (skonsolidować) wykonane pliki nagłówkowe:

```
$ g++ -o reciprocal zadanie2_3.o reciprocal.o
```

`gdzie -o reciprocal` – nazwa pliku z programem wynikowym

4. Uruchomić i zweryfikować poprawność działania wykonanego programu.

Zadanie 4. Automatyzacja kompilacji z wykorzystaniem GNU Make

1. Przygotować plik *Makefile* dla programu z poprzedniego zadania. Przykładowy plik może wyglądać następująco (linia z regułą MUSI zaczynać się od tabulacji (TAB) inaczej *make* nie przyjmie pliku):

```
$ cat Makefile
reciprocal: main.o reciprocal.o
	g++ $(CFLAGS) -o reciprocal main.o reciprocal.o
main.o: main.c reciprocal.hpp
	gcc $(CFLAGS) -c main.c
reciprocal.o: reciprocal.cpp reciprocal.hpp
	g++ $(CFLAGS) -c reciprocal.cpp
clean:
	rm -f *.o reciprocal
```

2. Opisać zadania wykonywane w poszczególnych liniach – jako komentarz w pliku *Makefile*, który należy umieścić w archiwum sprawozdania z zajęć.
3. Skompilować program z wykorzystaniem przygotowanego pliku Makefile.

```
$ make
```

4. Dokonać modyfikacji kodu w pliku źródłowym programu głównego. Ponownie skompilować program i porównać z procesem wykonanym w poprzednim punkcie. Wnioski zapisać jako komentarz w pliku *Makefile*.
5. Wyczyścić skompilowany program (`make clean`). Ustawić wartość zmiennej CFLAGS, np. na optymalizację kodu (opcja -O2):

```
$ make CFLAGS=-O2
```

6. Porównać wynik kompilacji z kompilacjami wykonanymi poprzednio – wynik zapisać jako komentarz w pliku *Makefile*.

Zadanie 5. Procesy

1. Każdą zmianę wykonywaną w poszczególnych punktach wykonywać w nowo utworzonych plikach. Pliki utworzyć za pomocą dowolnego edytora.
2. Utworzyć proces, wprowadzić identyfikator procesu, wyświetlić jego wartość, utworzyć przykładową zmienną (np. `int test = 5`), sprawdzić poleceniem *ps* jak widziany jest proces przez system.
3. Utworzyć proces potomny do wcześniej wywołanego procesu – rozwidlenie za pomocą funkcji *fork()*, wyświetlić wartość zmiennej `test` w procesie macierzystym i potomnym (w funkcji warunkowej). Następnie zmodyfikować ich wartości i ponownie wyświetlić. Zdefiniować w obu

procesach nową zmienna (np. `int test2`) i przypisać im różne wartości. Wyświetlić wartości zmiennych obu procesów poza funkcją warunkową.

4. Utworzyć nowy proces potomny i wyświetlić wartość identyfikatora procesu.
5. Wywołać funkcją systemową polecenia *ls -l* i *ps -aux | grep nazwa programu w procesie potomnym i macierzystym*. Zmodyfikować proces potomny tak aby wykorzystane zostały razem funkcje *exec()* i *fork()*.
6. Zakończyć proces potomny a następnie macierzysty – kolejno funkcje exit() i wait().
7. Na podstawie poprzedniego programu utworzyć procesy zombie (defunct).
8. Dodać funkcję czyszczącą procesy potomne.

Wywołanie funkcji systemowej

```
#include <stdlib.h>
int main () {
    int return_value;
    return_value = system ("ls -l /");
    return return_value;
}
```

Użycie funkcji fork

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main () {
pid_t child_pid;
printf ("the main program process ID is %d\n", (int) getpid ());
child_pid = fork ();
if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int)getpid ());
    printf ("the child's process ID is %d\n", (int) child_pid);
} else
   printf ("this is the child process, with id %d\n", (int) getpid ());
return 0;
}
```

Użycie funkcji fork i exec

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

int spawn (char* program, char** arg_list) {
    pid_t child_pid;
    /* Duplicate this process. */
    child_pid = fork ();
    if (child_pid != 0)
        /* This is the parent process. */
        return child_pid;
    else {
        /* Now execute PROGRAM, searching for it in the path. */
        execvp (program, arg_list);
        /* The execvp function returns only if an error occurs. */
        fprintf (stderr, "an error occurred in execvp\n");
```

```
        abort ();
    }
}
int main () {
    /* The argument list to pass to the "ls" command. */
    char* arg_list[] = {
         "ls", /* argv[0], the name of the program. */
         "-l",
         "/",
        NULL /* The argument list must end with a NULL. */
    };
    spawn ("ls", arg_list);
    printf ("done with main program\n");
    return 0;
}
```

Użycie funkcji czyszczenia potomków.

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
sig_atomic_t child_exit_status;
void clean_up_child_process (int signal_number) {
    int status;
    wait (&status);
    child_exit_status = status;
}
int main () {
    struct sigaction sigchld_action;
    memset (&sigchld_action, 0, sizeof (sigchld_action));
    sigchld_action.sa_handler = &clean_up_child_process;
    sigaction (SIGCHLD, &sigchld_action, NULL);
    return 0;
}
```

Pomocne funkcje

- system()
- getpid()
- fork()
- exec()
- instrukcje warunkowe if/else lub switch/case

3. Ćwiczenie 3

Zadanie 1. Tworzenie wątku

1. Każdą zmianę wykonywaną w poszczególnych punktach wykonywać w nowo utworzonych plikach. Pliki utworzyć za pomocą dowolnego edytora.
2. Utworzyć nowy program wyświetlający znak 'o' na standardowym wyjściu błędów.
3. Dodać do utworzonego wcześniej programu wątek, który będzie wyświetlał 'x' na standardowym wyjściu błędów (przykład kodu poniżej). W celu korzystania w programach z wątkami należy dołączyć bibliotekę <pthread.h> i umożliwić włączanie funkcji obsługi wątków biblioteki libpthread dodając w opcjach linkera -lpthread.

4. Na podstawie wyników działania programu opisać w komentarzu do kodu czy możliwe jest określenie kiedy zostaną wyświetlone poszczególne znaki z uzasadnieniem.

Zadanie 2. Przekazywanie danych do wątków

1. Każdą zmianę wykonywaną w poszczególnych punktach wykonywać w nowo utworzonych plikach. Pliki utworzyć za pomocą dowolnego edytora.
2. Utworzyć nowy program wyświetlający określoną liczbę przekazanego jako argument znaku standardowym wyjściu błędów. Do wyświetlenia znaku przygotować odpowiednią funkcję.
3. Dodać drugi wątek, który wyświetlał będzie inny znak wprowadzony jako argument – również określoną (inną niż poprzednio) liczbę razy.
4. Wykonać analizę wyników działania program, której wynik zapisać jako komentarz w pliku z kodem. Czy jest możliwe, że funkcja główna programu zakończy działanie przed jednym z utworzonych wątków?
5. Zmodyfikować wykonany program tak aby wykorzystać możliwość łączenia wątków. Wyjaśnić cel stosowania łączenia wątków oraz różnicę jaka jest w wynikach działania programu z wykorzystaniem łączenia wątków.

Zadanie 3. Zwracanie wartości z wątków

1. Utworzyć nowy program. Napisać funkcję obliczającą liczbę pierwszą o określonym numerze przekazywanym jako argument (kod funkcji w przykładowych kodach). Funkcja ma zwracać wynik obliczeń.
2. Uruchomić napisaną funkcję w wątku – określić, którą liczbę pierwszą program ma obliczyć.
3. Zapewnić aby wątek wykonał się do końca (łączenie wątków).
4. Wyświetlić wynik obliczeń.

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

void* print_xs (void* unused) {
      while (1)
      fputc (‘x’, stderr);
      return NULL;
}

int main () {
      pthread_t thread_id;

      pthread_create (&thread_id, NULL, &print_xs, NULL);
      while (1)
            fputc (‘o’, stderr);
      return 0;
}
```

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

struct char_print_parms {
      char character;
      int count;
};
void* char_print (void* parameters) {
      struct char_print_parms* p = (struct char_print_parms*)
parameters;
      int i;
      for (i = 0; i < p->count; ++i)
            fputc (p->character, stderr);
      return NULL;
}

int main () {
      pthread_t thread1_id;
      pthread_t thread2_id;
      struct char_print_parms thread1_args;
      struct char_print_parms thread2_args;

      thread1_args.character = 'x';
      thread1_args.count = 30000;
      pthread_create (&thread1_id, NULL, &char_print,
      &thread1_args);

      thread2_args.character = 'o';
      thread2_args.count = 20000;
      pthread_create (&thread2_id, NULL, &char_print,
      &thread2_args);
      return 0;
}
```

```
int main () {
      pthread_t thread1_id;
      pthread_t thread2_id;
      struct char_print_parms thread1_args;
      struct char_print_parms thread2_args;

      thread1_args.character = 'x';
      thread1_args.count = 30000;
      pthread_create (&thread1_id, NULL, &char_print,
      &thread1_args);

      thread2_args.character = 'o';
      thread2_args.count = 20000;
      pthread_create (&thread2_id, NULL, &char_print,
      &thread2_args);

      pthread_join (thread1_id, NULL);
      pthread_join (thread2_id, NULL);
      return 0;
}
```

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

void* compute_prime (void* arg) {
      int candidate = 2;
      int n = *((int*) arg);
      while (1) {
            int factor;
            int is_prime = 1;
            /* Test primality by successive division. */
            for (factor = 2; factor < candidate; ++factor)
                  if (candidate % factor == 0) {
                        is_prime = 0;
                  break;
                  }
            if (is_prime) {
                  if (--n == 0)
                        return (void*) candidate;
            }
            ++candidate;
      }
      return NULL;
}

int main () {
      pthread_t thread;
      int which_prime = 5000;
      int prime;

      pthread_create (&thread, NULL, &compute_prime,
      &which_prime);
      pthread_join (thread, (void*) &prime);
      printf("The %dth prime number is %d.\n", which_prime,
      prime);
      return 0;
}
```

4. Projekt

Zaprogramuj system transakcyjny pozwalający na obsługę kont bankowych użytkowników.

Wymagania:

1. System musi być wyposażony w oprogramowanie serwera obsługujące konta użytkowników. (3)
2. System musi być wyposażony w oprogramowanie klienckie pozwalające użytkownikowi na łączenie się z serwerem wyświetlanie danych konta i wykonywanie określonych operacji. (3)
3. Oprogramowanie klienta i serwera muszą pracować pod kontrolą systemu operacyjnego Linux lub Unix. (3)
4. Każdy zalogowany użytkownik musi być obsługiwany przez oddzielny wątek lub proces w serwerze. (3)
5. System musi umożliwiać weryfikację konta użytkownika. (3)
6. Serwer musi zapisywać wszystkie wykonywane operacje w pliku dziennika (log). (3)
7. System musi umożliwiać oprogramowaniu klienta dwie operacje – zwiększanie i zmniejszanie środków na jego „koncie”. (3)
8. System powinien umożliwiać logowanie się użytkownikowi z wielu miejsc jednocześnie. (4)
9. System powinien weryfikować możliwość zmniejszania środków tak, by ich stan nie został zmniejszony poniżej 0. (4)
10. System powinien wyświetlać stan konta użytkownika po zalogowaniu. (4)
11. Klient powinien mieć możliwość wyświetlenia dostępnych operacji dla użytkownika (pomoc). (4)
12. System może obliczać założone odsetki dla każdego użytkownika (ustawiane podczas definiowania użytkownika) i obliczać je co określony czas (definiowany dla serwera), zapisywać wyniki obliczeń w logu oraz wyświetlać je użytkownikowi po zalogowaniu do serwera. (5)
13. System może przekazywać nazwę użytkownika oraz jego hasło w zabezpieczony sposób. (5)