Politechnika Wrocławska

Programowanie Systemowe

LABORATORIUM

Temat: **Programy wielowątkowe i komunikacja sieciowa**

Hanna Kieszek – 283797

22.01.2026

1. Cel ćwiczenia

Celem ćwiczenia było utworzenie programu wielowątkowego korzystającego z komunikacji sieciowej do przesyłania danych wejściowych.

2. Wykorzystane narzędzia

1. Ubuntu
2. Terminal

3. Przebieg laboratorium

3.1. Zmiana danych wejściowych w Makefile i ogólny zamysł programu

Plik Makefile posłuży do skompilowania plików które zostały utworzone podczas tego laboratorium. Poniżej znajduje się zmodyfikowany Makefile:

```
  GNU nano 7.2                          Makefile
OUTDIR=build9
SRCDIR=laboratorium9
CC=gcc
TARGET=$(OUTDIR)/plik
SRC=$(wildcard $(SRCDIR)/*.c)
OBJ=$(patsubst $(SRCDIR)/%.c, $(OUTDIR)/%.o, $(SRC))

all: $(TARGET)

clean:
        rm -rf $(OUTTDIR)

$(TARGET): $(OBJ)
        $(CC) $^ -o $@

$(OUTDIR)/%.o: $(SRCDIR)/%.c | $(OUTDIR)
        $(CC) -c $< -o $@

$(OUTDIR):
        mkdir $@
```

Obraz 1: Wygląd pliku Makefile

Ogólna zasada działania skompilowanego programu polega na tym, że klient łączy się z serwerem TCP za pomocą komendy nc, a następnie przesyła do niego tekst wprowadzony przez użytkownika. Serwer gdy odbierze dane, przesyła je dalej do workera który został uruchomiony jako wątek, worker modyfikuje tekst zamieniając jego pierwsza literę na ‘X’ a następnie odsyła z powrotem do

serwera. Serwer odbiera dane i przesyła do klienta który te dane wyświetla. Gdy zostało wprowadzone słowo ‘exit’ program się kończy. Jest to program wielowątkowy który korzysta z pamięci współdzielonej, a do synchronizacji dostępu do danych używa mechanizmów mutex. Zostało utworzone 6 plików: main.c, server.c, worker.c, server.h, worker.h oraz shared.h który został stworzony na rzecz tego laboratorium i odpowiada on za zdefiniowanie struktury danych współdzielonych pomiędzy wątkami programu.

3.2. Plik shared.h

Jest to plik w którym została opisana zawartość pamięci współdzielonej i wygląda on następująco:

```
 GNU nano 7.2                    laboratorium9/shared.h
#ifndef SHARED_H
#define SHARED_H

#include <pthread.h>

typedef struct {
    char buf[100];
    int data_ready;
    int exit_flag;

    pthread_mutex_t mutex;
    pthread_cond_t cond;
} Shared;

#endif
```

Obraz 2: Wygląd pliku shred.h

Dyrektywy na samym początku pliku zabezpieczają proces kompilacji przed tym aby ten plik nie został dołączony ponownie jeżeli już wcześniej został dołączony. Następnie został dodany nagłówek *pthread.h* który jest niezbędny do prawidłowego działania.

W następnej kolejności zaczyna się sekcja *typedef struct* w której została zdefiniowana struktura danych Shared dla pamięci współdzielonej:

char buf[100] - bufor przeznaczony do przechowywania tekstu
int data_ready - zmienna informująca o stanie bufora, która przyjmuje wartości 0 (brak danych do przetworzenia) i 1 (nowe dane do przetworzenia)
int exit_flag - zmienna sygnalizująca o zakończeniu programu
pthread_mutex_t mutex - mutex który zapewnia kontrolowany dostęp do bufora przez wątki programu, odpowiada za to, że w tym samym czasie tylko jeden wątek ma dostęp do medium
pthread_cond_t cond - zmienna która umożliwia wątkom czekanie na to, aż jakis warunek zostanie spełniony

3.3. Pliki worker.h i server.h

Pliki worker.h oraz server.h odpowiadają za deklarację funkcji worker_thread(), która jest funkcja wątku roboczego oraz funkcji start_server(), która jest funkcją odpowiedzialna za działanie serwera TCP.

```
 GNU nano 7.2                    laboratorium9/worker.h
#ifndef WORKER_H
#define WORKER_H

#include "shared.h"

void* worker_thread(void* arg);

#endif
```

Obraz 3: Wygląd pliku worker.h

```
 GNU nano 7.2                    laboratorium9/server.h
#ifndef SERVER_H
#define SERVER_H

#include "shared.h"

void start_server(Shared* s);

#endif
```

Obraz 4: Wygląd pliku server.h

Funkcja start_server() przyjmuje jako swój argument wskaźnik do struktury danych Shared, a funkcja worker_thread() przyjmuje jako argument wskaźnik void*, który w pliku worker.c zostanie zamieniony na wskaźnik do struktury Shared. Nie można od razu przekazać wskaźnika ze względu na ograniczenia funkcji pthread_create która wymaga argumentu void*.

3.4. Plik main.c

Plik main.c jest głównym plikiem programu który rozpoczyna jego działanie, odpowiada za stworzenie wątków, uruchomienie funkcji server oraz zamknięcie programu.

```
 GNU nano 7.2                    laboratorium9/main.c
#include "server.h"
#include "worker.h"
#include <pthread.h>
#include "shared.h"

int main() {
    pthread_t worker;
    Shared shared = {
        .data_ready = 0,
        .exit_flag = 0,
        .mutex = PTHREAD_MUTEX_INITIALIZER,
        .cond = PTHREAD_COND_INITIALIZER
    };

    pthread_create(&worker, NULL, worker_thread, &shared);
    start_server(&shared);
    pthread_join(worker, NULL);

    return 0;
}
```

Obraz 5: Wygląd pliku main.c

Na początku programu zostały dołączone niezbędne nagłówki aby program działał poprawnie, a następnie rozpoczyna się funkcja main(). Zostaje zadeklarowany identyfikator wątku worker za pomocą *pthread_t* oraz zmienna Shared która przechowuje dane współdzielone między wątkami oraz przypisuje im wartości początkowe. Następnie za pomocą funkcji *pthread_create* zostaje utworzony wątek roboczy z dołączonym wskaźnikiem do struktury Shared. W następnej linijce uruchamiany jest serwer z argumentem wskaźnika do Shared. Jeżeli serwer zakończy swoje działanie, funkcja *pthread_join()* czeka na zakończenie wątku worker a następnie odłącza go.

3.5. Plik worker.c

Plik worker.c zawiera definicję funkcji worker_thread() która jest uruchamiana w programie jako wątek roboczy którego zadaniem jest modyfikacja danych wejściowych z bufora.

Na początku pliku zostały zamieszczone niezbędne nagłówki aby program działał prawidłowo, a następnie została zdefiniowana funkcja worker_thread(). Rozpoczyna się ona od określenia argumentu funkcji jako wskaźnik do struktury Shared. Następnie została zdefiniowana zmienna local_buf[100] która odpowiada za lokalny bufor. W kolejnej linijce rozpoczyna się pętla while w której na początku zostaje zablokowany mutex za pomocą funkcji *pthread_mutex_lock()*. W kolejnej pętli while() program sprawdza czy dane są gotowe do przetworzenia, jeżeli nie program czeka. Jest to możliwe dzięki funkcji *pthread_cond_wait()* która sprawia, że wątek podczas oczekiwania nie zajmuje procesora. Gdy dane są gotowe, zostają one skopiowane do bufora lokalnego. Następnie badana jest ich treść, to znaczy czy nie zostało wpisane słowo “exit” które jest sygnałem do zakończenia programu, jeżeli tak, ustawia exit_flag na 1. Jeżeli nie, to tekst jest

modyfikowany zamieniając pierwszą literę na ‘X’ i przesyłany do bufora. Następnie worker ustawia zmienna data_ready na 0 co oznacza, że nie ma już danych które powinny zostać zmodyfikowane, wysyła sygnał do serwera aby go uaktywnić oraz zwalnia mutex. Na końcu znajduje się instrukcja warunkowa odpowiadająca za zakończenie programu jeżeli exit_flag jest równa 1.

```
 GNU nano 7.2                    laboratorium9/worker.c
#include <string.h>
#include "worker.h"

void* worker_thread(void* arg) {
    Shared* s = (Shared*)arg;
    char local_buf[100];

    while (1) {
        pthread_mutex_lock(&s->mutex);
        while (!s->data_ready)
            pthread_cond_wait(&s->cond, &s->mutex);

        strcpy(local_buf, s->buf);

        if (strcmp(local_buf, "exit") == 0) {
            s->exit_flag = 1;
        } else {
            local_buf[0] = 'X';
            strcpy(s->buf, local_buf);
        }

        s->data_ready = 0;
        pthread_cond_signal(&s->cond);
        pthread_mutex_unlock(&s->mutex);

        if (s->exit_flag)
            break;
    }
    return NULL;
}
```

Obraz 6: Wygląd pliku worker.c

3.6. Plik server.c

Plik server.c zawiera definicję funkcji start_server(), która odpowiada za komunikację sieciową, obsługuje serwer TCP oraz za wymianę danych z wątkiem roboczym.

GNU nano 7.2 laboratorium9/server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include "server.h"

void start_server(Shared* s) {
    int server_fd, client_fd;
    struct sockaddr_in addr;
    char buf[100];
    ssize_t n;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(1234);
    addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(server_fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        perror("bind");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 1) < 0) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Serwer działa na porcie 1234\n");

    client_fd = accept(server_fd, NULL, NULL);
    if (client_fd < 0) {
        perror("accept");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
```

Obraz 7: Wygląd pliku server.c cz.1

```
    client_fd = accept(server_fd, NULL, NULL);
    if (client_fd < 0) {
        perror("accept");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    while (1) {
        n = read(client_fd, buf, sizeof(buf));
        if (n <= 0) {
            perror("read");
            break;
        }
        buf[strcspn(buf, "\r\n")] = 0;

        pthread_mutex_lock(&s->mutex);
        strcpy(s->buf, buf);
        s->data_ready = 1;
        pthread_cond_signal(&s->cond);
        pthread_mutex_unlock(&s->mutex);

        pthread_mutex_lock(&s->mutex);
        while (s->data_ready)
            pthread_cond_wait(&s->cond, &s->mutex);

        strcpy(buf, s->buf);
        pthread_mutex_unlock(&s->mutex);

        n = write(client_fd, buf, strlen(buf));
        if (n < 0) {
            perror("write");
            break;
        }

        n = write(client_fd, "\n", 1);
        if (n < 0){
            perror("write");
            break;
        }
        if (strcmp(buf, "exit") == 0)
            break;
    }

    close(client_fd);
    close(server_fd);
}
```

Obraz 8: Wygląd pliku server.c cz.2

Na początku pliku zostały zdefiniowane potrzebne nagłówki do poprawnego działania programu. Następnie rozpoczyna się definicja funkcji start_server() która jako argument przyjmuje wskaźnik do pamięci współdzielonej Shared. Zostały zadeklarowane zmienne:

int server_fd, client_fd - gniazdo serwera, gniazdo dla połączenia z klientem
struct sockaddr_in addr - struktura dla adresu IP oraz portu serwera
char buf[100] - bufor
ssize_t n - zmienna zawierająca wielkość danych w bajtach

W następnej kolejności, za pomocą funkcji *socket()* zostało utworzone gniazdo sieciowe typu TCP (*SOCK_STREAM*) o domenie IPv4 (*AF_INET*). Linijkę niżej znajduje się instrukcja warunkowa odpowiedzialna za sprawdzenie czy gniazdo zostało poprawnie utworzone. Następnie został skonfigurowany adres serwera:

addr.sin_family = AF_INET - ustawia rodzinę adresów IPv4
addr.sin_port = htons(1234) - ustawia numer portu na którym działa serwer
addr.sin_addr.s_addr = INADDR_ANY - definiuje z jakich interfejsów sieciowych serwer akceptuje połączenia, w tym przypadku ze wszystkich

Następnie znajdują się dwie instrukcje warunkowe. Pierwsza z nich przypisuje gniazdo do adresu IP oraz zdefiniowanego portu za pomocą funkcji *bind(),* a w razie niepowodzenia obsługuje błąd. Druga instrukcja warunkowa jest odpowiedzialna za rozpoczęcie nasłuchiwania połączeń przez serwer, a jeżeli się to nie uda, obsługuje błąd. Następnie jeżeli instrukcje warunkowe nie zwróciły błędu, program wyświetla informację o tym na jakim porcie działa serwer i czeka na połączenie a kiedy nadejdzie akceptuje je za co odpowiada funkcja *accept().* W kolejnej linijce serwer obsługuje błąd gdyby połączenie się nie udało. Następnie rozpoczyna się pętla while() jeżeli połączenie zostało nawiązane. Na samym jej początku serwer odczytuje dane przesłane przez klienta za pomocą funkcji *read(),* po czym została zawarta obsługa błędu na wypadek gdyby połączenie między klientem a serwerem zostało przerwane. Następnie program usuwa znaki nowej linii z końca odebranego napisu, co umożliwia poprawne porównywanie łańcuchów znaków w przypadku słowa kończącego ‘exit’.

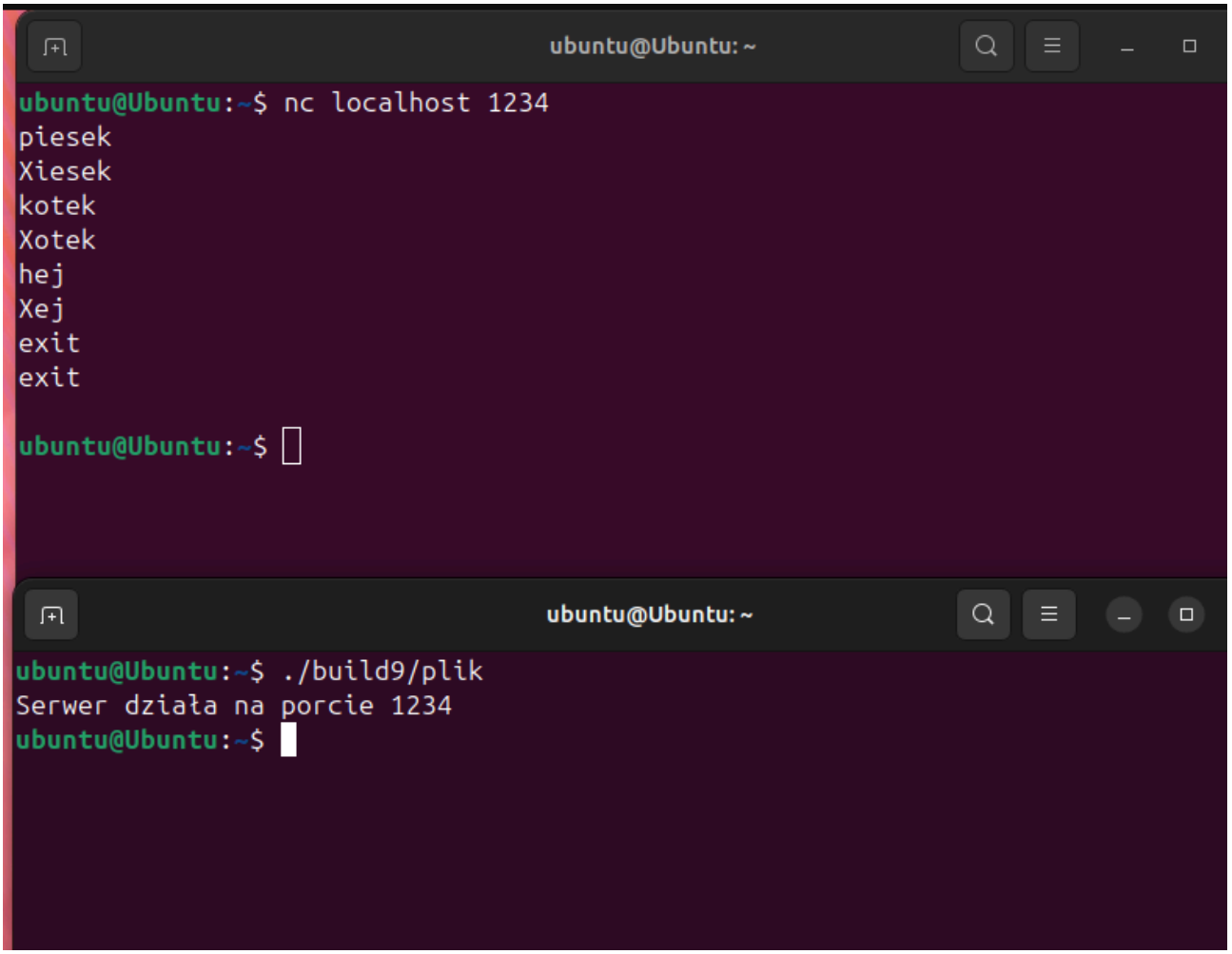
W kolejnym kroku serwer rozpoczyna przekazywanie danych do workera. Blokuje mutex przed dostępem do pamięci współdzielonej za pomocą funkcji *pthread_mutex_lock()*, kopiuje dane przesłane przez klienta do bufora pamięci współdzielonej za pomocą funkcji *strcpy()*, zmienia wartość zmiennej data_ready na 1 aby poinformować workera, że dane są gotowe do przetwarzania, za pomocą funkcji *pthread_cond_signal()* serwer odblokowuje wątek roboczy wysyłając mu sygnał aby rozpoczął swoje działanie, a na końcu odblokowuje mutex funkcją *pthread_mutex_unlock()*.

Następnie serwer czeka aż worker zakończy swoją pracę i ustawi flagę data_ready na 0. Kiedy to się stanie, serwer wznawia swoje działanie i kopiuje zawartość bufora do lokalnego bufora buf. Następnie odblokowuje mutex i wysyła dane do klienta za pomocą funkcji *write().* Instrukcja warunkowa po tej linijce sprawdza czy zapis do gniazda się powiódł. Następnie serwer wysyła do klienta znak nowej linii aby output był czytelny a następnie zostaje sprawdzony ten sam błąd co powyżej.

W kolejnej linijce znajduje się instrukcja warunkowa która sprawdza czy nie zostało wpisane słowo kończące ‘exit’. Jeżeli tak pętla się kończy a następnie program zamyka połączenie z klientem oraz gniazdo serwera za pomocą funkcji *close()*.

3.7. Działanie programu

Najpierw został uruchomiony skompilowany plik, a kiedy serwer był już dostępny i został wyświetlony komunikat, na drugim Terminalu zostało nawiązane z nim połączenie za pomocą komendy *nc localhost 1234*.



Obraz 9: Terminale po prawidłowym zakończeniu działania programu