Tomasz Ligęza

Programowanie równoległe. Przetwarzanie równoległe i rozproszone.

Sprawozdanie z laboratorium 4.

Cel zajęć:

- opanowanie umiejętności pisania programów z synchronizacją wątków.

W ramach zajęć zrealizowałem następujące kroki:

1. Utworzyłem katalog roboczy lab_4,
2. Zabezpieczyłem program pub_sym_1.c przed możliwością wystąpienia race condition przy pomocy muteksów,
3. Dodałem sprawdzenie warunku czy na końcu liczba kufli zgadza się z ilością na początku,
4. Sprawdziłem działanie programu bez muteksów i z muteksami. W pierwszym przypadku pojawiały się błędne ilości kuflóœ po zakończeniu pracy programu, natomiast w drugiej problem ten już nie występował,
5. Rozwiązałem problem z klientami pobierającymi kufle mimo ich braku. W momencie pobierania kufla najpierw blokujemy muteks, następnie sprawdzamy czy ilość kufli jest większa od zera (jeśli tak, to pobieramy kufel) i odblokowujemy muteks.

Kod programu pub_sym_1.c

```
#define ILE_MUSZE_WYPIC 1000

int liczba_wolnych_kufli;
//inicjalizacja muteksa w przestrzeni globalnej
pthread_mutex_t straznik_kufli = PTHREAD_COND_INITIALIZER;
int l_kf;

void *watek_klient(void *arg);

int main(void) {

   pthread_t *tab_klient;
   int *tab_klient_id;
   int l_kl, l_kr, i;
   printf("\nLiczba klientow: ");
   scanf("%d", &l_kl);
   printf("\nLiczba kufli: ");
   scanf("%d", &l_kf);
   pthread_mutex_init(&straznik_kufli, NULL);
   liczba_wolnych_kufli = l_kf;

   l_kr = 1000000000; // wystarczająco dużo, żeby nie było rywalizacji

   tab_klient = (pthread_t *) malloc(l_kl * sizeof(pthread_t));
```

```
    tab_klient_id = (int *) malloc(l_kl * sizeof(int));
    for (i = 0; i < l_kl; i++) tab_klient_id[i] = i;

    printf("\nOtwieramy pub (simple)!\n");
    printf("\nLiczba kufli %d\n", l_kf);
    printf("\nLiczba wolnych kufli %d\n", liczba_wolnych_kufli);

    for (i = 0; i < l_kl; i++) {
        pthread_create(&tab_klient[i], NULL, watek_klient, &tab_klient_id[i]);
    }
    for (i = 0; i < l_kl; i++) {
        pthread_join(tab_klient[i], NULL);
    }
    printf("\nZamykamy pub!\n");

    printf("\nLiczba kufli %d\n", l_kf);
    printf("\nLiczba wolnych kufli %d\n", liczba_wolnych_kufli);
    if (l_kf != liczba_wolnych_kufli) {
        printf("Blad! Rozna liczba kufli na starcie i na koncu pracy programu.\n");
        exit(-1);
    }
}

void *watek_klient(void *arg_wsk) {

    int moj_id = *((int *) arg_wsk);
    int i, j, kufel, result;
    int ile_musze_wypic = ILE_MUSZE_WYPIC;
     for (i = 0; i < ile_musze_wypic; i++) {
//       wybieranie kufla
        int success = 0;
        do {
            while (pthread_mutex_trylock(&straznik_kufli) != 0) {}
            if (liczba_wolnych_kufli > 0) {
                --liczba_wolnych_kufli;
                success = 1;
            }
            pthread_mutex_unlock(&straznik_kufli);
        } while (success == 0);
        j = 0;
//       odkladanie kufla
        while (pthread_mutex_trylock(&straznik_kufli) != 0) {}
        ++liczba_wolnych_kufli;
        pthread_mutex_unlock(&straznik_kufli);
    }
     return (NULL);
}
```

Odpowiedzi na pytania:

1. Jaka najprostsza reprezentacja pozwala na rozwiązanie problemu bezpiecznego korzystania z kufli w pubie w przypadku liczby kufli większej od liczby klientów (jeden kufel jest posiadany tylko przez jednego klienta)?

Najprostszym sposobem jest lockowanie muteksa przy interakcji z kuflami. Nie ma potrzeby sprawdzania warunku czy jakikolwiek kufel jest jeszcze dostępny skoro jest ich więcej niż klientów.

2. Jak wygląda rozwiązanie problemu bezpiecznego korzystania z kufli? Jak połączyć je ze sprawdzeniem dostępności kufli? Jaka jest wada rozwiązania z wykorzystaniem tylko muteksów?
Należy sprawdzać dostępność kufli zaraz po zalockowaniu muteksów. W przypadku dostępnego kufla należy go pobrać i odlokować, w przeciwnym wypadku odlokować muteks i ponowić sprawdzanie warunku.

Wadą jest niska wydajność.

Zrzuty ekranu:
Wersja z muteksami z zapewnionym bezpieczeństwem korzystania z kufli:

```
-> % ./pub_sym_1

Liczba klientow: 100

Liczba kufli: 1000

Otwieramy pub (simple)!

Liczba kufli 1000

Liczba wolnych kufli 1000

Zamykamy pub!

Liczba kufli 1000

Liczba wolnych kufli 1000
```

Ostateczna wersja sprawdzająca dostępność kufli i zapewniająca bezpieczeństwo przy korzystaniu z kufla:

```
wigryz@msi-wigryz [18:09:21]
-> % ./pub_sym_1

Liczba klientow: 100

Liczba kufli: 2

Otwieramy pub (simple)!

Liczba kufli 2

Liczba wolnych kufli 2

Zamykamy pub!

Liczba kufli 2

Liczba wolnych kufli 2
```

Wnioski:

Dobre zarządzanie muteksami pozwala osiągnąć odporność na błędy synchronizacji, jednocześnie wystawiając nas na możliwe problemu z optymalizacją. Ostateczna wersja programu zapewniająca ochronę przed każdym możliwym problemem (nad którymi zastanawialiśmy się w trakcie laboratorium) była niewiarygodnie powolna. Kombinacja 1000 kufli do wypicia, 1000 klientów oraz 10 kufli uniemożliwiała zakończenie pracy programu w rozsądnym czasie.