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. Sprawdziłęm 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.

1. Jak wygląda rozwiązanie problemu bezpiecznego korzystania z kufli? Jak połączyć je

ze sprawdzeniem dostępności kufli? Jaka jest wada rozwiązania z wykorzystaniem tylko

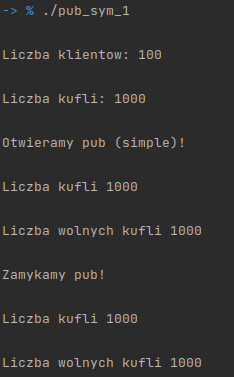
muteksów?

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.

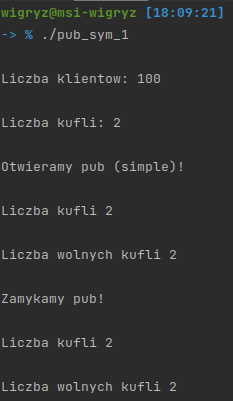
Wadą jest niska wydajność.

Zrzuty ekranu:

Wersja z muteksami z zapewnionym bezpieczeństwem korzystania z kufli:



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



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.