Wojciech Celej
Nr. albumu: 271248

Ćwiczenie 3
Synchronizacja procesów z wykorzystaniem semaforów

I. Zadanie do zrealizowania

Bufor n-elementowy FIFO. Jest jeden producent i trzech konsumentów (A, B, C). Producent produkuje jeden element, jeżeli jest miejsce w buforze. Element jest usuwany z bufora, jeżeli zostanie przeczytany przez albo obu konsumentów A i B, albo przez obu konsumentów B i C. Konsument A nie może przeczytać elementu, jeżeli został on już przez niego wcześniej przeczytany, albo został przeczytany przez konsumenta C i na odwrót. Konsument B również nie może wielokrotnie czytać tego samego elementu. Ponadto, żaden konsument nie może dwa razy z rzędu usunąć elementu z bufora.

II. Proponowane rozwiązanie

1. Kolejka

Będzie to tablica o wymiarze N. Zaimplementowana będzie w pamięci współdzielonej procesów. Rozmiar tablicy będzie zdefiniowany jako makro.

- usunięcie elementu będzie polegało na zwiększeniu indeksu el. początkowego o 1 (gdy dojdzie do ostatniego indeksu, wtedy przeskoczy na 0)
- dodanie nowego elementu będzie polegało na zwiększeniu indeksu el. początkowego o 1 (gdy dojdzie do ostatniego indeksu, wtedy przeskoczy na 0)
- semafory zliczające zapobiegną przepełnieniu tej tablicy

2. Semafory

Zaimplementowane w pamięci współdzielonej:

- 2 semafory zliczające: *empty* i *full* – zapobiegające przed zapisywaniem do pełnego bufora i czytaniem i usuwaniem z pustego (*empty* zabezpiecza przed pisaniem do pełnego bufora, *full* zapobiega usuwaniu z pustego bufora)
- 3 semafory binarne: *mutex, read* i *remove:*
    - *mutex* – dzięki niemu czytanie, usuwanie bądź dodawania elementu nie nastąpi jednocześnie
    - *read* – chroni ustawianie flag z grupy READ
    - *remove* – zapewnia, żeby procesy usuwające nie przeszkadzały sobie i chroni flagi z grupy FLAG

3. Pamięć współdzielona

W pamięci współdzielonej zaimplementowana zostanie tablica n-elementów buffora. Oprócz niej znajdzie się również miejsce na tablicę znaczników. Będzie to tablica 9-elementowa z przypisanymi odpowiednio makrami dla poprawy czytelności. Jej zawartość to:

- indeksy początku i końca bufora
- 3 flagi z grupy FLAG – jeśli np. proces A ostatni usuwał element, to ma ustawioną flagę FLAG_A na 1, reszta flag z tej grupy wtedy na 0
- 3 flagi z grupy READ – ustawiona na 1, jeżeli dany konsument odczytał dany element (po usunięciu elementu flagi z tej grupy są zerowane)

Odpowiednie flagi zdefiniowane będą przez makra (dla poprawy czytelności).

4. Potrzebne funkcje (wywołania) systemowe

- int semget(key_t key, int nsems, int semflg) – utworzenie nowego semafora (zwracana wartość to identyfikator semafora dla danego procesu; key – identyfikator globalny, jeśli wcześniej nie był wywoływany z taką wartością, to utworzy nowy semafor, w przeciwnym razie zwróci identyfikator do danego semafora dla wołającego procesu, nsems – liczba tworzonych semaforów, semflags – odpowiednie flagi (musimy ustawić IPC_CREAT))

- int semop(int semid, struct sembuf *sops, size_t nsops) – operacje na semaforze (dekrementacja i ew. czekanie; inkrementacja)

- int semctl(int semid, int semnum, int cmd, ...) – operacja kontrolne na semaforze – funkcja z odpowiednimi argumentami posłuży do zainicjowania wartości semafora, pobrania wartości semafora bądź jego usunięcia

- int shmget(key_t key, size_t size, int shmflg) – pobranie pamięci współdzielonej (działanie analogiczne jak w semget, pole size to rozmiar alokowanej pamięci, shmflg – odpowiednie flagi)

- void *shmat(int shmid, const void *shmaddr, int shmflg) – funkcja dołączająca pamięć współdzieloną do przestrzeni adresowej procesu wołającego (zwraca wskaźnik na adres tej pamięci)

- int shmctl(int shmid, int cmd, struct shmid_ds *buf) – operacja na pamięci współdzielonej (możemy np. usunąć dany segment z odpowiednią flagą IPC_RMID)

Definicje funkcji zaczerpnąłem z dokumentacji Linuxa (*Ubuntu, komedna man ___*)

5. Struktura programu

Odpowiednie 4 procesy będą tworzone przez kopiowanie procesu macierzystego (nadzorcy). Potrzebne będą makra definiujące pola *key,* definiujące jednoznacznie odpowiednie segmenty w obrębie pamięci współdzielonej dla komunikujących się procesów a także odpowiednie semafory.
Zakończenie pracy programy następuje wtedy, kiedy producent wyprodukuje wszystko to, co miał wyprodukować. Wysyła wtedy sygnały zabicia procesów konsumentów (ma ich PIDy poprzez fork()) i dealokuje pamięć i semafory.

Potrzebne funkcje pomocnicze: wstawianie/usuwanie elementu oraz oczekiwanie przez losowy czas.

6. Pseudokody odpowiednich procesów:

```
while(TRUE)
{
        wait_rand_time();
        produce_item();
        down(empty);
        lock(mutex);
        enter_item();
        unlock(mutex);
        up(full);
}

// CONSUMER A
while(TRUE)
{
        wait_rand_time();
        down(full);
        lock(read);
        if(readenA == 0 && readenC==0)
        {
                lock(mutex);
                read_item();
                readenA=1;
                unlock(mutex);
        }
        unlock(read);
        up(full);
        lock(remove)
        if(flagA != 1)
        {
                lock(read);
                if((readenA == 1 && readenB == 1)||(readenB == 1 && readen C ==1))
                {
                        down(full);
                        lock(mutex);
                        remove_item();
```

```
                        unlock(mutex);
                        up(empty);
                        readenA=0;
                        readenB=0;
                        readenC=0;
                        unlock(read);
                        flagA=1;
                        flagB=0;
                        flagC=0;
                }
                else unlock(read);
        }
        unlock(remove);
}

// CONSUMER B
while(TRUE)
{
        wait_rand_time();
        down(full);
        lock(read);
        if(readenB == 0)
        {
                lock(mutex);
                read_item();
                readenB=1;
                unlock(mutex);
        }
        unlock(read);
        up(full);
        lock(remove)
        if(flagB != 1)
        {
                lock(read);
                if((readenA == 1 && readenB == 1)||(readenB == 1 && readen C ==1))
                {
                        down(full);
                        lock(mutex);
                        remove_item();
                        unlock(mutex);
                        up(empty);
                        readenA=0;
                        readenB=0;
                        readenC=0;
                        unlock(read);
                        flagA=0;
                        flagB=1;
                        flagC=0;
                }
                else unlock(read);
        }
        unlock(remove);
```

```
}
// CONSUMER C
while(TRUE)
{
        wait_rand_time();
        down(full);
        lock(read);
        if(readenC == 0 && readenA ==0)
        {
                lock(mutex);
                read_item();
                readenC=1;
                unlock(mutex);
        }
        unlock(read);
        up(full);
        lock(remove)
        if(flagC != 1)
        {
                lock(read);
                if((readenA == 1 && readenB == 1)||(readenB == 1 && readen C ==1))
                {
                        down(full);
                        lock(mutex);
                        remove_item();
                        unlock(mutex);
                        up(empty);
                        readenA=0;
                        readenB=0;
                        readenC=0;
                        unlock(read);
                        flagA=0;
                        flagB=0;
                        flagC=1;
                }
                else unlock(read);
        }
        unlock(remove);
}
```