Systemy Operacyjne 1
Laboratorium 5
„Komunikacja IPC - semafory”
(jeden tydzień)

dr inż. Arkadiusz Chrobot
dr inż. Grzegorz Łukawski

24 października 2024

Wstęp

Ta instrukcja dotyczy kolejnego typu zasobów w IPC (ang. *InterProcess Communication*) - semaforów. Nie są one łączem komunikacyjnym, ale stanowią mechanizm synchronizacji procesów współbieżnych, które ze sobą współpracują. Jest to bardzo ważne i istotne zagadnienie. W środowisku współbieżnym nie da się dokładnie przewidzieć w jakiej kolejności procesy będą się wykonywały, ani który proces w danej chwili będzie korzystał z procesora lub innego zasobu. Jeśli te procesy korzystają ze wspólnego zasobu lub wielu takich zasobów, to mogą pojawić się *sytuacje hazardowe*, czyli zdarzenia polegające na przykład na tym, że jeden z procesów rozpocznie operację odczytu zasobu, zanim inny skończy operację modyfikacji. To prowadzi nie tylko do błędnych wyników działania obu procesów, ale może też spowodować nieprawidłowy stan końcowy współdzielonego zasobu. Aby uniknąć sytuacji hazardowych należy zadbać, aby żaden proces nie mógł rozpocząć operacji odczytu zasobu zanim inny proces nie zakończy jego modyfikacji i odwrotnie. Do tego służą właśnie semafory. Należy zauważyć, że operacje modyfikacji (tj. zapisu) zasobu współdzielonego musi odbywać się w trybie wyłączności, tzn. żadna inna taka operacja dotycząca tego samego zasobu, ani operacja odczytu związana z tym zasobem nie może się rozpocząć, dopóki ta operacja się nie zakończy. Z kolei na takim zasobie można współbieżnie wykonywać kilka operacji odczytu, ale dopóki się one nie zakończą nie może rozpocząć się żadna operacja modyfikacji tego zasobu.

W pierwszej części instrukcji objaśnione jest działanie semaforów. Druga część poświęcona jest API semaforów IPC w Linuksie. Instrukcja kończy się listą zadań do samodzielnego wykonania w ramach zajęć laboratoryjnych.

1. Semafory

Semafor jest abstrakcyjnym typem danych, dla którego zdefiniowano dwie niepodzielne operacje: `P` - *oczekuj* i `V` - *sygnalizuj*. Rozważmy najprostszy rodzaj semafora, czyli *semafor binarny*. Jest to zmienna przyjmująca tylko dwie wartości: `0` i `1`, dostępna tylko poprzez operacje `V` i `P`. Operacja `P` działa następująco: jeśli wartość semafora jest większa od zera, to zmniejsz ją o jeden. Jeśli wartość ta wynosi zero, to zawieś wykonanie procesu. Wykonanie operacji `V` przebiega w ten sposób: jeśli inny proces został zawieszony w oczekiwaniu na semafor, to wznów jego wykonanie. Jeśli żaden proces nie został zawieszony w oczekiwaniu na semafor, to zwiększ wartość semafora o jeden. Z powyższego opisu można wyciągnąć dwa wnioski. Po pierwsze semafor musi być dostępny dla wszystkich procesów ubiegających się o dostęp do zasobu strzeżonego przez ten semafor, a więc nie może być zmienną umieszczoną w przestrzeni adresowej do której dostęp ma tylko jeden proces. Po drugie zmiana wartości semafora musi się odbywać w sposób niepodzielny, tzn. jeśli proces rozpocznie operacje zmiany wartości semafora, to nie może zostać wywłaszczony i żaden inny proces w tym samym przedziale czasu nie może manipulować semaforem. Aby obydwa wymagania były spełnione semafory są tworzone w przestrzeni adresowej jądra i dostępne poprzez odpowiednie wywołania systemowe. Mechanizm IPC dostarcza interfejsu dla tych wywołań systemowych dla procesów pracujących w przestrzeni użytkownika. Semafory w Linuksie (Uniksie) mogą przyjmować większy zbiór wartości niż tylko `0` i `1`. Należy zadbać, aby wszystkie procesy, których działanie chcemy synchronizować przestrzegały odpowiedniego protokołu dostępu do zasobu (tzn. nie wolno dopuścić do sytuacji, w której proces uzyskuje dostęp do zasobu z pominięciem semafora lub kiedy nieodpowiednio posługuje się semaforem).

2. API semaforów

System Linux udostępnia szereg funkcji służący do obsługi semaforów z poziomu języka C. Ten rozdział stanowi krótki ich opis:

`semget()` funkcja ta tworzy zbiór semaforów i zwraca jego identyfikator lub zwraca identyfikator istniejącego zbioru semaforów, w zależności od przekazanych jej w wywołaniu flag. Jako argumenty wywołania przyjmuje klucz, który może być zwrócony przez funkcję `ftok()` (patrz poprzednia instrukcja) lub stałą `IPC_PRIVATE`, liczbę semaforów w zbiorze i flagi związane ze sposobem tworzenia oraz powiązane z prawami dostępu do semafora.
Szczegóły: `man semget`

`semop()` umożliwia przeprowadzenie operacji na wartości semafora w sposób niepodzielny. Funkcja ta pobiera trzy argumenty. Pierwszym argumentem przyjmowanym przez tę funkcję jest identyfikator zbioru semaforów, zwrócony przez `semget()`. Następnym jest wskaźnik na tablicę zawierającą elementy o strukturze przedstawionej w listingu 1.

```
1 struct sembuf {
2         unsigned sem_num;
3         short sem_op;
4         short sem_flg;
5 };
```

Listing 1: Struktura opisująca operację na semaforze

Pole `sem_num` zawiera numer semafora w zbiorze (semafory są numerowane od zera), którego ma dotyczyć operacja. Pole `sem_op` określa jaka operacja zostanie na semaforze przeprowadzona. Jeśli wartość tego pola jest dodatnia, to zostanie ona dodana do bieżącej wartości semafora. Jeśli wartość ta wynosi zero, to proces wykonujący tę operację będzie czekał do czasu aż semafor osiągnie wartość zero. Jeśli wartość pola `sem_op` jest ujemna, to proces będzie czekał do momentu kiedy semafor osiągnie wartość większą lub równą bezwzględnej wartości pola `sem_op`, a następnie dodaje tą ujemną wartość do bieżącej wartości semafora. Pole `sem_flg` może przyjmować dwie wartości `SEM_UNDO` i `IPC_NOWAIT`. Ostatnia flaga oznacza, że proces nie będzie czekał na zakończenie operacji, natomiast pierwsza oznacza, że operacja zostanie automatycznie cofnięta po zakończeniu procesu, który ją wykonał. Trzeci argument funkcji `semop()` określa ile jest elementów w tablicy, której wskaźnik jest przekazywany jako drugi argument wywołania funkcji.
Szczegóły: `man semop`

`semctl()` służy do sterowania zbiorem semaforów. Jako pierwszy argument pobiera ona identyfikator zbioru semaforów. Drugim argumentem jest numer semafora w zbiorze (patrz opis funkcji `semop()`). Trzeci argument określa rodzaj operacji jaka ma być wykonana:

`IPC_RMID` powoduje usunięcie zbioru semaforów z systemu (drugi argument jest ignorowany),

`GETVAL` powoduje, ze wywołanie funkcji `semctl()` zwróci wartość określonego w jej argumentach semafora,

`SETVAL` może być użyta do zainicjowania semafora określoną wartością. Jej użycie wymaga przekazania do funkcji `semctl()` czwartego argumentu o typie określonym przez unię z listingu 2,

`IPC_STAT` pobiera informacje statystyczne o zbiorze semaforów. Jej użycie wymaga przekazania do funkcji `semctl()` czwartego argumentu o typie określonym przez unię z listingu 2,

`SETALL` ustawia wartości dla wszystkich semaforów w zbiorze. Jej użycie wymaga przekazania do funkcji `semctl()` czwartego argumentu o typie określonym przez unię z listingu 2,

`GETALL` pobiera wartości wszystkich semaforów w zbiorze. Jej użycie wymaga przekazania do funkcji `semctl()` czwartego argumentu o typie określonym przez unię z listingu 2,

```c
union semun {
        int val;
        struct semid_ds  *buff;
        unsigned short   *array;
        struct seminfo *__buf;
} arg;
```

Listing 2: Unia `semun`

Pierwsze pole unii `semun` z listingu 2 jest wykorzystywane przez operację `SETVAL` i zawiera nową wartość dla semafora, drugie pole jest wykorzystywane przez `IPC_STAT` i powinno zawierać w takim przypadku adres struktury typu `struct semid_ds`, trzecie pole jest używane przez operacje `SETALL` i `GETALL` i powinno zawierać adres tablicy z której `SETALL` pobierze wartości dla wszystkich semaforów w zbiorze, a `GETALL` zapisze bieżące wartości semaforów.

Uwaga! Operacje SETVAL, GETVAL, SETALL i GETALL nie są wykonywane w sposób niepodzielny i nie powinny być wykorzystywane do innych operacji niż nadanie wartości początkowej semaforom i pobranie o nich informacji w celach statystycznych. W szczególności nie powinny one być używane przez procesy współbieżne na równi z operacjami wykonywanymi za pomocą funkcji `semop()`.
Szczegóły: `man semctl`

Z poziomu powłoki systemowej można uzyskać informacje o zbiorach semaforów za pomocą polecenia `ipcs`[1], a usuwać je można za pomocą `ipcrm`[2].

2.1. Przykład

Listingi 3 i 4 zawierają kody źródłowe dwóch programów. Ten z listingu 3 tworzy zbiór semaforów składający się z jednego semafora, nadaje temu semaforowi wartość różną od zera, a następnie czeka, aż drugi z procesów go wyzeruje. Program z listingu 4 uzyskuje dostęp do semafora i go zeruje, tym samym pozwalając pierwszemu programowi się zakończyć.

[1] Pomocna jest zwłaszcza opcja `-s`
[2] Tutaj także pomocna jest opcja `-s`

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

void up_and_wait(int semset_id)
{
        struct sembuf up_operation = {0,1,0};
        struct sembuf wait_operation = {0,0,0};
        char error_message[15];

        if(semop(semset_id,&up_operation,1)<0) {
                sprintf(error_message,"semop %u",__LINE__-1);
                perror(error_message);
        }
        if(semop(semset_id,&wait_operation,1)<0) {
                sprintf(error_message,"semop %u",__LINE__-1);
                perror(error_message);
        }
}

int main(void)
{
        int key = ftok("/tmp",8);
        if(key<0)
                perror("ftok");
        int semset_id = semget(key,1,0600|IPC_CREAT|IPC_EXCL);
        if(semset_id<0)
                perror("semget");
        up_and_wait(semset_id);
        if(semctl(semset_id,0,IPC_RMID)<0)
                perror("semctl");
        return 0;
}

```

Listing 3: Program, który tworzy semafor, inicjuje go jedynką i czeka na jego wyzerowanie

Proszę zwrócić w programie z listingu 3 szczególną uwagę na funkcję `up_and_wait()`. W niej oprogramowano dwie operacje na semaforze. Pierwsza opisana jest strukturą o nazwie `up_operation`, której polom przypisywane są wartości: `{0,1,0}`. Pierwsza liczba oznacza numer semafora w zbiorze, druga wartość operacji, która na nim ma być wykonana, a trzecia to wartości flag (żadna flaga nie jest ustawiona). Ta operacja zwiększy wartość semafora o jeden. Druga operacja opisana jest strukturą tego samego typu, ale o nazwie `wait_operation`. Jej polom przypisywane są same wartości zerowe, co oznacza, że program będzie czekał tak długo, aż pierwszy (i jedyny) semafor się wyzeruje. Proszę zwrócić uwagę, że każda z tych operacji jest wykonywana przez osobne wywołanie funkcji `semop()`. Pojawia się pytanie, czy nie lepiej byłoby zrobić tablicę dwóch takich struktur i wykonać obie operacje przy pomocy jednego wywołania `semop()`. Niestety, taki zapis nie byłby równoważny temu, który został wykorzystany w bieżącej wersji programu. W jego przypadku obie operacje zostałby potraktowane jako jedna całość, a w związku z tym programu z listingu 4 nie byłby w stanie nigdy wykonać swojej operacji. Jeśli są one rozdzielone, to najpierw zostanie zmieniona wartość semafora, a dopiero potem program będzie oczekiwał na jego wyzerowanie. W ten sposób drugi program będzie miał szansę go wyzerować. Proszę zauważyć, że choć drugim argumentem wywołania `semop()` powinna być tablica struktur opisujących operację na semafo-

rze, to do niej przekazać pojedynczą taką strukturę, podając jej adres. Aby rozróżnić ewentualne wyjątki pochodzące od dwóch wywołań `semop()` wykorzystujemy wyrażenie z makrem `__LINE__` do dodania do nazwy funkcji numeru wiersza, w którym ta funkcja jest wywołana. Taki ciąg znaków przekazywany jest do wywołania `perror()`.

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

void down(int semset_id)
{
        struct sembuf down_operation = {0,-1,0};

        if(semop(semset_id,&down_operation,1)<0)
                perror("semop");
}

int main(void)
{
        int key = ftok("/tmp",8);
        if(key<0)
                perror("ftok");
        int semset_id = semget(key,1,0600);
        if(semset_id<0)
                perror("semget");
        down(semset_id);
        return 0;
}
```

Listing 4: Program, który uzyskuje dostęp do semafora i go zeruje

Kod źródłowy programu z listingu 4 jest zbliżony do tego z listingu 3. Na uwagę zasługuje inicjacja struktury `down_operation` z funkcji `down()`. Wartość `-1` operacji oznacza, że będzie ona odjęta od wartości semafora i w przypadku obu zaprezentowanych programów oznacza, że ten semafor zostanie wyzerowany.

Zadania

UWAGA: WE WSZYSTKICH PROGRAMACH TUŻ PRZED ZAKOŃCZENIEM ICH DZIAŁANIA WSZYSTKIE ZBIORY SEMAFORÓW Z JAKICH ONE KORZYSTAJĄ POWINNY ZOSTAĆ USUNIĘTE. PROGRAMY MUSZĄ BYĆ NAPISANE Z PODZIAŁEM NA FUNKCJE Z PARAMETRAMI ORAZ MUSZĄ SPRAWDZAĆ, CZY WYWOŁYWANE PRZEZ NIE FUNKCJE Z API SYSTEMU OPERACYJNEGO NIE SYGNALIZUJĄ WYJĄTKÓW.

1. Napisz program, w którym dwa spokrewnione procesy, które będą korzystały z prywatnego semafora o wartości początkowej 1 i po 50 razy wykonają następujące czynności:
   (a) zaczekają, aż semafor będzie miał wartość jeden,
   (b) wypiszą na ekranie czy są procesem rodzicielskim, czy potomnym,
   (c) zwiększą wartość semafora o jeden.
2. Napisz program podobny do tego z zadania 1 ale tak, aby procesy naprzemiennie wypisywały swoje komunikaty (najpierw rodzicielski, potem potomny lub odwrotnie). Możesz dokonać dowolnych

modyfikacji (np. zwiększyć liczbę semaforów, zastosować różne wartości semafora), pod warunkiem, że współbieżne operacje na semaforze lub semaforach będą wykonywane w sposób niepodzielny.

3. Napisz program, który stworzy zbiór dziesięciu semaforów, o wartości początkowej równej jeden i dziesięć procesów potomnych, które wstępnie zostaną uśpione na sekundę, a następnie ustawią wartość odpowiadającego im semafora na zero. Proces rodzicielski może się zakończyć dopiero wtedy, kiedy wszystkie semafory ze zbioru osiągną wartość zero i wszystkie procesy potomne zakończą swą pracę.

4. Problem jest podobny do tego z poprzedniego zadania, ale tym razem należy stworzyć jeden semafor o wartości początkowej zero i pięć procesów potomnych, które będą zwiększały go o jeden w określonym porządku: proces pierwszy sprawdzi, czy semafor ma wartość zero i zamieni ją na jeden, proces drugi poczeka, aż semafor będzie miał wartość jeden i zwiększy ją do dwóch, itd. Proces macierzysty może się zakończyć wtedy, gdy semafor osiągnie wartość pięć, a procesy potomne skończą swoją pracę.

5. Zademonstruj synchronizację operacji zapisu i odczytu dla łącza nazwanego, przy pomocy semaforów.

6. Pokaż w jaki sposób może dojść do zakleszczenia (ang. *deadlock*) procesów synchronizowanych przy pomocy semaforów.

7. Zademonstruj działanie flagi `SEM_UNDO`.

8. Zademonstruj działanie operacji `SETALL`, `GETALL` i `IPC_STAT`.

9. Stwórz kolejkę komunikatów, z której będą korzystały dwa procesy. Jeden będzie producentem komunikatów, a drugi konsumentem. W funkcjach operujących na kolejce użyj flagi `IPC_NOWAIT`. Do synchronizacji pracy procesów, tak aby nie sygnalizowały one błędów kiedy kolejka jest pusta lub pełna, użyj semaforów.

10. Napisz program, który podzieli się na dwa procesy komunikujące się przez łącze nienazwane (ang. *pipe*). Oba te procesy będą również miały dostęp do wspólnego semafora, któremu będzie nadana wartość początkowa większa od zera. Proces pierwszy będzie wysyłał liczby od `1` do `10` do procesu drugiego. Proces drugi będzie wyświetlał je na ekran, a po otrzymaniu liczby `10` wyzeruje semafor. Po wyzerowaniu semafora oba procesy powinny się zakończyć.