Systemy Operacyjne 1
Laboratorium 6
„Komunikacja IPC - pamięć dzielona”
(jeden tydzień)

dr inż. Arkadiusz Chrobot
dr inż. Grzegorz Łukawski

8 listopada 2019

Wstęp

W tej instrukcji zawarto informacje na temat pamięci dzielonej, która umożliwia komunikację między dwoma lub większą liczbą procesów. W pierwszej części opisano zasadę działania pamięci dzielonej. Druga część zawiera opis API tej formy komunikacji między procesami. Zawarto w niej również przykład komunikacji opartej na schemacie czytelników i pisarzy, w której zastosowano pamięć dzieloną. Instrukcja kończy się listą zadań do samodzielnego wykonania w ramach zajęć laboratoryjnych.

1. Pamięć dzielona

Pamięć dzielona (ang. *shared memory*), nazywana również w literaturze polskiej pamięcią wspólną lub pamięcią współdzieloną, służy do szybkiej komunikacji między procesami. Szybkość zyskiwana jest dzięki ograniczeniu do minimum roli jądra w obsłudze wymiany danych, co pozwala zaoszczędzić czas związany z kopiowaniem między kontekstami (proces użytkownika - jądro - proces użytkownika). Zasada działania tego środka komunikacji opiera się na pomyśle współdzielenia przez procesy pewnego obszaru pamięci w przestrzeni adresowej użytkownika. Zwykle przestrzenie adresowe procesów są dokładnie odseparowane, tak aby procesy nie mogły wzajemnie uszkodzić się nadpisując sobie dane lub kod. Dzięki odpowiednim wywołaniom systemowym procesy użytkownika mogą zażądać przydzielenia im przez jądro fragmentu pamięci, który będą mogły współdzielić, tzn. każdy z nich będzie miał do niej dostęp ze swojej przestrzeni adresowej. W takim przypadku system nie kontroluje operacji, które są wykonywane w tej pamięci, a dba jedynie, aby żaden z komunikujących się procesów nie próbował odczytywać lub zapisywać pamięci leżącej poza wyznaczonym obszarem. Zaletą takiego rozwiązania jest szybkość działania i łatwość obsługi (procesy mogą korzystać z tej pamięci w taki sam sposób, jak z pamięci która jest im przydzielana przez funkcję `malloc()`), natomiast wadą jest brak synchronizacji komunikacji, o którą musi zadbać programista aplikacji, przy użyciu semaforów lub innych środków synchronizacji.

2. API pamięci dzielonej

Pamięć dzielona obsługiwana jest za pomocą następujących funkcji:

`shmget()` funkcja ta w zależności od flag przekazanych jej jako ostatni argument jej wywołania tworzy nowy obszar pamięci dzielonej i zwraca jego identyfikator lub zwraca identyfikator pamięci już istniejącej. Przyjmuje ona trzy argumenty wywołania. Pierwszym jest klucz, który może być wygenerowany przez funkcję `ftok()` lub w przypadku, gdy z pamięci będą korzystały procesy spokrewnione, może być stałą `IPC_PRIVATE`. Drugim argumentem jest wielkość tworzonego obszaru pamięci dzielonej wyrażona w bajtach[1]. Wartość trzeciego argumentu może wynosić zero lub być sumą logiczną stałych `IPC_CREAT`, `IPC_EXCL` i praw dostępu do tworzonego obszaru pamięci dzielonej. Funkcja w przypadku niepowodzenia zwraca wartość `-1`.
Szczegóły: `man shmget`

`shmat()` funkcja ta przyłącza obszar pamięci dzielonej, określony poprzez identyfikator podany jej jako pierwszy argument wywołania, do przestrzeni adresowej procesu, który ją wywołał. Jako drugi argument jest pobierany przez tę funkcję adres początkowy od którego ma być dołączona pamięć dzielona. Jeśli jest on równy `NULL`, to system wybierze dowolny nieużywany adres, natomiast, jeśli ma on określoną inną wartość, a jako trzeci argument została przekazana stała `SHM_RND`, to adres pod który będzie przyłączona pamięć dzielona równy będzie adresowi podanemu w argumencie wywołania zaokrąglonemu w dół do wielokrotności wartości stałej `SHMLBA` (obecnie ta stała jest równa stałej `PAGE_SIZE`, czyli w przypadku komputerów z procesorami o architekturze x86 wynosi 4K lub inaczej 4096). Oprócz opisanej powyżej wartości trzeci argument, czyli flagi może również przyjąć wartość `SHM_RDONLY`, która oznacza, że przyłączony obszar pamięci dzielonej będzie przeznaczony tylko do czytania. Jeśli wywołanie funkcji się powiedzie to zwróci ona adres pod którym będzie dołączona pamięć dzielona, w przeciwnym przypadku zwróci wartość `-1`.
Szczegóły: `man shmat`

[1] Funkcja zaokrągla tę wielkość do wielokrotności rozmiaru strony pamięci.

shmdt() funkcja ta odłącza obszar pamięci dzielonej o podanym w argumencie adresie od przestrzeni adresowej procesu, który ją wywołał. Jeśli wywołanie zakończy się sukcesem zwracane jest zero, w przeciwnym przypadku `-1`.
Szczegóły: `man shmdt`

shmctl() funkcja zarządzająca pamięcią dzieloną. Obszar pamięci dzielonej na którym ma zostać wykonana operacja jest określony poprzez identyfikator podany w pierwszym argumencie wywołania funkcji. Drugim argumentem jest rodzaj operacji:

`IPC_STAT` pozwala na uzyskanie informacji o wskazanym obszarze. Informacje te zapisywane są w zmiennej typu `struct shmid_ds`, której adres musi być przekazany jako trzeci argument wywołania funkcji.

`IPC_SET` pozwala zmodyfikować prawa dostępu związane z danym obszarem pamięci dzielonej, te prawa muszą być zapisane w polu `shm_perm` struktury `shmid_ds`, które jest typu `struct ipc_perm`. Adres zmiennej typu `shmid_ds` musi być przekazany jako ostatni argument wywołania funkcji.

`IPC_RMID` powoduje usunięcie podanego obszaru pamięci dzielonej. Trzeci argument funkcji jest ignorowany, więc jego wartość może wynosić `0` lub `NULL`.

Pozostałe operacje albo są dostępne z poziomu użytkownika uprzywilejowanego i tylko w systemie Linux (`SHM_LOCK` i `SHM_UNLOCK`), albo mogą zostać w przyszłości zmodyfikowane lub usunięte (`SHM_STAT`, `SHM_INFO`, `IPC_INFO`).
Szczegóły: `man shmctl`

Z poziomu powłoki systemowej można uzyskać informacje o pamięci dzielonej za pomocą polecenia `ipcs`[2], a usuwać je można za pomocą `ipcrm`[3].

2.1. Przykład

Listing 1 zawiera kod źródłowy programu, który tworzy procesy czytelników i pisarzy komunikujących się z użyciem pamięci dzielonej.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<unistd.h>
4  #include<time.h>
5  #include<wait.h>
6  #include<string.h>
7  #include<sys/types.h>
8  #include<sys/ipc.h>
9  #include<sys/sem.h>
10 #include<sys/shm.h>
11
12 #define NR_PROC 16
13
14 union semun {
15         int val;
16         struct semid_ds *buf;
17         unsigned short *array;
18         struct seminfo *__buf;
19 } semval;
```

[2]Pomocna jest zwłaszcza opcja `-m`
[3]Tutaj także pomocna jest opcja `-m`

```
20  struct shdata {
21          int readers_number;
22          char message[100];
23  };
24
25  void writer(int id, int mux, int shmid)
26  {
27          struct sembuf down = {0,-1,0},
28                        up = {0,1,0};
29          char name[512];
30          struct shdata *shm = (struct shdata *)shmat(shmid,NULL,0);
31          if(shm==(void *)-1)
32                  perror("writer shmat");
33          if(semop(mux,&down,1)<0)
34                  perror("writer semop down");
35          snprintf(name,512,"Jestem pisarzem nr %d\n",id);
36          write(STDOUT_FILENO,name,strlen(name));
37          snprintf(shm->message,4096,"Wiadomość od pisarza nr %d\n",id);
38          if(semop(mux,&up,1)<0)
39                  perror("writer semop up");
40          if(shmdt(shm))<0)
41                  perror("writer shmdt");
42  }
43
44  void reader(int id, int mux, int shmid)
45  {
46          struct sembuf down0 = {0,-1,0},
47                        up0 = {0,1,0},
48                        down1 = {1,-1,0},
49                        up1 = {1,1,0};
50
51          char name[512];
52          struct shdata *shm = (struct shdata *)shmat(shmid,NULL,0);
53          if(shm==(void *)-1)
54                  perror("reader shmat");
55          if(semop(mux,&down1,1)<0)
56                  perror("reader semop 1 down1");
57          shm->readers_number++;
58          if(shm->readers_number==1)
59                  if(semop(mux,&down0,1)<0)
60                          perror("reader semop down0");
61          if(semop(mux,&up1,1)<0)
62                  perror("reader semop 1 up1");
63          snprintf(name,512,"Jestem czytelnikiem nr %d\n",id);
64          write(STDOUT_FILENO,name,strlen(name));
65          write(STDOUT_FILENO,shm->message,sizeof(shm->message));
66          if(semop(mux,&down1,1)<0)
67                  perror("reader semop 2 down1");
68          shm->readers_number--;
69          if(!shm->readers_number)
70                  if(semop(mux,&up0,1)<0)
71                          perror("reader semop up0");
```

```
        if(semop(mux,&up1,1)<0)
                perror("reader semop 2 up1");
        if(shmdt(shm)<0)
                perror("reader shmdt");
}

int main(void)
{
        int reader_id=0, writer_id=0;
        int mux = semget(IPC_PRIVATE,2,0600|IPC_CREAT|IPC_EXCL);
        if(mux<0)
                perror("semget");

        semval.val = 1;

        if(semctl(mux,0,SETVAL,semval)<0)
                perror("semctl 0");

        if(semctl(mux,1,SETVAL,semval)<0)
                perror("semctl 1");

        int shmid = shmget(IPC_PRIVATE,4096,0600|IPC_CREAT|IPC_EXCL);
        if(shmid<0)
                perror("shmget");

        srand(time(NULL));
        int i;
        for(i=0;i<NR_PROC;i++) {
                int choice = rand()%2;
                if(choice)
                        reader_id++;
                else
                        writer_id++;
                int pid = fork();
                if(pid==-1)
                        perror("fork");
                if(pid==0) {
                        if(choice)
                                reader(reader_id,mux,shmid);
                        else
                                writer(writer_id,mux,shmid);
                        return 0;
                }
        }

        for(i=0;i<NR_PROC;i++)
                if(wait(NULL)<0)
                        perror("wait");

        if(shmctl(shmid,IPC_RMID,NULL)<0)
                perror("shmctl");
        if(semctl(mux,0,IPC_RMID,NULL)<0)
                perror("semctl IPC_RMID");
```

```
127         return 0;
128 }
```

Listing 1: Program, ilustrujący problem czytelników i pisarzy

Program z listingu 1 rozwiązuje problem czytelników i pisarzy, gdzie czytelnicy są uprzywilejowani. Przypomnijmy, że jeśli na zasobie współdzielonym jest wykonywana wyłącznie operacja odczytu, to jest to działanie bezpieczne, a więc pamięć dzieloną może współbieżnie odczytywać wiele procesów, ale zapisywać może w danym czasie tylko jeden proces i dodatkowo w czasie tej modyfikacji nie może być wykonywany żaden odczyt. W wierszach 20-23 kodu programu jest zdefiniowana struktura danych, które będą współdzielone przez czytelników i pisarzy. Do jej ochrony służą dwa semafory, pracujące jako muteksy. Pierwszy chroni pole `message` zawierające wiadomość zostawianą przez pisarzy dla czytelników. Drugi chroni pole `readers_number`, które zlicza liczbę czytelników odczytujących wiadomość. Oba semafory będą miały wartość początkową `1`, o co zadba funkcja `main()` (wiersze 87-94).

Kod realizowany przez proces pisarza jest całkowicie zawarty w funkcji `writer()`. Ten proces najpierw przyłącza segment pamięci dzielonej do swojej przestrzeni adresowej (wiersz nr 30). Typ tego segmentu będzie określony struktura `struct shdata`. Proszę zwrócić uwagę na sposób sprawdzania, czy funkcja `shmat()` zwróciła wartość `-1` (wiersz 31). Po przyłączeniu segmentu pamięci dzielonej proces próbuje zająć pierwszy semafor. Jeśli mu się to uda, to wypisuje na ekran komunikat (wiersze 35 i 36), używając do tego funkcji `write()`, gdyż w przeciwieństwie do `printf()` nie buforuje ona informacji, którą ma wypisać na ekranie. Jest to istotne w przypadku współbieżnego wypisywania komunikatów przez procesy. Następnie pisarz zapisuje informację dla czytelnika w pamięci dzielonej (wiersz nr 37) i zwalnia semafor (wiersz nr 38) oraz odłącza pamięć dzieloną (wiersz nr 40).

Kod czytelnika zapisany jest w funkcji `reader()`. Ten proces najpierw przyłącza pamięć dzieloną do swojej przestrzeni adresowej (wiersz 53), następnie próbuje zająć drugi semafor. Jeśli mu się to uda, to zwiększa wartość pola `readers_number` o jeden. Jeśli wartość wynikowa tego pola wynosi `1`, to znaczy, że jest on pierwszym czytelnikiem, który uzyskał dostęp do pamięci dzielonej i dlatego próbuje zająć także pierwszy semafor (wiersz nr 60). Po zakończeniu tych czynności zwalnia drugi semafor, wypisuje na ekranie swoją wiadomość oraz wiadomość odczytaną z pola `message` dzielonej struktury (wiersze 64--66). Po tym ponownie zajmuje drugi semafor, zmniejsza wartość pola `readers_number` i bada, czy jego wartość wynikowa wynosi zero. Jeśli tak, to oznacza to, że jest on ostatnim czytelnikiem korzystającym z pamięci dzielonej, a więc zwalnia on pierwszy semafor, a na końcu drugi i odłącza segment pamięci dzielonej.

W funkcji `main()` programu tworzony jest prywatny obszar pamięci dzielonej (wiersz nr 95) oraz zbiór semaforów, które będą go chronić (wiersz nr 83). Następnie program tworzy `16` procesów potomnych, przy czym losowo wybiera rolę dla każdego z nich - czytelnik lub pisarz. Proces macierzysty czeka na ich zakończenie (wiersze 119-121), a następnie usuwa wszystkie wykorzystywane przez siebie zasoby `IPC` i kończy swoje działanie.

Proszę zwrócić uwagę, że kolejność wchodzenia procesów do sekcji krytycznej nie jest wymuszana w tym programie, więc może się zdarzyć, że czytelnicy otrzymają dostęp do pamięci dzielonej przed wszystkimi pisarzami i odczytają pustą wiadomość lub, że pisarz nadpisze wiadomość zostawioną przez innego pisarza, zanim odczyta ją którykolwiek z czytelników.

Zadania

UWAGA: WE WSZYSTKICH PROGRAMACH TUŻ PRZED ZAKOŃCZENIEM ICH DZIAŁANIA WSZYSTKIE ZBIORY SEMAFORÓW I OBSZARY PAMIĘCI DZIELONEJ, 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. Zademonstruj użycie przez program prywatnego obszaru pamięci dzielonej.
2. Stwórz obszar pamięci dzielonej, z którego będą korzystać trzy procesy. Zorganizuj dostęp do tej pamięci tak, aby procesy mogły z niej korzystać w ściśle określonej kolejności. Postaraj się nie używać semaforów.

3. Pokaż rozwiązanie problemu czytelników i pisarzy, w którym pisarze i czytelnicy wchodzą do sekcji krytycznej w określonej kolejności, tzn. zanim czytelnicy nie odczytają wiadomości, pisarz nie może zapisać kolejnej. Do wypisywania na ekran komunikatów zamiast funkcji `printf()` użyj funkcji `write()` (Szczegóły: `man 2 write`).

4. Pokaż rozwiązanie problemu producenta i konsumenta na przykładzie operacji zapisu i odczytu do pamięci dzielonej. Do synchronizacji komunikacji użyj semaforów. Do wypisywania na ekran komunikatów zamiast funkcji `printf()` użyj funkcji `write()` (Szczegóły: `man 2 write`).

5. Napisz dwa programy, które będą komunikować się poprzez pamięć dzieloną. W trakcie działania programów (np. po dwóch wysłanych i odebranych komunikatach) niech program będący właścicielem pamięci zamieni jej prawa dostępu, tak, aby tylko on mógł z niej korzystać. Pokaż co się wtedy stanie.

6. Zademonstruj działanie flagi `SHM_RDONLY`.

7. Stwórz trzy procesy, które będą wymieniały między sobą dane poprzez dwa obszary pamięci dzielonej (np. niech proces drugi pośredniczy w wymianie danych pomiędzy pierwszym i drugim obszarem pamięci dzielonej).

8. Użyj flagi `IPC_STAT` dla funkcji `shmctl()` celem uzyskania informacji na temat używanego przez proces/procesy obszaru pamięci dzielonej.

9. Napisz program, który stworzy dwa spokrewnione procesy, które prześlą między sobą 10 komunikatów przez kolejkę komunikatów, a następnie stworzy dwa kolejne procesy, które prześlą te same komunikaty za pomocą pamięci dzielonej. W obu przypadkach dokonaj pomiaru czasu przesyłania komunikatów i wyświetl otrzymane wyniki. Do pomiaru czasu użyj funkcji `clock_gettime()` (Szczegóły: `man 3 clock_gettime`).