6 Pamięć dzielona standardu POSIX

6.1 Wprowadzenie

Pamięć dzielona, inaczej wspólna, (ang. *shared memory*) jest zasobem umożliwiającym najszybszy sposób komunikacji między procesami, z reguły wymagający jednak dodatkowego mechanizmu synchronizacji. Podobnie jak w przypadku semaforów, w nowszych wersjach systemów uniksowych (np. w Linuksie od wersji jądra 2.4 z biblioteką glibc od wersji 2.2) dostępna jest pamięć dzielona standardu POSIX. Ma ona prostszy i wygodniejszy interfejs niż pamięć dzielona standardu UNIX System V, opisana w dodatku B.

Schemat korzystania z pamięci dzielonej standardu POSIX jest następujący. Najpierw jeden z procesów tworzy obiekt pamięci dzielonej ustawiając do niego prawa dostępu, a następnie ustawia jego rozmiar. Kolejny proces może uzyskać dostęp do tego obiektu poprzez jego otwarcie, o ile ma do tego odpowiednie uprawnienia. Aby procesy mogły odczytywać/zapisywać coś w pamięci dzielonej muszą odwzorować jej obszar w swoje przestrzenie adresowe. Dostęp do pamięci dzielonej najczęściej kontroluje się przy pomocy semaforów. Kiedy proces przestaje korzystać z obiektu pamięci dzielonej, to powinien usunąć do niego odwzorowanie ze swojej przestrzeni adresowej, a następnie go zamknąć. Po zakończeniu używania obiektu przez wszystkie korzystające z niego procesy powinien on zostać usunięty (zwykle za to odpowiedzialny jest proces, który dany obiekt utworzył). W systemie Linux obiekty pamięci dzielonej tworzone są w wirtualnym systemie plików i zwykle montowane pod `/dev/shm`.

→ Uwaga: Aby można było używać pamięci dzielonej w programach w języku C, należy je linkować z opcją: -lrt (w celu dołączenia biblioteki librt).

6.2 Tworzenie/otwieranie i usuwanie obiektu pamięci dzielonej

Do tworzenia i otwierania nowego lub otwierania istniejącego obiektu pamięci dzielonej standardu POSIX służy funkcja shm_open przedstawiona w poniższej tabeli. Operacja

Pliki włączane	`<sys/mman.h>, <sys/stat.h>, <fcntl.h>`		
Prototyp	`int shm_open(const char *name, int flags, mode_t mode);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	Deskryptor obiektu	-1	Tak

funkcji shm_open jest analogiczna do operacji funkcji open dla plików, tzn. pomyślnie wykonana tworzy/otwiera obiekt pamięci dzielonej i zwraca jego *deskryptor*, przy użyciu którego można wykonywać dalsze operacje na tym obiekcie.

• Parametry:

`name` nazwa obiektu pamięci dzielonej zaczynająca sie od znaku ukośnika (maksymalnie `NAME_MAX`, tj. 255, znaków),

`flags` opcje,

`mode` prawa dostępu do obiektu (podobie jak dla pliku)

• Opcje `flags`:

`O_RDONLY`	otwórz obiekt do czytania,
`O_RDWR`	otwórz obiekt do czytania i pisania,
`O_CREAT`	jeśli obiekt nie istnieje, to stwórz go,
`O_EXCL`	przy równocześnie ustawionej fladze `O_CREAT` przekaż błąd, jeśli obiekt już istnieje,
`O_TRUNC`	jeśli obiekt istnieje, zmniejsz jego długość do zera bajtów (obetnij go).

Jedną z pierwszych dwóch powyższych opcji można łączyć z dowolną z trzech pozostałych przy pomocy sumy bitowej, np. `O_RDWR | O_CREAT | O_EXCL`.

Nowo utworzony obiekt pamięci dzielonej ma długość zero. Aby ustawić niezerowy rozmiar, należy dla obiektu *otwartego do czytania i pisania* użyć funkcji ftruncate, przedstawionej w poniższej tabeli. Funkcja ta ustawia długość obiektu pamięci dzielonej o

Pliki włączane	`<unistd.h>, <sys/types.h>`		
Prototyp	`int ftruncate(int fd, off_t length);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

deskryptorze fd na wartość przekazaną przez parametr length.

Kiedy obiekt pamięci dzielonej otwarty funkcją shm_open nie jest już potrzebny w procesie, to można go zamknąć przy pomocy funkcji close, przedstawionej w poniższej tabeli. Działa ona analogicznie jak dla plików.

Pliki włączane	`<unistd.h>`		
Prototyp	`int close(int fd);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

Obiekt pamięci dzielonej można usunąć przy użyciu funkcji shm_unlink, przekazując jego nazwę przez parametr name. Obiekty pamięci dzielonej standardu POSIX są obiektami trwałymi jądra (ang. *kernel persistence*), więc jeśli taki obiekt nie zostanie usunięty funkcją shm_unlink, to będzie istniał aż do zamknięcia systemu (ang. *shutdown*).

Pliki włączane	`<sys/mman.h>`		
Prototyp	`int shm_unlink(const char *name);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

6.3 Odwzorowywanie pamięci dzielonej w przestrzeń adresową procesu

Aby proces mógł używać obiektu pamięci dzielonej, który otworzył funkcją shm_open, musi odwzorować (ang. *map*) go w swoją wirtualną przestrzeń adresową. Do tego służy funkcja mmap, przestawiona w poniższej tabeli. Funkcja ta tworzy odwzorowanie (ang.

Pliki włączane	`<sys/mman.h>`		
Prototyp	`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	adres odwzorowanego obszaru	`MAP_FAILED` (tzn. `(void *) -1`)	Tak

mapping) obiektu pamięci dzielonej (lub pliku) o deskryptorze fd w wirtualną przestrzeń adresową wywołującego procesu. Jeżeli addr ma wartość `NULL`, to jądro systemu wybiera adres początkowy takiego odwzorowania (wyrównany do granicy strony pamięci), w przeciwnym wypadku przekazywany adres stanowi sugestię dla jądra, gdzie wykonać odwzorowanie, np. Linux wybiera najbliższą granicę strony pamięci. Parametr length oznacza długość odwzorowanego obszaru w bajtach (> 0) liczoną od miejsca (marginesu) obiektu pamięci dzielonej (lub pliku) przekazywanego przez parametr offset (zwykle 0), prot opisuje żądaną ochronę pamięci, a flags określa opcje odwzorowania.

- Możliwe wartości parametru ochrony `prot`:

`PROT_EXEC`	prawo wykonywania,
`PROT_READ`	prawo czytania,
`PROT_WRITE`	prawo pisania,
`PROT_NONE`	brak dostępu.

Trzy pierwsze z powyższych praw można łączyć sumą bitową, np. dla prawa czytania i pisania: `PROT_READ | PROT_WRITE`.

Możliwych opcji odwzorowania przekazywanych przez parametr flags jest wiele, jednak z punktu widzenia pamięci dzielonej najważniejszą (i wystarczającą dla naszych celów) jest `MAP_SHARED`, która sprawia, że uaktualnienia obszaru takiej pamięci są widoczne dla wszystkich procesów, które odwzorowały go w swoje przestrzenie adresowe.

Zakończone sukcesem wywołanie funkcji mmap zwraca adres początku odwzorowanego obszaru pamięci dzielonej. Ponieważ zwracany adres jest postaci uniwersalnego wskaźnika `(void *)`, więc przypisując go do wskaźnika określonego typu powinno się wykonać odpowiednie rzutowanie typu[5], np. `int *p = (int *) mmap(...);`. W tak odwzorowanym obszarze pamięci dzielonej można wykonywać operacje w taki sam sposób jak w zwykłej pamięci procesu, tzn. pobierać z niego dane za pomocą zmiennych odwołujących się do tego obszaru i wstawiać do niego dane przy użyciu zwykłego operatora przypisania (=).

[5] W języku C takie rzutowanie jest opcjonalne, jednak dla lepszej przejrzystości kodu dobrze je zastosować, natomiast w języku C++ jest ono obowiązkowe.

Proces może usunąć odwzorowanie obszaru pamięci dzielonej ze swojej wirtualnej przestrzeni adresowej przez wywołanie funkcji munmap, przedstawionej w poniższej tabeli. Ka-

Pliki włączane	`<sys/mman.h>`		
Prototyp	`int munmap(void *addr, size_t length);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	−1	Tak

suje ona odwzorowanie obszaru o adresie `addr` i długości `length`, które wcześniej zostało utworzone odpowiednim wywołaniem funkcji mmap. Odwzorowanie takie jest usuwane automatycznie kiedy proces kończy swoje działanie, natomiast zamknięcie deskryptora obiektu pamięci dzielonej nie powoduje usunięcia odwzorowania jego obszaru. Generalnie kiedy proces kończy używać jakiegoś obszaru pamięci dzielonej, to najpierw powinien usunąć jego odwzorowanie przy pomocy funkcji munmap, a następnie zamknąć deskryptor obiektu funkcją close.

Oprócz opisanych powyżej podstawowych funkcji dotyczących pamięci dzielonej standardu POSIX istnieją trzy pomocnicze funkcje: fstat – zwracająca strukturę z informacją na temat obiektu pamięci dzielonej, fchown – do zmiany właściciela takiego obiektu i fchmod – do zmiany praw dostępu do niego. Więcej szczegółow na temat tych oraz pozostałych funkcji można znaleźć w podręczniku systemowym `man shm_overview`.

ĆWICZENIE 7: Producent–Konsument: pamięć dzielona i semafory

Przy pomocy **pamięci dzielonej** oraz **semaforów** standardu POSIX zaimplementować problem **,,producenta–konsumenta"** z **ćwiczenia 4** przedstawiony poniższym pseudokodem. Zamiast potoku użyć N-elementowego bufora cyklicznego (tzn. po dojściu do końca bufora wracamy na jego początek) umieszczonego w pamięci dzielonej, gdzie elementem bufora jest pewna ustalona porcja bajtów (> 1). Dostęp do wspólnego bufora synchronizować przy pomocy semaforów nazwanych standardu POSIX.

Programy producenta i konsumenta uruchamiać przez execlp w procesach potomnych utworzonych przez fork w procesie macierzystym – proces ten powinien wcześniej utworzyć i zainicjować semafory oraz pamięć dzieloną, a po utworzeniu procesów potomnych zaczekać na ich zakończenie, po czym zrobić odpowiednie porządki. Podobnie jak w ćwiczeniu 6, usuwanie obiektu pamięci dzielonej oraz semaforów umieścić w funkcji rejestrowanej przez atexit oraz funkcji obsługi sygnału `SIGINT`.

Bufor jednostek towaru można zdefiniować (najlepiej we wspólnym pliku naglówkowym) jako tablicę dwuwymiarową i umieścić go wewnątrz struktury wraz z iteratorami dla producenta i konsumenta, np.

```
#define NELE 20  // Rozmiar elementu bufora (jednostki towaru) w bajtach
#define NBUF  5  // Liczba elementow bufora

// Segment pamieci dzielonej
typedef struct {
  char bufor[NBUF][NELE];  // Wspolny bufor danych
  int  wstaw, wyjmij;      // Pozycje wstawiania i wyjmowania z bufora
} SegmentPD;
```

Taki obiekt należy odwzorować w przestrzenie adresowe procesów producenta i konsumenta, np.

```
SegmentPD *wpd = (SegmentPD *) mmap(NULL,sizeof(SegmentPD),
                                    PROT_READ,MAP_SHARED,des,0);
```

Następnie, przy użyciu tak zdefiniowanego wskaźnika można wykonywać operacje na tym segmencie, np.

```
wpd->wyjmij = (wpd->wyjmij + 1) % NBUF;
```

Podobnie jak dla semaforów, stworzyć własną bibliotekę funkcji do obsługi pamięci dzielonej.

→ *Z własnych mudułów funkcji do obsługi semaforów i pamięci dzielonej stworzyć (jedną) bibliotekę statyczną oraz (jedną) bibliotekę dynamiczną/dzieloną (ang. dynamic/shared library); umieścić je w podkatalogu* `./lib`. *Program z bibiloteką dynamiczną uruchamiać na dwa sposoby; patrz przykład w* `StartSO`.

→ *Podać w pseudokodzie uogólnienie synchronizacji dla tego problemu na przypadek wielu producentów i wielu konsumentów (dla uproszczenia przyjmijmy, że kolejność przesyłanych porcji danych nie jest istotna).*

→ *Przykład pseudokodu dla* **ćwiczenia 7***:*

```
// Pseudokod dla problemu Producenta i Konsumenta z buforem cyklicznym.
// Wspolny bufor do przesylania danych znajduje sie w pamieci dzielonej.
// Dostep do bufora jest synchronizowany semaforami.

#define N ?                   // Rozmiar bufora

typdef struct { ... } Towar; // Definicja typu dla jednostek towaru
Towar bufor[N];              // Bufor mogacy pomiescic N jednostek towaru
int wstaw = 0, wyjmij = 0;   // Pozycje wstawiania oraz wyjmowania towaru
                             // z bufora (mozna umiescic w pamieci dzielonej)

semaphore PROD = N;          // Semafor do wstrzymywania Producenta
semaphore KONS = 0;          // Semafor do wstrzymywania Konsumenta


// Proces Producent
// ----------------
  Towar towarProd;
  while (1) {
    // Produkcja towaru
    P(PROD);                     // Opusc semafor Producenta
    bufor[wstaw] = towarProd;    // Umiesc towar w buforze
    wstaw = (wstaw + 1) % N;     // Przesun pozycje wstawiania o 1 dalej
    V(KONS);                     // Podniesc semafor Konsumenta
  }

// Proces Konsument
// ----------------
  Towar towarKons;
  while (1) {
    P(KONS);                     // Opusc semafor Konsumenta
    towarKons = bufor[wyjmij];   // Umiesc towar w buforze
    wyjmij = (wyjmij + 1) % N;   // Przesun pozycje wstawiania o 1 dalej
    V(PROD);                     // Podniesc semafor Producenta
    // Konsumpcja towaru
  }
```