Komunikacja międzyprocesowa (IPC)

Procesy tworzone w ramach systemu operacyjnego mogą się komunikować. Jeżeli duża aplikacja jest budowana z wielu współpracujących procesów, to ta komunikacja może być intensywna, i jej poprawna oraz sprawna realizacja może być krytyczna dla poprawnej pracy aplikacji. Usługi komunikacji między procesowej (*Inter-Process Communication* — IPC), należą do ważnych funkcji systemu operacyjnego.

Ogólnie, rozróżnia się następujące tryby komunikacji międzyprocesowej:

- przesyłanie komunikatów (*message passing*),
- pamięć wspólna/współdzielona (*shared memory*).

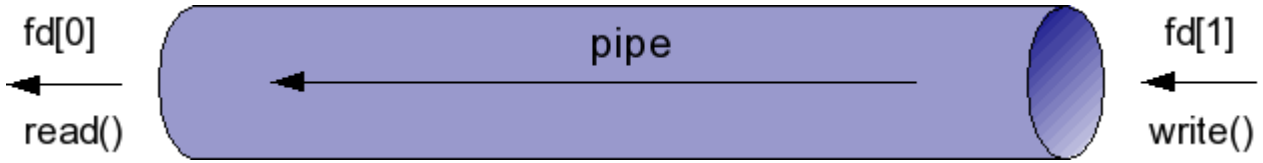
Następujące zagadnienia muszą być rozwiązane dla zapewnienia komunikacji międzyprocesowej:

- mechanizm komunikacji,
- zapobieganie kolizjom dostępu do wspólnego medium,
- organizacja oczekiwania i kolejkowania, gdy żądanie jednego procesu nie może być od razu zaspokojone.

Potoki

Potok (*pipe*) jest jednym z najbardziej podstawowych mechanizmów komunikacji międzyprocesowej. Potok jest urządzeniem komunikacji szeregowej, jednokierunkowej, o następujących własnościach:

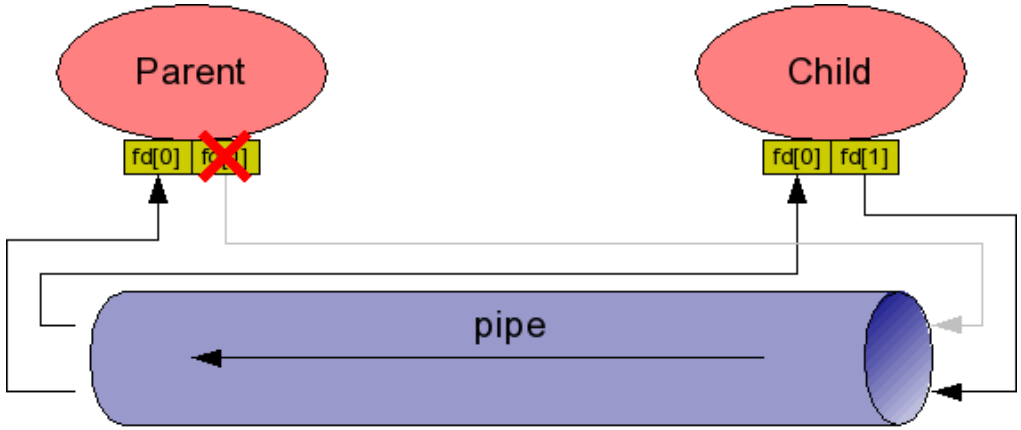
- na potoku można wykonywać tylko operacje odczytu i zapisu, funkcjami `read()` i `write()`, jak dla zwykłych plików,
- potoki są dostępne i widoczne w postaci jednego lub dwóch deskryptorów plików, oddzielnie dla końca zapisu i odczytu,
- potok ma określoną pojemność, i w granicach tej pojemności można zapisywać do niego dane bez odczytywania,
- próba odczytu danych z pustego potoku, jak również zapisu ponad pojemność potoku, powoduje zawiśnięcie operacji I/O (normalnie), i jej automatyczną kontynuację gdy jest to możliwe; w ten sposób potok **synchronizuje** operacje I/O na nim wykonywane.



Dobrą analogią potoku jest rurka, gdzie strumień danych jest odbierany jednym końcem, a wprowadzany drugim. Gdy rurka się zapełni i dane nie są odbierane, nie można już więcej ich wprowadzić.

Potoki: funkcja `pipe`

Funkcja `pipe()` tworzy tzw. „anonimowy" potok, dostępny w postaci dwóch otwartych i gotowych do pracy deskryptorów:



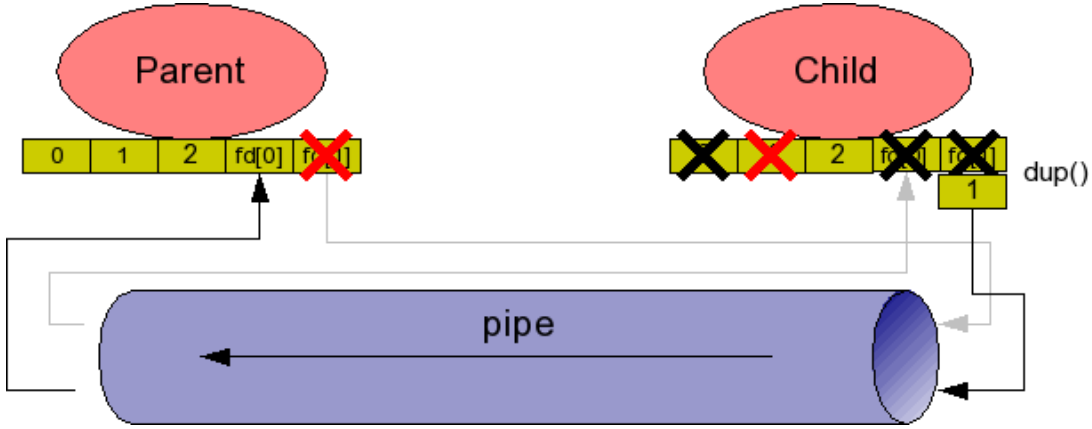

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define KOM "Komunikat dla rodzica.\n"
int main() {
  int potok_fd[2], licz, status;
  char bufor[BUFSIZ];

  pipe(potok_fd);
  if (fork() == 0) {
    write(potok_fd[1], KOM, strlen(KOM));
    exit(0);
  }
  close(potok_fd[1]);  /* wazne */
  while ((licz=read(potok_fd[0], bufor, BUFSIZ)) > 0)
    write(1, bufor, licz);
  wait(&status);
  return(status);
}
```

Funkcja `read()` natychmiast zwraca 0 po próbie odczytu z pustego potoku gdy jest on zamknięty do zapisu, lecz gdy jakiś proces w systemie ma ten potok otwarty do zapisu, funkcja `read()` „zawisa" na próbie odczytu (proces przechodzi do stanu oczekiwania).

Potoki: zasady użycia

- Potok (anonimowy) zostaje zawsze utworzony otwarty, i gotowy do zapisu i odczytu.
- Próba odczytania z potoku większej liczby bajtów, niż się w nim aktualnie znajduje, powoduje przeczytanie dostępnej liczby bajtów i zwrócenie w funkcji `read()` liczby bajtów rzeczywiście przeczytanych.
- Próba czytania z pustego potoku, którego koniec piszący jest nadal otwarty przez jakiś proces, powoduje „zawiśnięcie" funkcji `read()`, i powrót gdy jakieś dane pojawią się w potoku.
- Czytanie z pustego potoku, którego koniec piszący został zamknięty, daje natychmiastowy powrót funkcji `read()` z wartością 0.
- Zapis do potoku odbywa się poprawnie i bez czekania pod warunkiem, że nie przekracza pojemności potoku; w przeciwnym wypadku `write()` „zawisa" aż do ukończenia operacji.
- Próba zapisu na potoku, którego koniec czytający został zamknięty, kończy się porażką i proces piszący otrzymuje sygnał `SIGPIPE`.
- Standard POSIX określa potoki jako jednokierunkowe. Jednak implementacja potoków większości współczesnych systemów uniksowych zapewnia komunikację dwukierunkową.

Potoki: przekierowanie standardowego wyjścia




```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int potok_fd[2], licz;  char bufor[BUFSIZ];

  pipe(potok_fd);
  if (fork() == 0) {          /* podproces tylko piszacy */
    close(1);                 /* zamykamy stdout prawdziwy */
    dup(potok_fd[1]);         /* odzyskujemy fd 1 w potoku */
    close(potok_fd[1]);       /* dla porzadku */
    close(potok_fd[0]);       /* dla porzadku */
    close(0);                 /* dla porzadku */
    execlp("ps","ps","-fu",getenv("LOGNAME"),NULL);
  }
  close(potok_fd[1]);         /* wazne */
  while ((licz=read(potok_fd[0], bufor, BUFSIZ)) > 0)
    write(1, bufor, licz);
}
```

Potoki nazwane (FIFO)

- istnieją trwale w systemie plików (`mknod potok p`)
- wymagają otwarcia `O_RDONLY` lub `O_WRONLY`
- próba otwarcia potoku jeszcze nieotwartego w trybie komplementarnym zawisa i czeka aż do innej próby otwarcia w drugim trybie

SERWER:

```
#include <fcntl.h>
#define FIFO "/tmp/potok_1"
#define MESS \
  "To jest komunikat serwera\n"

void main() {
  int potok_fd;

  potok_fd = open(FIFO, O_WRONLY);
  write(potok_fd,
        MESS, sizeof MESS);
}
```

KLIENT:

```
#include <fcntl.h>
#define FIFO "/tmp/potok_1"


void main() {
  int potok_fd, licz;
  char bufor[BUFSIZ];

  potok_fd = open(FIFO, O_RDONLY);
  while ((licz=read(potok_fd, bufor,
                    BUFSIZ)) > 0)
    write(1, bufor, licz);
}
```

Operacje na FIFO

- Pierwszy proces otwierający FIFO zawisa na operacji otwarcia, która kończy się gdy FIFO zostanie otwarte przez inny proces w komplementarnym trybie (`O_RDONLY/O_WRONLY`).
- Można wymusić nieblokowanie funkcji `open()` opcją `O_NONBLOCK` lub `O_NDELAY`, lecz takie otwarcie w przypadku `O_WRONLY` zwraca błąd.
- Próba odczytu z pustego FIFO w ogólnym przypadku zawisa gdy FIFO jest otwarte przez inny proces do zapisu, lub zwraca 0 gdy FIFO nie jest otwarte do zapisu przez żaden inny proces.

  To domyślne zachowanie można zmodyfikować ustawiając flagi `O_NDELAY` i/lub `O_NONBLOCK` przy otwieraniu FIFO. RTFM.
- W przypadku zapisu zachowanie funkcji `write()` nie zależy od stanu otwarcia FIFO przez inne procesy, lecz od zapełnienia buforów. Ogólnie zapisy krótkie mogą się zakończyć lub zawisnąć gdy FIFO jest pełne, przy czym możemy wymusić niezawisanie podając opcje `O_NDELAY` lub `O_NONBLOCK` przy otwieraniu FIFO.
- Oddzielną kwestią jest, że dłuższe zapisy do FIFO ($\geq$`PIPE_BUF` bajtów) mogą mieszać się z zapisami z innych procesów.

Potoki — podsumowanie

Potoki są prostym mechanizmem komunikacji międzyprocesowej opisane standardem POSIX i istniejącym w wielu systemach operacyjnych. Pomimo iż definicja określa potok jako mechanizm komunikacji jednokierunkowej, to wiele implementacji zapewnia komunikację dwukierunkową. Oznacza to istnienie w potoku dwóch przeciwbieżnych strumieni danych.

Podstawowe zalety potoków to:

- brak limitów przesyłania danych,
- synchronizacja operacji zapisu i odczytu przez stan potoku.
  Praktycznie synchronizuje to komunikację pomiędzy procesem, który dane do potoku zapisuje, a innym procesem, który chciałby je odczytać, niezależnie który z nich wywoła swoją operację pierwszy.

Jednak nie ma żadnego mechanizmu umożliwiającego synchronizację operacji I/O pomiędzy procesami, które chciałyby jednocześnie wykonać operacje odczytu, lub jednocześnie operacje zapisu. Z tego powodu należy uważać potoki za mechanizm komunikacji typu 1-1, czyli jeden do jednego. Komunikacja typu wielu-wielu przez potok jest utrudniona i wymaga zastosowania innych mechanizmów synchronizacji.

Kolejki komunikatów

Kolejki komunikatów są mechanizmem komunikacji o własnościach podobnych do potoków. Istnieje wiele standardów implementacji kolejek komunikatów. Tu dokonamy przeglądu podstawowych własności kolejek komunikatów standardu POSIX:

- kolejki nie są plikami

  Na kolejkach komunikatów nie wykonuje się operacji funkcjami `open/read/write` jak na zwykłych plikach. Kolejki posiadają swoje specjalne operacje I/O.

- dwukierunkowa komunikacja

  Kolejka może być otwarta w jednym z trybów: `O_RDONLY`, `O_WRONLY`, `O_RDWR`.

- stały rozmiar komunikatu

  Odmiennie niż potoki (anonimowe i FIFO), które są strumieniami bajtów, kolejki przekazują komunikaty jako całe jednostki.

- priorytety komunikatów

  Komunikaty przesyłane przez kolejki POSIX posiadają priorytety — zawsze odbierany jest najstarszy komunikat o najwyższym priorytecie. Możliwe jest zatem odczytywanie komunikatów w kolejności innej niż kolejność nadania.

- blokujące lub nieblokujące zapisy i odczyty

  Kolejki POSIX posiadają zdolność blokowania procesu w oczekiwaniu na odczyt komunikatu gdy kolejka jest pusta, lub natychmiastowego powrotu z kodem braku komunikatu. Analogicznie, proces usiłujący zapisać do pełnej kolejki może blokować (czekać), lub wrócić natychmiast z błędem.

  Zauważmy, że mechanizm blokujących zapisów i odczytów pozwala synchronizować komunikację procesów, podobnie jak w przypadku potoków.

- asynchroniczne powiadamianie

  Kolejki komunikatów POSIX posiadają dodatkową funkcję pozwalającą zażądać asynchronicznego powiadomienia o nadejściu komunikatu do kolejki. Dzięki temu proces może zajmować się czymś innym, a w momencie nadejścia komunikatu może zostać powiadomiony przez:
  - doręczenie sygnału
  - uruchomienie określonej funkcji jako nowego wątku

Ze względu na istnienie priorytetów i możliwość selektywnego odczytywania komunikatów, kolejki komunikatów nie narzucają tak ściśle komunikacji 1:1 jak potoki. Możliwych jest szereg wariantów komunikacji wielu-wielu.

Kolejki komunikatów POSIX

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);

int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio, const struct timespec *abs_timeout);
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                   unsigned int msg_prio);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                   unsigned int msg_prio, const struct timespec *abs_timeout);

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
                   struct mq_attr *oldattr);

int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Przedstaw ogólne własności i działanie potoków.

2. Jaki wynik może przynieść próba odczytu danych z pustego potoku?

3. Wyjaśnij w jaki sposób komunikacja przez potok może synchronizować pracę procesów.

4. Wyjaśnij w jaki sposób procesy mogą nawiązać komunikację przez potok anonimowy, a w jaki przez potok nazwany.

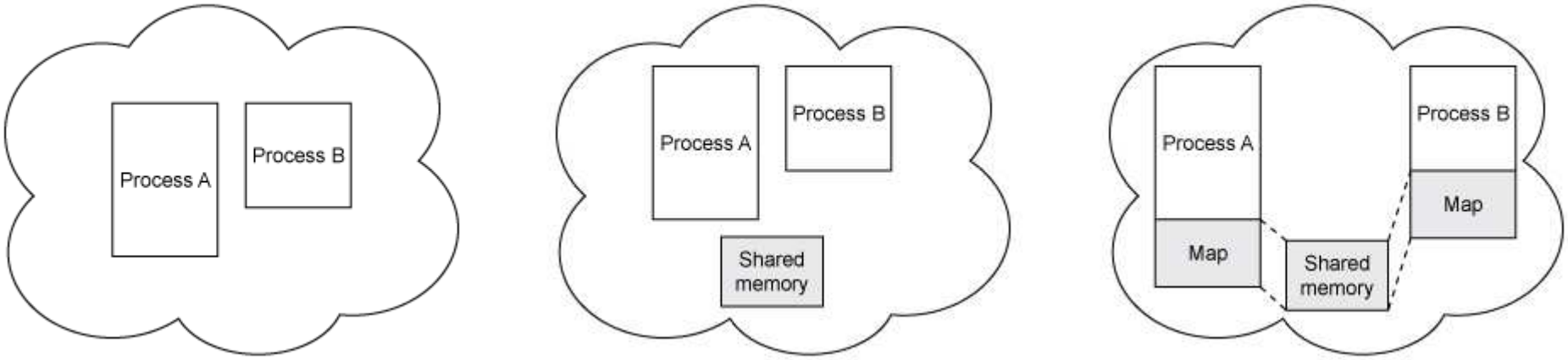
5. Wyjaśnij co to znaczy, że potoki są mechanizmem komunikacji 1-1.

6. Przeczytaj opis API kolejek komunikatów POSIX i wyjaśnij jaką rolę pełni priorytet komunikatu.

Komunikacja międzyprocesowa — obszary pamięci wspólnej

Komunikacja przez pamięć wspólną wymaga stworzenia obszaru pamięci wspólnej w systemie operacyjnym przez jeden z procesów, oraz **odwzorowania** tej pamięci do własnej przestrzeni adresowej wszystkich pragnących się komunikować procesów. Następnie komunikacja odbywa się przez zwykłe operacje na zmiennych, lub dowolną funkcją wykonującą odczyty/zapisy pamięci, np. `strcpy`, `memcpy`, itp.

Komunikacja przez pamięć wspólną jest najszybszym rodzajem komunikacji międzyprocesowej, ponieważ dane nie są nigdzie przesyłane. W momencie ich utworzenia w lokalizacji źródłowej są od razu również dostępne w lokalizacji docelowej. Jednak wymaga synchronizacji za pomocą oddzielnych mechanizmów, takich jak muteksy albo blokady zapisu i odczytu.



Obrazki zapożyczone bez zezwolenia z:

`http://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/`

Mechanizmy komunikacji międzyprocesowej standardu POSIX Realtime opierają identyfikację na deskryptorach plików, do których dostęp można uzyskać przez identyfikatory zbudowane identycznie jak nazwy plików. Nazwy plików muszą zaczynać się od slash-a „/" (co podkreśla fakt, że mają charakter globalny), jednak standard nie określa, czy te pliki muszą istnieć/być tworzone w systemie, a jeśli tak to w jakiej lokalizacji. Takie rozwiązanie pozwala systemom, które mogą nie posiadać systemu plików (jak np. systemy wbudowane) tworzyć urządzenia komunikacyjne w jakiejś wirtualnej przestrzeni nazw, natomiast większym systemom na osadzenie ich w systemie plików według dowolnej konwencji.

Urządzenia oparte o konkretną nazwę pliku zachowują swój stan (np. kolejka komunikatów swoją zawartość, a semafor wartość) po ich zamknięciu przez wszystkie procesy z nich korzystające, i ponownym otwarciu. Standard nie określa jednak, czy ten stan ma być również zachowany po restarcie systemu.

Kroki niezbędne przy komunikacji przez pamięć współdzieloną:

1. Otwarcie/utworzenie pliku obszaru pamięci wspólnej `shm_open()`
2. Ustalenie rozmiaru obszaru pamięci wspólnej `ftruncate()`
3. Odwzorowanie obszaru pamięci wspólnej do obszaru w pamięci procesu `mmap()`
4. Praca: operacje wejścia/wyjścia `read()/write()`
5. Skasowanie odwzorowania `munmap()`
6. Ewentualnie skasowanie pliku obszaru pamięci wspólnej `shm_unlink()`

Pamięć współdzielona: serwer

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/mman.h>

#define SHM_SEGMENT "/shm_segment"
#define MODES 0666

typedef struct {
  int client_wrote;
  char text[BUFSIZ];
} shared_struct;

int main()
{
  int shmd, shared_size;
  shared_struct *segment;

  // na wszelki wypadek
  printf("Usuwam segment wspolny.\n");
  if(shm_unlink(SHM_SEGMENT) < 0)
    perror("nie moge usunac segmentu");
  else printf("Segment usuniety.\n");

  shmd = shm_open(SHM_SEGMENT,O_RDWR|O_CREAT,MODES);
  if (shmd == -1) {
    perror("shm_open padlo");
    exit(errno);
  }

  shared_size = sizeof(shared_struct);
  ftruncate(shmd, shared_size);
  segment =
    (shared_struct *)
    mmap(NULL, shared_size, PROT_READ|PROT_WRITE,
         MAP_SHARED, shmd, 0);

  srand((unsigned int)getpid());
  segment->client_wrote = 0;
  do {
    printf("Czekam na dane ...\n");
    sleep( rand() % 4 );          /* troche czekamy */
    if (segment->client_wrote) {
      printf("Otrzymane: \"%s\"\n", segment->text);
      sleep( rand() % 4 );        /* znow poczekajmy*/
      segment->client_wrote = 0;
    }
  } while (strncmp(segment->text, "koniec", 6) !=0);

  munmap((char *)segment, shared_size);
  return 0;
}
```

Pamięć współdzielona: klient

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/mman.h>

#define SHM_SEGMENT "/shm_segment"
#define MODES 0666

typedef struct {
    int client_wrote;
    char text[BUFSIZ];
} shared_struct;

int main()
{
  int shmd;
  shared_struct *segment;
  char buf[BUFSIZ];

  shmd = shm_open(SHM_SEGMENT, O_RDWR, MODES);
  if (shmd == -1) {
    perror("shm_open padlo");
    exit(errno);
  }

  segment =
    (shared_struct *)
    mmap(NULL, sizeof(shared_struct),
         PROT_READ|PROT_WRITE, MAP_SHARED, shmd, 0);
  do {
    while(segment->client_wrote == 1) {
      sleep(1);
      printf("Czekam na odczytanie...\n");
    }
    printf("Podaj tekst do przeslania: ");
    fgets(buf, BUFSIZ, stdin);
    strcpy(segment->text, buf);
    segment->client_wrote = 1;
  } while (strncmp(buf, "koniec", 6) != 0);

  munmap((char *)segment, sizeof(shared_struct));
  return 0;
}
```

Wstęp do synchronizacji: semafory

W teorii **semafor** jest mechanizmem synchronizacyjnym, domyślnie kontrolującym dostęp lub przydział pewnego zasobu. Wartość semafora oznacza liczbę dostępnych jednostek zasobu. Określone są następujące operacje na semaforze:

P(sem) — oznacza zajęcie zasobu sygnalizowane zmniejszeniem wartości semafora o 1, a jeśli jego aktualna wartość jest 0 to oczekiwanie na jej zwiększenie,

V(sem) — oznacza zwolnienie zasobu sygnalizowane zwiększeniem wartości semafora o 1, a jeśli istnieje(ą) proces(y) oczekujący(e) na semaforze to, zamiast zwiększać wartość semafora, wznawiany jest jeden z tych procesów.

Analogią, ilustrującą zastosowanie semaforów może być wystawa (np. malarstwa), na którą ze względów bezpieczeństwa można wpuścić jednorazowo maksymalnie N osób. Semafor wstępnie ustawiamy na wartość N. Każda wchodząca osoba zgłasza żądanie jednostkowego zajęcia semafora, co dekrementuje jego licznik (wyrażający liczbę wolnych miejsc na wystawie). Po osiągnięciu maksimum (wyzerowaniu semafora), następne osoby zgłaszające żądania zajęcia semafora muszą czekać. Każda wychodząca osoba zwalnia (inkrementuje) semafor, co może spowodować wpuszczenie jednej osoby do środka, o ile czeka w żądaniu zajęcia semafora.

Semafory — implementacja

Nietrudno zauważyć, że semafor może być po prostu zmienną całkowitoliczbową o wartościach nieujemnych. Istotna jest jednak niepodzielna realizacja każdej z operacji P, V. To znaczy, każda z tych operacji może albo zostać wykonana w całości, albo w ogóle nie zostać wykonana. Z tego powodu niemożliwa jest prywatna implementacja operacji semaforowych przy użyciu zmiennej globalnej przez proces pracujący w warunkach przełączania procesów. Operacje semaforowe realizowane są przez system operacyjny, który zapewnia ich niepodzielność.

Semafory POSIX

Poniższy interfejs funkcyjny dotyczy semaforów standardu POSIX Realtime. Semafory tworzone w tym standardzie istnieją jako pliki w systemie plików, aczkolwiek standard nie określa jaki ma to być system plików. Na przykład, może to być wirtualny system plików, niezlokalizowany na dysku komputera. Istnieją również semafory anonimowe, dostępne tylko w pamięci (`sem_init/sem_destroy`).

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);

int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

Zastosowanie semaforów do synchronizacji

Przydatnym przypadkiem szczególnym jest semafor binarny. Wartość takiego semafora może wynosić 1 lub 0. Semafory binarne zwane są również **muteksami** (ang. *mutex* = *mut*ual *ex*clusion).

Semafor binarny może być wykorzystany do kontroli dostępu do współdzielonego zasobu, takiego jak pamięć wspólna. W celu wykonania zapisu do pamięci wspólnej proces byłby zobowiązany wpierw do zajęcia semafora, gwarantując w ten sposób brak jednoczesnych zapisów przez różne procesy.

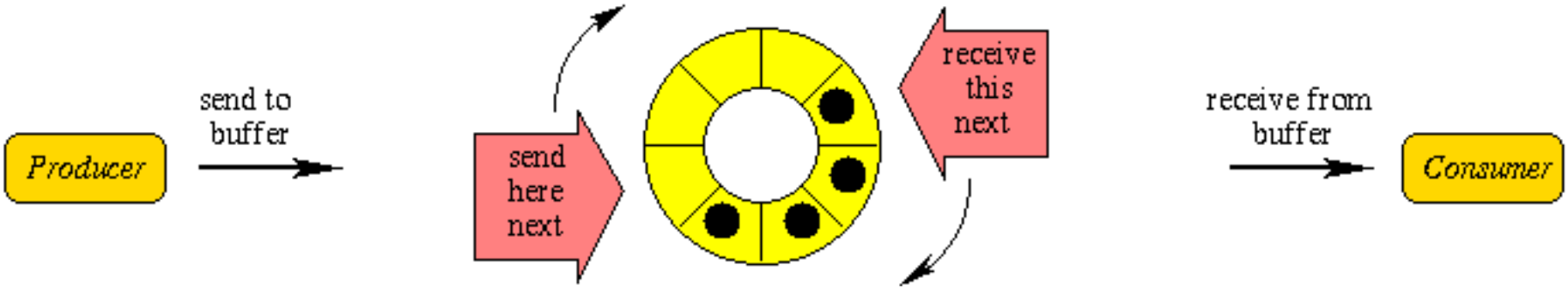
Zauważmy jednak, że w takim przypadku proces odczytujący obszar pamięci wspólnej musiałby również zająć semafor, aby zapobiec ewentualnej operacji zapisu do obszaru, który zamierza odczytać.[1]

[1] Określenie semafora jako licznika zasobów w sztukach może wydawać się mylące. W kolejnictwie semafor jest stosowany do sygnalizacji zajętego toru. Informatycznym odpowiednikiem tego mechanizmu jest właściwie mutex. Semafory wprowadził w 1965 holenderski informatyk Edsger Dijkstra. Pojęcie muteksu, jako semafora binarnego, zostało zdefiniowane dużo później.

Zagadnienie ograniczonego bufora

Modelowym przykładem zagadnienia, który można rozwiązać za pomocą komunikacji i synchronizacji, jest tzw. **problem ograniczonego bufora** (*bounded buffer problem*). Jest to wariant ogólnego zagadnienia producentów i konsumentów, w którym pewna grupa producentów produkuje jakieś dane, które muszą być następnie pobrane i dalej przetworzone („skonsumowane") przez grupę konsumentów.
W problemie ograniczonego bufora przekazywanie danych między producentami a konsumentami odbywa się za pośrednictwem bufora, którego pojemność jest dużo mniejsza niż liczba produkowanych i przekazywanych elementów. Zatem konieczna jest skuteczna koordynacja pracy wszystkich aktorów przy przekazywaniu wyprodukowanych elementów.

Zagadnienie ograniczonego bufora (cd.)




```
void producer()
{
  for(int i=0; i<REPEAT; i++) {
    //czekaj gdy brak pustych
    sem_wait(&(share->empty));
    sem_wait(&(share->mutex));
    printf("Prod: count=%d in=%d out=%d\n",
           share->count,share->in,share->out);
    sprintf(share->buf[share->in],
            "Komunikat %03d", i);
    share->count++;
    share->in = (share->in + 1) % B_SIZE;
    sem_post(&(share->mutex));
    //dodaj jeden zapelniony
    sem_post(&(share->full));
    sleep(1);
  }
  exit(0);
}
```

```
void consumer(void)
{
  for(int i=0; i<(PROD_NO*REPEAT); ++i) {
    //czekaj gdy brak gotowych
    sem_wait(&(share->full));
    sem_wait(&(share->mutex));
    printf("Cons: count=%d rcvd=%s\n",
           share->count,
           share->buf[share->out]);
    share->count--;
    share->out = (share->out + 1) % B_SIZE;
    sem_post(&(share->mutex));
    //dodaj jeden oprozniony
    sem_post(&(share->empty));
    sleep(1);
  }
  exit(0);
}
```

Zagadnienie ograniczonego bufora (cd.)

```
#define SHM_SEGMENT "/shm_segment"
#define MODES   0600  // prawa dostepu
#define B_SIZE  5     // rozmiar bufora
#define L_SIZE  80    // dlugosc wiersza
#define REPEAT  20    // liczba produktow
#ifndef PROD_NO
#define PROD_NO 2     // liczba producen.
#endif

typedef struct {
  char buf[B_SIZE][L_SIZE];
  int next_in;   // pierwsze wolne na prod
  int next_out;  // ost.zajete gdy count>0
  int count;     // liczba zajetych w buf.
  sem_t mutex;
  sem_t empty;
  sem_t full;
} shared_struct;

shared_struct *segment;

int main()
{
  int i, shmd;

  // utworz i zainicjalizuj segment
  shm_unlink(SHM_SEGMENT);
  shmd=shm_open(SHM_SEGMENT,
                O_RDWR|O_CREAT, MODES);
  ftruncate(shmd, B_SIZE);
  segment = (shared_struct *)
            mmap(0, B_SIZE, PROT_READ|
                 PROT_WRITE, MAP_SHARED,shmd,0);

  // inicjalizacja wartosci semaforow
  segment->count = 0;
  segment->in = 0;
  segment->out = 0;
  sem_init(&(segment->mutex), 1, 1));
  sem_init(&(segment->empty), 1, B_SIZE));
  sem_init(&(segment->full), 1, 0));

  // uruchom produkcje/konsumpcje
  for(i=1; i<=PROD_NO; ++i)
    if (fork() == 0) producer(i);
  if (fork() == 0) consumer();

  // odlacz semafory proc.glownemu
  sem_close(&(segment->mutex));
  sem_close(&(segment->empty));
  sem_close(&(segment->full));


  // czekaj na potomkow i zakoncz
  for(i=1; i<=PROD_NO; ++i) wait(NULL);
  wait(NULL);
  return 0;
}
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Jakie właściwości ma komunikacja procesów przez pamięć wspólną?

2. Jakie kroki są wymagane w celu użycia segmentu pamięci wspólnej w standardzie POSIX Realtime?

3. Dlaczego komunikacja przez pamięć wspólną wymaga synchronizacji?

4. W jaki sposób muteks może zostać użyty do synchronizacji komunikacji procesów przez pamięć wspólną?

5. Co to znaczy, że realizacja operacji na semaforze musi być niepodzielna, i dlaczego musi?

6. Na czym polega zagadnienie ograniczonego bufora?

7. W jaki sposób wykorzystywane są semafory `empty` i `full` w przedstawionym rozwiązaniu problemu ograniczonego bufora?

Uzupełnienie: deskryptory plików

```
#include <stdio.h>        /* wersja 5: operacje IO */
#include <fcntl.h>        /* niskiego poziomu      */

main(int argc, char *argv[])
{
  char buf[1024];
  int fd1, fd2, fdin;

  fd1 = open(argv[1], O_RDONLY, 0);
  fd2 = open(argv[2], O_WRONLY|O_CREAT, 0644);
  while ((fdin = read(fd1, buf, sizeof buf)) > 0)
    write(fd2, buf, fdin);
  exit(0);
}
```

- Funkcje `open()`, `read()`, `write()`, zwane wbudowanym systemem wejścia/wyjścia, lub systemem I/O niskiego poziomu, są głównym mechanizmem operacji na plikach i urządzeniach w Uniksie.

- Odwołują się one do otwartych plików przez numery deskryptorów.

Deskryptory plików — zasady

- Deskryptory są małymi liczbami przyporządkowanymi kolejno otwieranym plikom. Program ma gwarancję uzyskania najniższego wolnego deskryptora. Deskryptory 0,1,2 reprezentują automatycznie otwierane pliki `stdin`, `stdout`, i `stderr`.
- Numery deskryptorów otwartych plików są lokalne dla danego procesu. Jeśli inny proces otworzy ten sam plik to otrzyma własny (choć być może ten sam) numer deskryptora, z którym związany jest kursor pliku wskazujący pozycję wykonywanych operacji zapisu/odczytu.
- Operacje odczytu pliku otwartego jednocześnie przez dwa procesy mogą poprawnie przebiegać niezależnie od siebie.
- Jeśli dwa procesy jednocześnie otworzą ten sam plik do zapisu (lub zapisu i odczytu), to ich operacje zapisu będą się na siebie nakładały, jedne dane nadpiszą drugie, i otrzymamy w pliku tzw. sieczkę.
- Jednak dwa procesy mogą współdzielić jeden deskryptor otwartego pliku, i wtedy ich operacje będą się ze sobą przeplatać. Jeśli nie będziemy tego przeplatania kontrolować to również możemy otrzymać sieczkę, aczkolwiek w tym przypadku dane nie zostaną nadpisane.
- W każdym przypadku do synchronizowania operacji wejścia/wyjścia na plikach możemy użyć blokad.

Funkcje niskiego poziomu a biblioteka `stdio`

Funkcje I/O niskiego poziomu są innym, bardziej surowym, sposobem wykonywania operacji wejścia/wyjścia na plikach niż biblioteka `stdio`.

- Operacje I/O niskiego poziomu nie mają dostępu do struktur danych biblioteki `stdio` i w odniesieniu do danego otwartego pliku nie można ich mieszać z operacjami na poziomie tej biblioteki.
- Jednak w czasie pracy z biblioteką `stdio` funkcje niskiego poziomu też pracują (na niższym poziomie), i można korzystać z niektórych z nich, jeśli się wie co się robi.
- Na przykład, numer deskryptora pliku można uzyskać z file pointera biblioteki `stdio` (`fileno(fp)`), ale nie na odwrót (dlaczego?).

Szczegóły działania funkcji `read, write`

- Przy czytaniu funkcją `read()` liczba znaków przeczytanych może nie być równa liczbie znaków żądanych ponieważ w pliku może nie być tylu znaków.
  - W szczególności przy czytaniu z terminala funkcja `read()` normalnie czyta tylko do końca linijki.
  - Wartość 0 oznacza brak danych do czytania z pliku (koniec pliku), co jednak nie oznacza, że przy kolejnej próbie czytania jakieś dane się nie pojawią (inny proces może zapisać coś do pliku), zatem nie jest to błąd.
  - Wartość ujemna oznacza błąd.
- Przy pisaniu funkcją `write()` wartość zwrócona jest normalnie równa liczbie znaków żądanych do zapisu; jeśli nie to wystąpił jakiś błąd.
- Drugi parametr funkcji `open()` może przyjmować trzy wartości zadane następującymi stałymi: `O_RDONLY` (tylko czytanie), `O_WRONLY` (tylko pisanie), lub `O_RDWR` (czytanie i pisanie). Poza tym można dodać do tej wartości modyfikatory (bitowo), które zmieniają sposób działania operacji na pliku.
- Ostatni parametr funkcji `open()` zadaje 9 bitów praw dostępu dla tworzonego pliku (np. ósemkowo 0755).

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Co to są deskryptory plików?

2. W jaki sposób przydzielane są numery deskryptorów kolejno otwieranych plików?

3. Jaki jest związek deskryptorów plików z wskaźnikami plików biblioteki `stdio`?

4. Jakie wartości zwracają funkcje `read()` i `write()`?

5. W jakich przypadkach funkcja `read()` może przeczytać inną liczbę znaków niż deklarowana wielkość bufora?

6. W jakich przypadkach funkcja `write()` może zapisać inną liczbę znaków niż zadana argumentem?