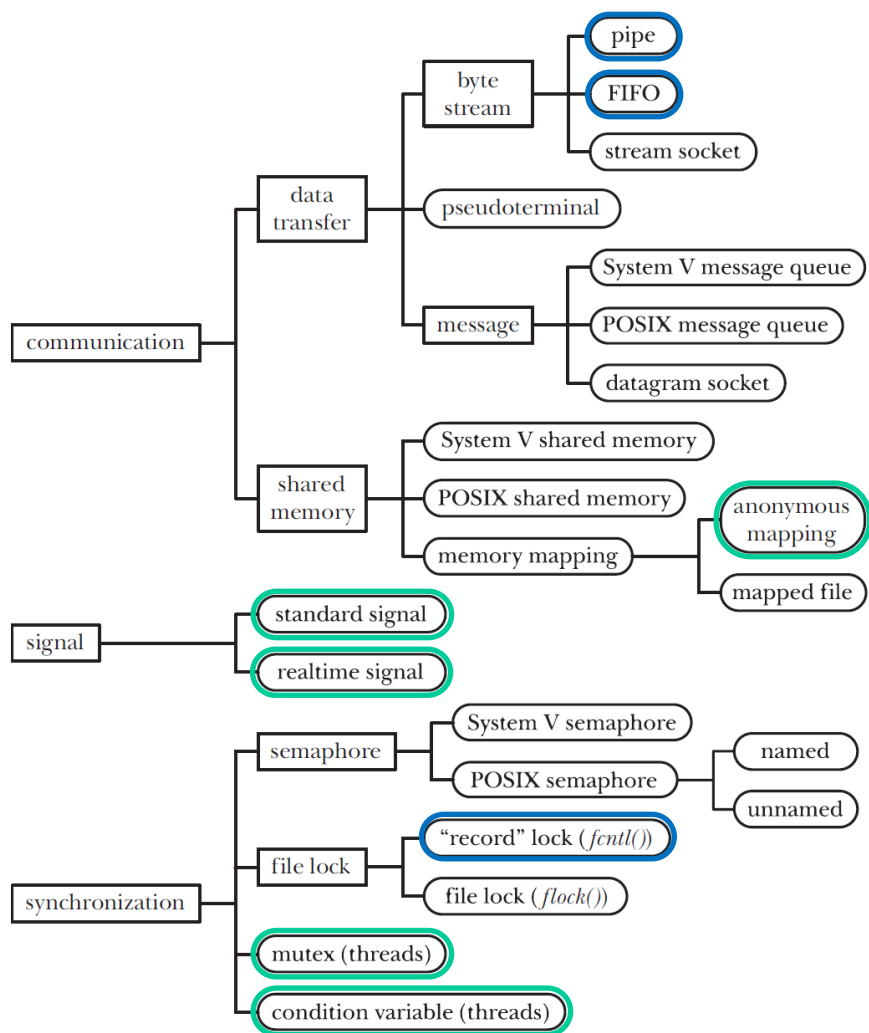


KOMUNIKACJA MIĘDZYPROCESOWA

Plan wykładu

- Wprowadzenie
- Mechanizmy IPC V
 - Kolejki komunikatów
 - Pamięć współdzielona
 - Semaforey
- Mechanizmy POSIX
 - Kolejki komunikatów
 - Pamięć współdzielona
 - Semaforey nazwane i nienazwane
- Gniazda

Podstawowe mechanizmy komunikacji i synchronizacji



Facility type	Name used to identify object	Handle used to refer to object in programs
Pipe	no name	file descriptor
FIFO	pathname	file descriptor
UNIX domain socket	pathname	file descriptor
Internet domain socket	IP address + port number	file descriptor
System V message queue	System V IPC key	System V IPC identifier
System V semaphore	System V IPC key	System V IPC identifier
System V shared memory	System V IPC key	System V IPC identifier
POSIX message queue	POSIX IPC pathname	<i>mqd_t</i> (message queue descriptor)
POSIX named semaphore	POSIX IPC pathname	<i>sem_t</i> * (semaphore pointer)
POSIX unnamed semaphore	no name	<i>sem_t</i> * (semaphore pointer)
POSIX shared memory	POSIX IPC pathname	file descriptor
Anonymous mapping	no name	none
Memory-mapped file	pathname	file descriptor
<i>flock()</i> lock	pathname	file descriptor
<i>fcntl()</i> lock	pathname	file descriptor

Część mechanizmów komunikacji wyposażona jest w środki synchronizacji, część wymaga stosowania niezależnych mechanizmów synchronizacji.

Mechanizmy IPC

- Wątki
- **Mechanizmy IPC**
 - Kolejki komunikatów
 - Semafor
 - Pamięć współdzielona
- Mechanizmy POSIX
- Gniazda

Identyfikatory i klucze

System V wprowadził trzy rodzaje mechanizmów komunikacji międzyprocesowej: **kolejki komunikatów, zestawy semaforów i pamięć współdzieloną**.

- Każdy z zasobów IPC ma określony **klucz** oraz **identyfikator** (32-bitowe nieujemne liczby całkowite).
- **Klucz** stanowi pewnego rodzaju "nazwę" zasobu, na podstawie której możemy uzyskać jego identyfikator. Nazwa ta musi być unikalna w systemie (w obrębie danego typu zasobu), oraz muszą ją znać wszystkie procesy korzystające z danego zasobu.
- **Identyfikator** jest przydzielany przez system podczas otwierania zasobu (odpowiednik "deskryptora pliku") - musimy go przekazać do każdej funkcji systemowej operującej na danym zasobie.
- Dostęp do zasobów IPC kontrolowany jest analogicznie jak dla plików - każdy zasób jest własnością pewnego **użytkownika i grupy**, oraz ma określone prawa **odczytu i zapisu** dla użytkownika, grupy i innych (bit określający prawa do wykonywania jest ignorowany).

Dostępne funkcje

	Kolejki komunikatów	Semaforey	Pamięć współdzielona
Plik nagłówkowy	<code><sys/msg.h></code>	<code><sys/sem.h></code>	<code><sys/shm.h></code>
Funkcja systemowa tworzenia lub otwierania	<code>msgget</code>	<code>semget</code>	<code>shmget</code>
Funkcja systemowa operacji sterujących	<code>msgctl</code>	<code>semctl</code>	<code>shmctl</code>
Funkcje operacji na obiektach IPC	<code>msgsnd</code> <code>msgrcv</code>	<code>semop</code>	<code>shmat</code> <code>shmdt</code>

Polecenia konsoli związane z IPC: `ipcs`, `ipcrm`

Mechanizmy IPC – kolejki komunikatów

- Wątki
- **Mechanizmy IPC**
 - **Kolejki komunikatów**
 - Pamięć współdzielona
 - Semaforey
- Mechanizmy POSIX
- Gniazda

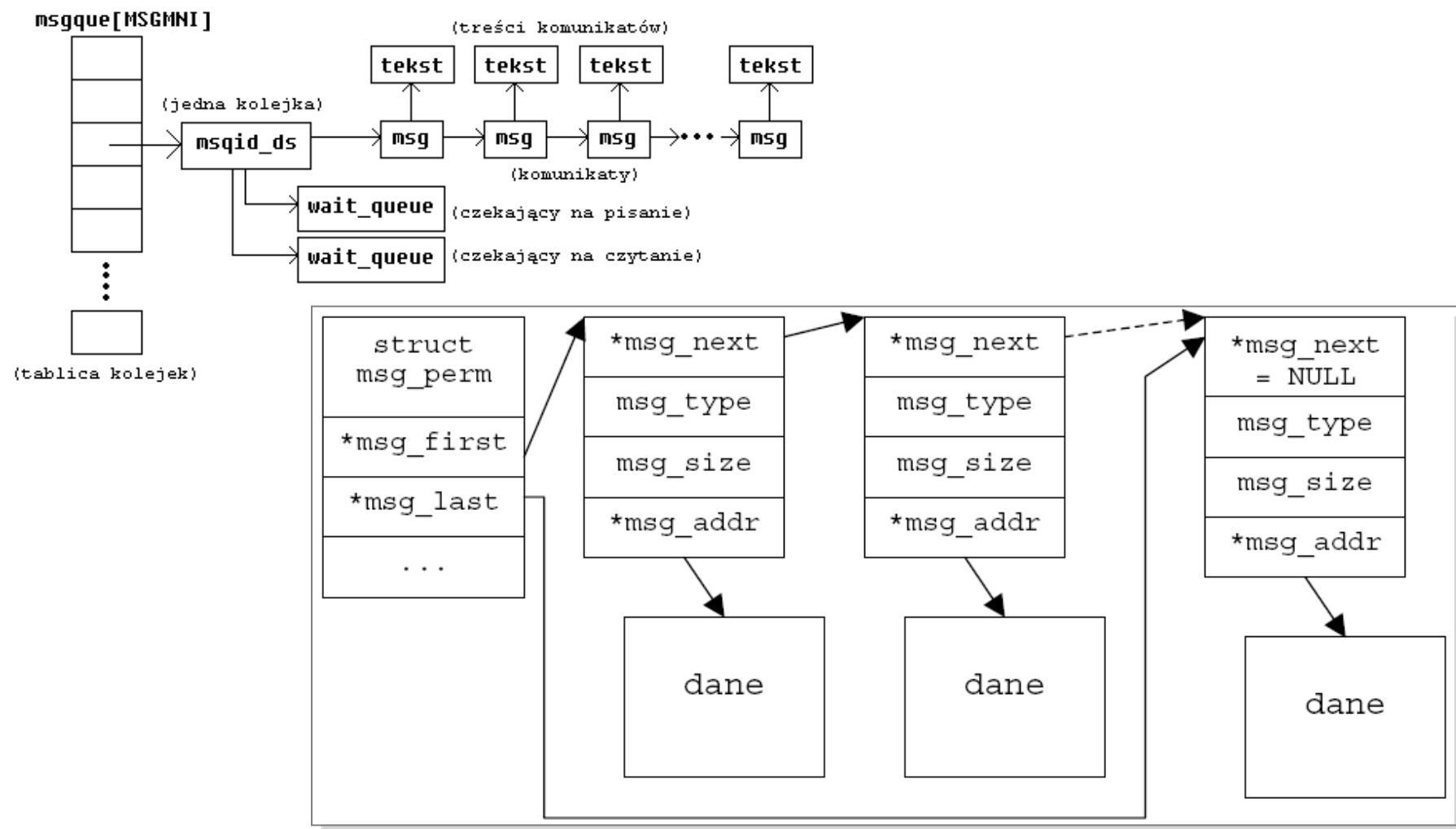
Kolejki komunikatów (1/2)

- Kolejki komunikatów umożliwiają przesyłanie pakietów danych, nazywanych komunikatami, pomiędzy różnymi procesami. Komunikat jest zbudowany jako struktura:

```
struct msgbuf{  
    long mtype; //typ komunikatu (>0)  
    char mtext[1]; //treść komunikatu  
}
```

- Komunikat ma określony **typ i długość**. Typ komunikatu, pozwalający określić rodzaj komunikatu, nadaje proces inicjujący komunikat.
- Komunikaty są umieszczane w kolejce w kolejności ich wysyłania.
- Nadawca może wysyłać komunikaty nawet wówczas, gdy żaden z potencjalnych odbiorców nie jest gotów do ich odbioru (komunikaty są buforowane).
- Przy odbiorze komunikatu odbiorca może oczekiwać na pierwszy przybyły komunikat lub na pierwszy komunikat określonego typu.
- Komunikaty w kolejce są przechowywane nawet po zakończeniu procesu nadawcy tak długo, aż nie zostaną odebrane lub kolejka nie zostanie zlikwidowana.

Kolejki komunikatów (2/2)



Funkcja `msgget`

- `int msgget(key_t key, int msgflg);`
- F-cja zwraca identyfikator kolejki związanej z kluczem *key*. Jeżeli jako klucz podamy **IPC_PRIVATE** albo nie istnieje kolejka o podanym kluczu (a we flagach ustawimy **IPC_CREAT**) to zostanie stworzona nowa kolejka.
- Znaczniki **IPC_CREAT** i **IPC_EXCL** przekazywane parametrem *semflg* pełnią tę samą rolę w obsłudze kolejek komunikatów, co **O_CREAT** i **O_EXCL** w parametrze *mode* funkcji systemowej `open`.

Funkcja `msgctl` (1/2)

- `int msgctl(int msqid, int cmd, struct msqid_ds *buf);`
- F-cja wykonuje operację określoną przez parametr *cmd* na kolejce komunikatów o identyfikatorze *msqid*.
- Możliwe wartości *cmd*:
 - **IPC_STAT** - kopiowanie informacji ze struktury kontrolnej kolejki komunikatów *msqid* pod adres *buf*.
 - **IPC_SET** - zapis wartości niektórych pól struktury *msqid_ds* wskazywanej przez parametr *buf* do struktury kontrolnej kolejki komunikatów *msqid*.
 - **IPC_RMID** - usunięcie kolejki komunikatów i skojarzonej z nią struktury danych.

Funkcja msgctl (2/2)

Zawartość struktury msqid_ds:

```
struct msqid_ds {  
    struct ipc_perm msg_perm; /* Własności i uprawnienia */  
    time_t          msg_stime; /* Czas ostatniego msgsnd() */  
    time_t          msg_rtime; /* Czas ostatniego msgrcv() */  
    time_t          msg_ctime; /* Czas ostatniej zmiany */  
    unsigned long   msg_cbytes; /* Bieżąca liczba bajtów w  
                                kolejce*/  
    msgqnum_t       msg_qnum;  /* Bieżąca liczba komunikatów w  
                                kolejce */  
    msglen_t        msg_qbytes; /* Maksymalna liczba dostępnych  
                                bajtów w kolejce */  
    pid_t           msg_lspid;  /* PID ostatniego msgsnd() */  
    pid_t           msg_lrpid;  /* PID ostatniego msgrcv() */  
};
```

Funkcja `msgsnd`

- `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);`
- F-cja służy do wysyłania komunikatu *msgp* o długości *msgsz* do kolejki *msqid*. Komunikat musi rozpoczynać się polem typu **long int** określającym typ komunikatu, po którym umieszczone zostaną pozostałe bajty wiadomości. Przykładowo może być to struktura:

```
struct mymsg {  
    long int mtype; /* message type */  
    char mtext[1]; /* message text */  
}
```

- **W celu wysłania lub odebrania komunikatu, proces powinien zaalokować strukturę danych!**

Funkcja `msgrcv` (1/2)

- `ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);`
- F-cja odczyta komunikat z kolejki wskazanej przez *msqid* do struktury *msgbuf* wskazywanej przez *msgp* usuwając odczytany komunikat z kolejki.
- Parametr *msgsz* określa maksymalny rozmiar (w bajtach) pola *mtext* struktury wskazywanej przez parametr *msgp*. Dłuższe komunikaty są obcinane (tracone) jeżeli flaga ustawiona jest na **MSG_NOERROR**, w przeciwnym przypadku komunikat nie jest usuwany z kolejki.

Funkcja `msgrcv` (2/2)

Parametr *msgtyp* określa rodzaj komunikatu w następujący sposób:

- Jeśli jest **równy 0**, to czytany jest pierwszy dostępny komunikat w kolejce (czyli najdawniej wysłany).
- Jeśli ma wartość **większą niż 0**, to z kolejki odczytywany jest pierwszy komunikat wskazanego typu, chyba że w parametrze *msgflg* zostanie ustawiony znacznik **MSG_EXCEPT**, kiedy to z kolejki zostanie odczytany pierwszy komunikat o typie innym niż podany w *msgtyp*.
- Jeśli *msgtyp* ma wartość **mniejszą niż 0**, to z kolejki zostanie odczytany pierwszy komunikat o najniższym numerze typu, o ile jest on mniejszy lub równy wartości bezwzględnej *msgtyp* .

Użycie kolejki komunikatów (1/2)

test-msg.c [1/2]

```
...  
int main()  
{  
    key_t key = ftok( ".", 'z');  
    int id = msgget( key, IPC_CREAT | 0600);  
    struct { long type; char a[10]; } data;  
    int r;  
    data.type = 2; strcpy( data.a, "hello");  
    msgsnd( id, ( struct msgbuf*)&data,  
            sizeof( data) - 4, 0);  
    data.type = 1; strcpy( data.a, "world");  
    msgsnd( id, ( struct msgbuf*)&data,  
            sizeof( data) - 4, 0);  
    ...  
}
```


Użycie kolejki komunikatów (2/2)

test-msg.c [2/2]

```
...  
for(;;){  
    r = msgrcv( id, ( struct msgbuf*)&data,  
               sizeof(data) - 4,  
               -2,  
               IPC_NOWAIT);  
    if(r<0)  
        break;  
    puts(data.a);  
}  
msgctl( id, IPC_RMID, NULL);  
}  
...
```

Mechanizmy IPC – pamięć współdzielona

- Wątki
- **Mechanizmy IPC**
 - Kolejki komunikatów
 - **Pamięć współdzielona**
 - Semaforey
- Mechanizmy POSIX
- Gniazda

Pamięć współdzielona

- Pamięć współdzielona jest specjalnie utworzonym **segmentem wirtualnej przestrzeni adresowej**, do którego dostęp może mieć wiele procesów. Jest to najszybszy sposób komunikacji pomiędzy procesami.
- Podstawowy schemat korzystania z pamięci współdzielonej wygląda następująco: jeden z procesów tworzy segment pamięci współdzielonej, dowiązuje go powodując jego **odwzorowanie w bieżący obszar danych procesu**, opcjonalnie zapisuje w stworzonym segmencie dane.
- Następnie, w zależności od praw dostępu inne procesy mogą odczytywać i/lub zapisywać wartości w pamięci współdzielonej.

Funkcja `shmget`

- `int shmget(key_t key, size_t size, int shmflg);`
- Funkcja `shmget` służy do tworzenia segmentu pamięci współdzielonej i do uzyskiwania dostępu do już istniejących segmentów pamięci.
- W drugim przypadku wartością parametru *size* może być 0, ponieważ rozmiar segmentu został już wcześniej zadeklarowany przez proces, który go utworzył.

Funkcja `shmctl`

- `int shmctl(int shmid, int cmd, struct shmid_ds *buf) ;`
- Funkcja odpowiada funkcji `msgctl`. Przy próbie usunięcia segmentu odwzorowanego na przestrzeń adresową procesu system odpowiada komunikatem o błędzie.
- Możliwe wartości `cmd`:
 - **IPC_STAT** - pozwala uzyskać informację o stanie pamięci współdzielonej
 - **IPC_SET** - pozwala zmienić parametry segmentu pamięci
 - **IPC_RMID** - pozwala usunąć segment pamięci współdzielonej z systemu

```
struct shmid_ds {  
    struct ipc_perm shm_perm; /* Ownership and permissions */  
    size_t shm_segsz; /* Size of segment (bytes) */  
    time_t shm_atime; /* Last attach time */  
    time_t shm_dtime; /* Last detach time */  
    time_t shm_ctime; /* Creation time/time of last modification via shmctl() */  
    pid_t shm_cpid; /* PID of creator */  
    pid_t shm_lpid; /* PID of last shmat(2)/shmdt(2) */  
    shmatt_t shm_nattch; /* No. of current attaches */  
    ... };
```

Funkcja `shmat`

- `void *shmat(int shmid, const void *shmaddr, int shmflg);`
- F-cja **dołącza** segment pamięci wspólnej o deskryptorze *shmid* do **przestrzeni adresowej procesu**, który ją wywołał. Adres, pod którym segment ma być widoczny jest przekazywany parametrem *shmaddr*.
- Jeśli *shmaddr* jest równy **NULL**, wówczas system sam wybierze odpowiedni (nieużywany) adres, pod którym segment będzie widoczny.
- W wyniku poprawnego wykonania f-cja zwraca adres początku obszaru odwzorowania segmentu.

Funkcja `shmdt`

- `int shmdt(const void *shmaddr);`
- Funkcja `shmdt` wyłącza segment pamięci wspólnej odwzorowany pod adresem podanym w *shmaddr* z przestrzeni adresowej procesu wywołującego tę funkcję.
- Przekazany funkcji w parametrze *shmaddr* adres musi być równy adresowi zwróconemu wcześniej przez wywołanie `shmat`.

Użycie pamięci współdzielonej

test-shm.c

```
...
int main()
{
    key_t key = ftok( ".", 'z');
    int id = shmget( key, 1024, IPC_CREAT | 0600);
    char * base = shmat( id, NULL, 0);
    if( !base) return;

    printf( "Zawartość obszaru: %s\n", base);
    sprintf( base, "tu byłem (proces %d)", getpid());
    printf( "Zmieniłem na: %s\n", base);

    shmdt( base);
}
...
```

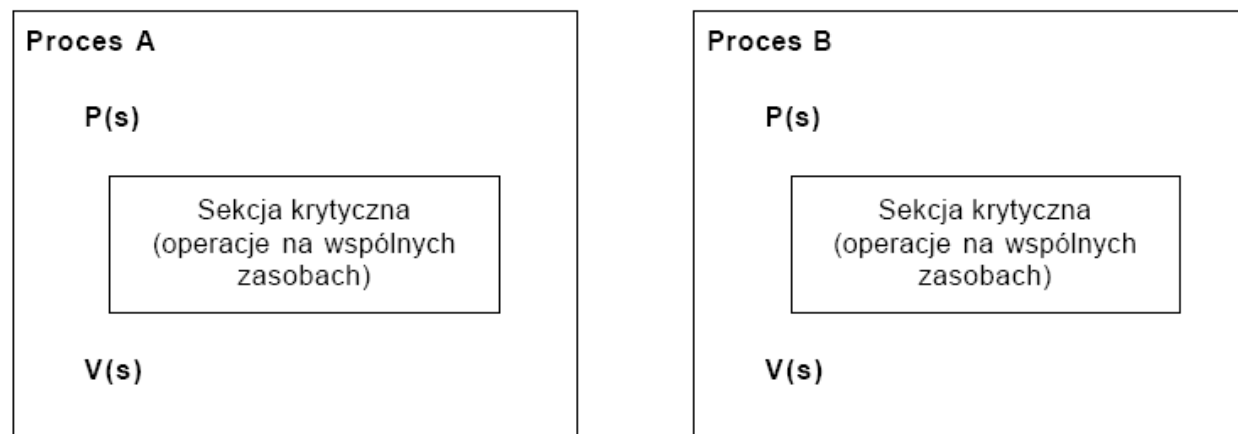

Mechanizmy IPC - semafor

- Wątki
- **Mechanizmy IPC**
 - Kolejki komunikatów
 - Pamięć współdzielona
 - **Semafor**
- Mechanizmy POSIX
- Gniazda

Semafor (1/2)

- Semafor są strukturami danych wspólnie użytkowanymi przez kilka procesów. Najczęściej znajdują one zastosowanie w **synchronizowaniu działania kilku procesów** korzystających ze wspólnego zasobu, przez co zapobiegają niedozwolonemu wykonaniu operacji na określonych danych jednocześnie przez większą liczbę procesów.
- Podstawowym rodzajem semafora jest semafor binarny, przyjmujący dwa stany:
 - opuszczony (**zamknięty**) - wówczas proces, który napotyka semafor musi zawiesić swoje działanie do momentu podniesienia semafora (opuść - **P**),
 - podniesiony (**otwarty**) - proces może kontynuować działanie (podnieś - **V**)

Semaforey (2/2)



W **IPC** zdefiniowano jedynie strukturę semafora i zbiór funkcji do operacji na nich, bez określania jakie wartości semafora odpowiadają jakim jego stanom. Możliwe są dwie konwencje:

Podejście 1

opuszczony ma **wartość = 0**

podniesiony ma **wartość > 0**

Podejście 2

opuszczony ma **wartość > 0**

podniesiony ma **wartość = 0**

Funkcja `semget`

- `int semget(key_t key, int nsems, int semflg);`
- F-cja zwraca identyfikator zestawu semaforów, skojarzonego z parametrem *key*. Jeśli *key* ma wartość **IPC_PRIVATE** lub, gdy z wartością *key* nie jest skojarzony żaden istniejący zestaw semaforów, a w parametrze *semflg* został przekazany znacznik **IPC_CREAT** to tworzony jest nowy zestaw złożony z *nsems* semaforów.
- Znaczniki **IPC_CREAT** i **IPC_EXCL** przekazywane parametrem *semflg* pełnią tę samą rolę w obsłudze semaforów, co **O_CREAT** i **O_EXCL** w parametrze *mode* funkcji systemowej `open`.

Funkcja `semctl` (1/2)

- `int semctl(int semid, int semnum, int cmd, union semun *arg);`
- F-cja wykonuje operację sterującą określoną przez *cmd* na zestawie semaforów *semid* lub na *semnum*-tym semaforze tego zestawu (numeracja od 0).
- W zależności o wydanego polecenia *cmd* argument *arg* jest różnie interpretowany:

```
union semnum{  
    int val;  
    struct semid_ds *buf;  
    unsigned short *array;  
};
```

```
struct semid_ds {  
    struct ipc_perm sem_perm; /* Ownership and permissions */  
    time_t sem_otime; /* Last semop time */  
    time_t sem_ctime; /* Creation time/time of last modification via semctl() */  
    unsigned long sem_nsems; /* No. of semaphores in set */  
};
```

Funkcja `semctl` (2/2)

Możliwe wartości *cmd*:

- **IPC_STAT** Kopiowanie informacji ze struktury kontrolnej zestawu semaforów do struktury wskazywanej przez *arg.buf*
- **IPC_SET** - modyfikuje wybrane ustawienia zestawu semaforów
- **IPC_RMID** - usuwa zestaw semaforów z systemu
- **GETVAL** - zwraca wartość semafora (*semval*), wskazywanego jako *semnum*
- **SETVAL** - nadaje wartość semaforowi o numerze *semnum*
- **GETPID** - zwraca wartość *sempid*
- **GETNCNT** - pobranie liczby procesów oczekujących na to, aż semafor wskazywany przez *semnum* zwiększy swoją wartość
- **GETZCNT** - pobranie liczby procesów oczekujących na to, aż semafor wskazywany przez *semnum* osiągnie wartość zero
- **GETALL** - pobranie bieżących parametrów całego zestawu semaforów i zapisanie uzyskanych wartości w tablicy wskazanej czwartym argumentem funkcji
- **SETALL** - zainicjowanie wszystkich semaforów z zestawu wartościami przekazanymi w tablicy określonej przez wskaźnik przekazany czwartym argumentem funkcji

Funkcja `semop` (1/3)

- `int semop(int semid, struct sembuf *sops, size_t nsops);`
- Wykonanie operacji semaforowej (a dokładniej ich zestawu). Może być ona wykonywana jednocześnie na kilku semaforach w tym samym zestawie identyfikowanym przez *semid*.
- *sops* to wskaźnik na adres tablicy operacji semaforowych, a *nsops* to liczba elementów tej tablicy.
- Każdy element tablicy opisuje jedną operację semaforową i ma następującą strukturę:

```
struct sembuf {  
    short sem_num; /* numer semafora - od 0 */  
    short sem_op;  /* operacja semaforowa */  
    short sem_flg; /* flagi operacji */  
};
```

Funkcja `semop` (2/3)

- Pole `sem_op` zawiera wartość, która zostanie dodana do zmiennej semaforowej pod warunkiem, że zmienna semaforowa nie osiągnie w wyniku tej operacji wartości **mniejszej od 0**. Dodatnia liczba całkowita oznacza zwiększenie wartości semafora (co z reguły oznacza zwolnienie zasobu), ujemna wartość `sem_op` oznacza zmniejszenie wartości semafora (próbę pozyskania zasobu).
- Funkcja `semop` podejmuje próbę wykonania wszystkich operacji wskazywanych przez `sops`. Gdy chociaż jedna z operacji nie będzie możliwa do wykonania nastąpi blokada procesu lub błąd wykonania funkcji `semop`, zależnie od ustawienia flagi **IPC_NOWAIT** i żadna z operacji semaforowych zdefiniowanych w tablicy `sops` nie zostanie wykonana.

Funkcja semop (3/3)

	podejście 1 (0-semafor opuszczony)		podejście 2 (0-semafor podniesiony)			
operacja:	- opuść semafor (jeżeli opuszczony to czekaj aż ktoś go podniesie i natychmiast opuść)	- podnieś	- opuść semafor (jeżeli opuszczony to czekaj aż ktoś go podniesie i natychmiast opuść)	- opuść	- czekaj aż ktoś podniesie	- podnieś
nsops	1	1	2	1	1	1
sem_op	-1	+1	0 +1	+1	0	-1

Użycie semaforów (1/2)

test-sem.c [1/2]

```
...  
struct sembuf sb;  
  
int main()  
{  
    key_t key = ftok( ".", 'z');  
    int id = semget( key, 1, IPC_CREAT | 0600);  
    int i = 0;  
    int pid;  
  
    struct semid_ds buf;  
    semctl( id, 0, SETVAL, 1);  
    semctl( id, 0, IPC_STAT, &buf);  
    pid = fork();  
    ...  
}
```

Użycie semaforów (2/2)

test-sem.c [2/2]

...

```
for(i=0;i<3;i++){
```

```
    sb.sem_num = 0; sb.sem_op = -1; sb.sem_flg = 0;  
    semop( id, &sb, 1);
```

```
    printf( "%d: Korzystam\n", getpid());
```

```
    sleep( rand()%3 + 1);
```

```
    printf( "%d: Przestałem korzystać\n", getpid());
```

```
    sb.sem_op = 1;
```

```
    semop( id, &sb, 1);
```

```
}
```

```
if( pid){
```

```
    wait( NULL);
```

```
    semctl( id, IPC_RMID, 0);
```

```
}
```

```
}
```

Gniazda

- Wątki
- Mechanizmy IPC
- **Mechanizmy POSIX**
- Gniazda

POSIX: IPC - przegląd

Interface	Message queues	Semaphores	Shared memory
Header file	<code><mqqueue.h></code>	<code><semaphore.h></code>	<code><sys/mman.h></code>
Object handle	<code>mqd_t</code>	<code>sem_t *</code>	<code>int</code> (file descriptor)
Create/open	<code>mq_open()</code>	<code>sem_open()</code>	<code>shm_open() + mmap()</code>
Close	<code>mq_close()</code>	<code>sem_close()</code>	<code>munmap()</code>
Unlink	<code>mq_unlink()</code>	<code>sem_unlink()</code>	<code>shm_unlink()</code>
Perform IPC	<code>mq_send()</code> , <code>mq_receive()</code>	<code>sem_post()</code> , <code>sem_wait()</code> , <code>sem_getvalue()</code>	operate on locations in shared region
Miscellaneous operations	<code>mq_setattr()</code> —set attributes <code>mq_getattr()</code> —get attributes <code>mq_notify()</code> —request notification	<code>sem_init()</code> —initialize unnamed semaphore <code>sem_destroy()</code> —destroy unnamed semaphore	(none)

POSIX: IPC - podstawowe założenia

- Interfejs obsługi mechanizmów IPC zgodnych z POSIXem został maksymalnie upodobniony do interfejsu obsługi plików (nazwy, flagi, prawa dostępu).
- Kolejki komunikatów, pamięć współdzielona i semaforey nazwane są identyfikowane jednoznacznie za pomocą **nazw**. Nazwy muszą być **unikalne** w ramach systemu i pozwalają na dostęp do obiektów IPC różnym procesom.
- Nazwa to ciąg znaków do długości 256 znaków. Nazwa ZAWSZE zaczyna się od znaku slash '/'. Uwaga! Pomimo występowania znaku slash nazwa **nie jest ścieżką!**
- Semaforey nienazwane nie używają nazw (posługujemy się adresami). Możemy ich użyć:
 - w przypadku synchronizacji wątków (zamiast użycia mutexów),
 - w przypadku synchronizacji procesów (umieszczają semafor w pamięci współdzielonej przez procesy).
- ...

POSIX: semafony

Nazwane

Semafor mają unikalne w ramach systemu nazwy i mogą być używane przez współpracujące procesy

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

Nienazwane

Semafony nie mają nazwy, dostęp do nich to dostęp do odpowiednio zainicjowanej zmiennej globalnej. Stąd najprościej używać je w wątkach jednego procesu. Jeżeli mają być użyte w różnych procesach, to procesy te muszą współdzielić pamięć (!).

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

```
int sem_wait(sem_t *sem);
```

...

```
int sem_post(sem_t *sem);
```

```
int sem_close(sem_t *sem);  
int sem_unlink(const char *name);
```

```
int sem_destroy(sem_t *sem);
```

biblioteka -lpthread

POSIX: semafony nazwane

```
...
int main(int ac, char **av) {
    if(ac > 1) {
        if (ac == 3 && av[2][0] == 'u') {
            sem_unlink(av[1]); printf("semaphore deleted\n");
            return 0;
        }

        sem_t *sem = sem_open(av[1], O_RDWR | O_CREAT, 0600, 1);

        sem_wait(sem); printf("semaphore down!\n");
        int i = 0;
        while(i<4) {
            printf("%d\n", i++);
            sleep(1);
        }
        sem_post(sem); printf("semaphore up!\n");
        sem_close(sem);
    }

    return 0;
}
```

biblioteka -lpthread

pts/1

```
$ ./sem /abc
semaphore down!
0
1
2
3
semaphore up!
$
```

nazwa rozpoczyna się znakiem slash (nie jest to ścieżka!)

pts/2

```
$ ./sem /abc

semaphore down!
0
1
2
3
semaphore up!

$ ls /dev/shm
sem.abc
$ ./sem /abc u
semaphore deleted
$ ls /dev/shm
$
```


POSIX: semafony nienazwane

...

```
void *thread(void *sem) {  
    pthread_t tid = pthread_self();  
    sem_wait(sem); printf("(%lu) semaphore down!\n", tid);  
    int i = 0;  
    while(i<4) {  
        printf("(%lu): %d\n", tid, i++);  
        sleep(1);  
    }  
    sem_post(sem); printf("(%lu) semaphore up!\n", tid);  
}
```

```
int main() {  
    sem_t sem;  
    pthread_t thr[2];  
    sem_init(&sem, 0, 1);  
  
    pthread_create(&thr[0], NULL, thread, &sem);  
    sleep(1);  
    pthread_create(&thr[1], NULL, thread, &sem);  
    pthread_join(thr[0], NULL);  
    pthread_join(thr[1], NULL);  
  
    sem_destroy(&sem);  
    return 0;  
}
```

wartość semafora
(1 – otwarty)

współdzielenie
(0 – wątki)

```
$ ./usem  
(140189459875584) semaphore down!  
(140189459875584): 0  
(140189459875584): 1  
(140189459875584): 2  
(140189459875584): 3  
(140189459875584) semaphore up!  
(140189451482880) semaphore down!  
(140189451482880): 0  
(140189451482880): 1  
(140189451482880): 2  
(140189451482880): 3  
(140189451482880) semaphore up!  
  
$
```

biblioteka -lpthread

POSIX: segmenty pamięci współdzielonej 1/2

...

```
#define SIZE 1024
```

```
int main(int ac, char **av) {
    int fd, retVal;
    char *buff;
    if(ac > 1) {
        if(ac == 2) {
            retVal = shm_unlink(av[1]);
            return 0;
        }

        fd = shm_open(av[1], O_RDWR | O_CREAT, 0600);
        ftruncate(fd, SIZE);
        buff = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        close(fd);
        if(buff) {
            printf("\tprevious: %s\n", buff);
            strcpy(buff, av[2]);
            printf("\tcurrent: %s\n", buff);
        }
        munmap(buff, SIZE);
    }
    return 0;
}
```

Plik musi mieć minimalnie rozmiar docelowej pamięci współdzielonej!!!

Bez tego uda się nam zamapować pamięć, ale przy próbie dostępu pojawi się sygnał SIGBUS i zakończy działanie procesu (błąd szyny).

Uchwyt nie jest już potrzebny

```
$ ./shm /abc 100
previous:
current: 100
```

```
$ ./shm /abc test
previous: 100
current: test
```

```
$ ls /dev/shm
abc
```

```
$ ./shm /abc
```

```
$ ls /dev/shm
```

```
$
```

biblioteka -lrt

POSIX: segmenty pamięci współdzielonej2/2

... tym razem używamy normalnych funkcji plikowych

...

```
#define SIZE 1024
```

```
int main(int ac, char **av) {
    int fd, retVal;
    char *buff;
    if(ac > 1) {
        if(ac == 2) {
            retVal = unlink(av[1]);
            return 0;
        }

        fd = open(av[1], O_RDWR | O_CREAT, 0600);
        ftruncate(fd, SIZE);
        buff = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        close(fd);
        if(buff) {
            printf("\tprevious: %s\n", buff);
            strcpy(buff, av[2]);
            printf("\tcurrent: %s\n", buff);
        }
        munmap(buff, SIZE);
    }
    return 0;
}
```

Nazwa to nazwa ścieżkowa pliku (czyli w tym przypadku plik w katalogu roboczym procesu).

```
$ ./shm abc 100
previous:
current: 100
```

```
$ ./shm abc test
previous: 100
current: test
```

```
$ ls .
abc mq
```

```
$ ./shm abc
```

```
$ ls
mq
```

```
$
```

POSIX: kolejka komunikatów

...

```
#define showAttr(attr) printf("getattr -> mq_curmsgs: %ld, mq_msgsize: %ld, mq_maxmsg: %ld\n", attr.mq_curmsgs, attr.mq_msgsize, attr.mq_maxmsg);
```

```
int main(int ac, char **av) {
    int prior, retVal;
    struct mq_attr attr;
```

```
    char *lorem[] = {"Lorem", "ipsum", "dolor", "sit", "amet", "consectetur", "adipiscing", "elit"};
```

```
    mqd_t mq = mq_open("/testmq", O_RDWR | O_CREAT | O_NONBLOCK, 0666, NULL); // showError( retVal, "mq_open", -1);
```

```
    mq_getattr(mq, &attr); showAttr(attr);
```

```
    retVal = mq_send(mq, lorem[0], strlen(lorem[0]) + 1, 2);
```

```
    retVal = mq_send(mq, lorem[1], strlen(lorem[1]) + 1, 4);
```

```
    retVal = mq_send(mq, lorem[2], strlen(lorem[2]) + 1, 1);
```

```
    retVal = mq_send(mq, lorem[3], strlen(lorem[3]) + 1, 4);
```

```
    retVal = mq_send(mq, lorem[4], strlen(lorem[4]) + 1, 2);
```

```
    mq_getattr(mq, &attr); showAttr(attr);
```

```
    int buffSize = attr.mq_msgsize;
```

```
    char *buff = malloc(buffSize);
```

```
    if(-1 != mq_receive(mq, buff, buffSize, &prior)) printf("\treceived: %s (%d)\n", buff, prior);
```

```
    if(-1 != mq_receive(mq, buff, buffSize, &prior)) printf("\treceived: %s (%d)\n", buff, prior);
```

```
    if(-1 != mq_receive(mq, buff, buffSize, &prior)) printf("\treceived: %s (%d)\n", buff, prior);
```

```
    mq_getattr(mq, &attr); showAttr(attr);
```

```
    retVal = mq_close(mq);
```

```
    retVal = mq_unlink("/testmq");
```

```
    return 0;
```

```
}
```

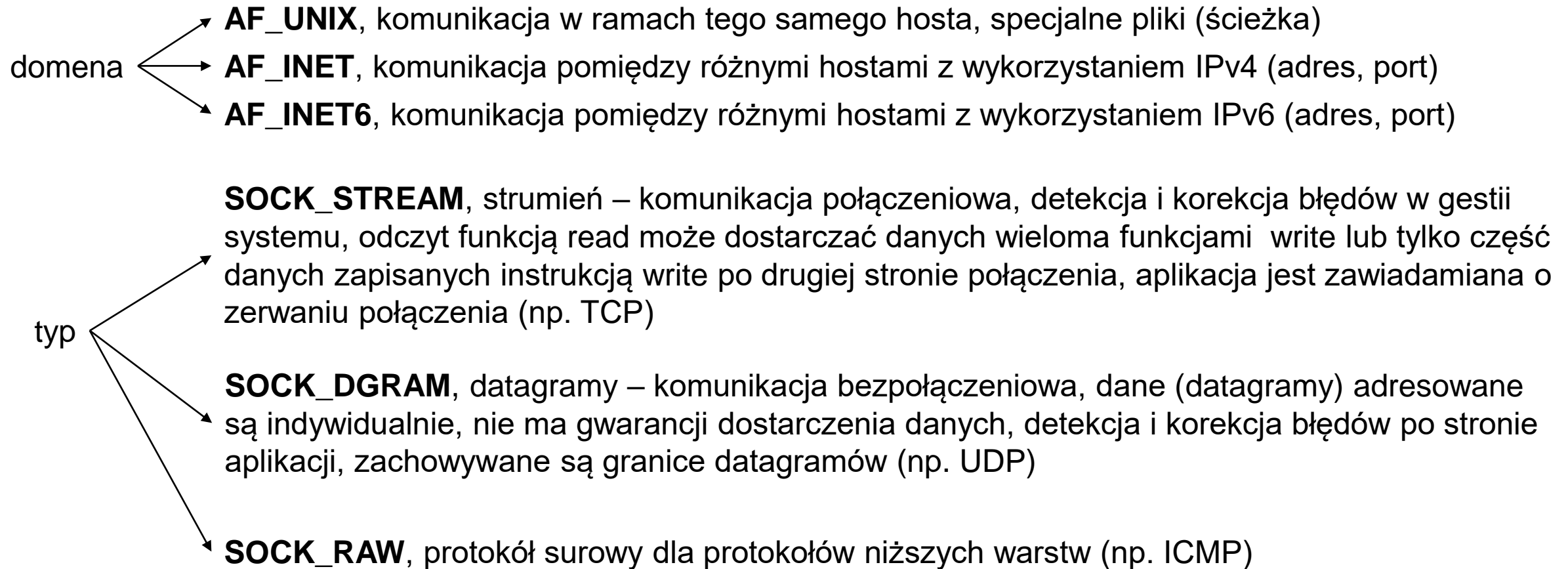
```
$ ./mq
getattr -> mq_curmsgs: 0, mq_msgsize: 8192, mq_maxmsg: 10
getattr -> mq_curmsgs: 5, mq_msgsize: 8192, mq_maxmsg: 10
received: ipsum (4)
received: sit (4)
received: Lorem (2)
getattr -> mq_curmsgs: 2, mq_msgsize: 8192, mq_maxmsg: 10
```

Próba odczytu do bufora o rozmiarze mniejszym niż **mq_msgsize** zakończy się błędem

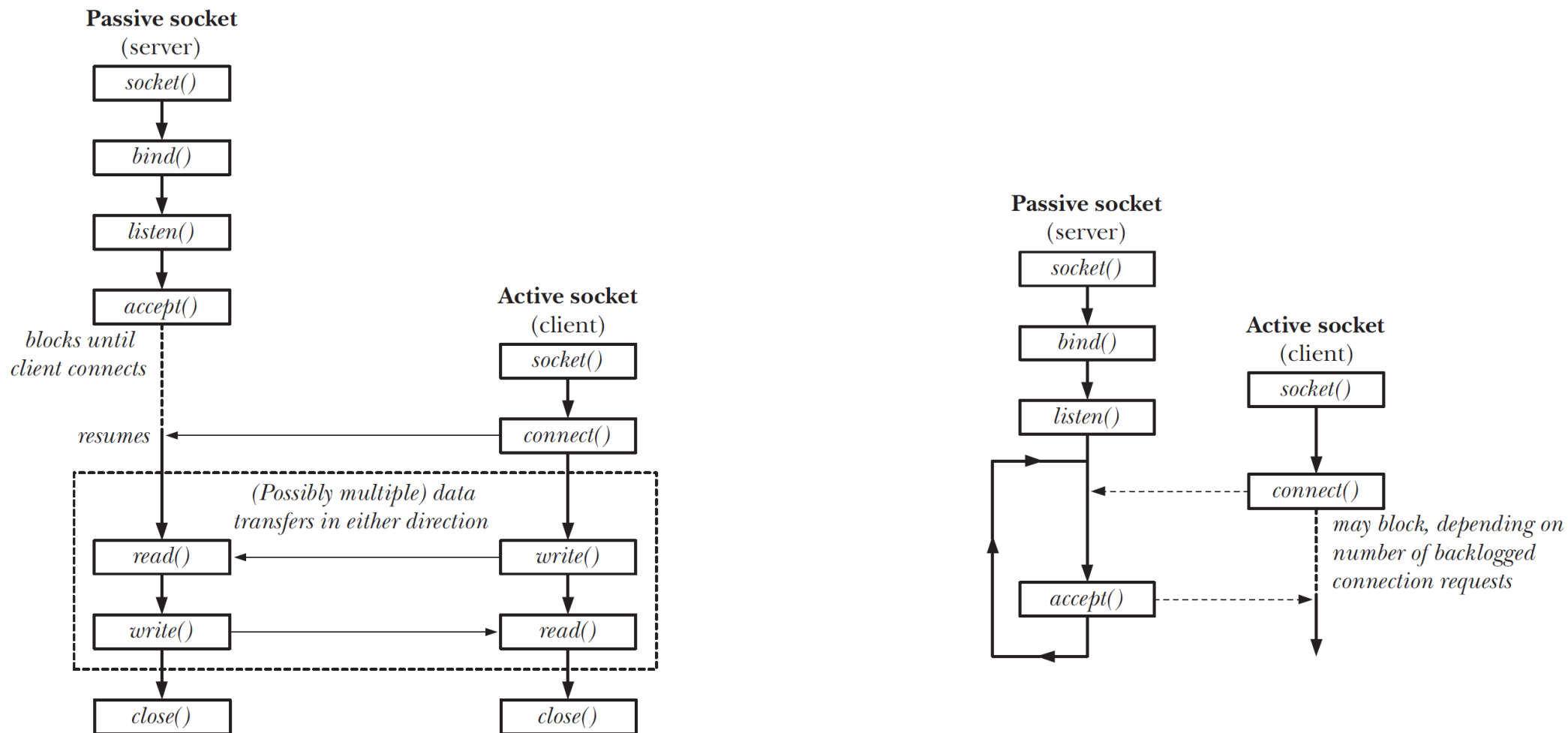
Gniazda

- Wątki
- Mechanizmy IPC
- Mechanizmy POSIX
- **Gniazda**

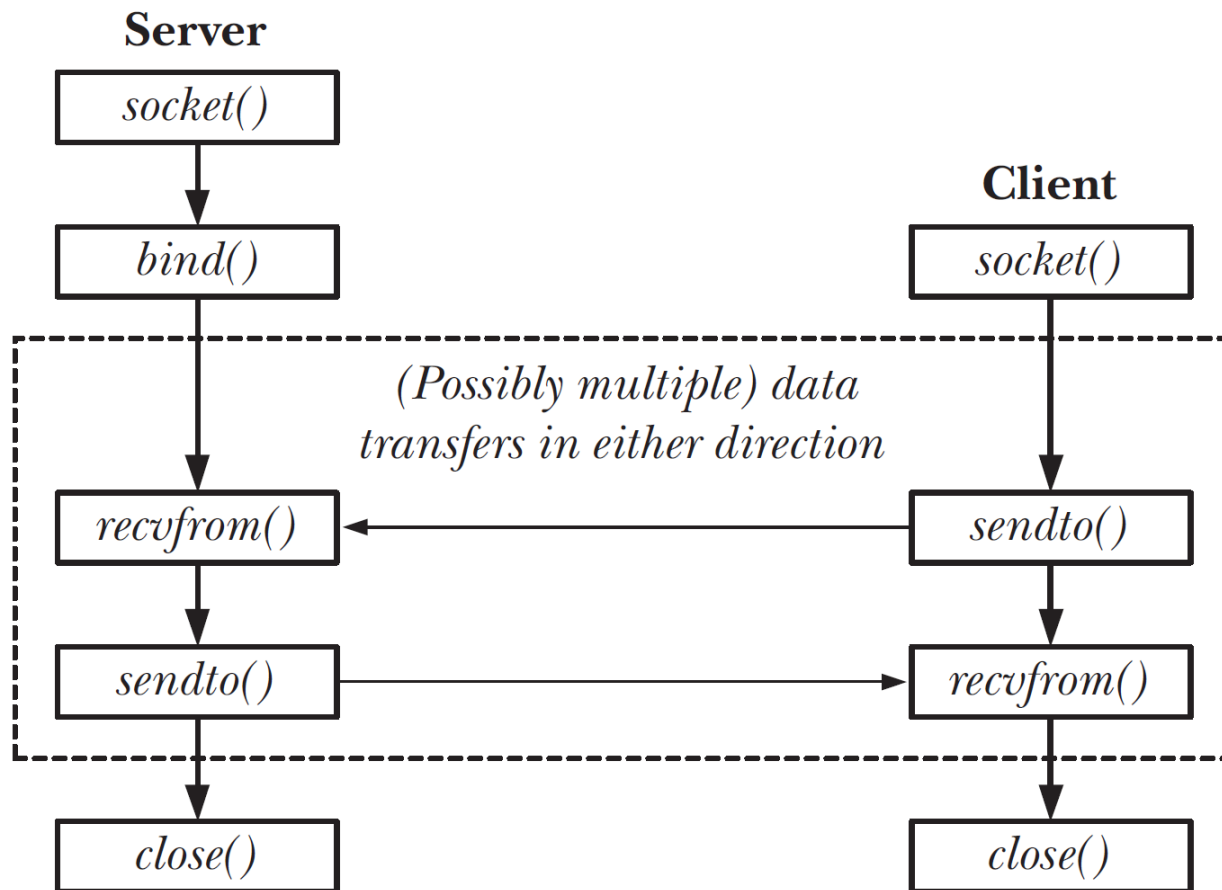
Gniazda



Gniazda – komunikacja połączeniowa



Gniazda – komunikacja bezpołączeniowa



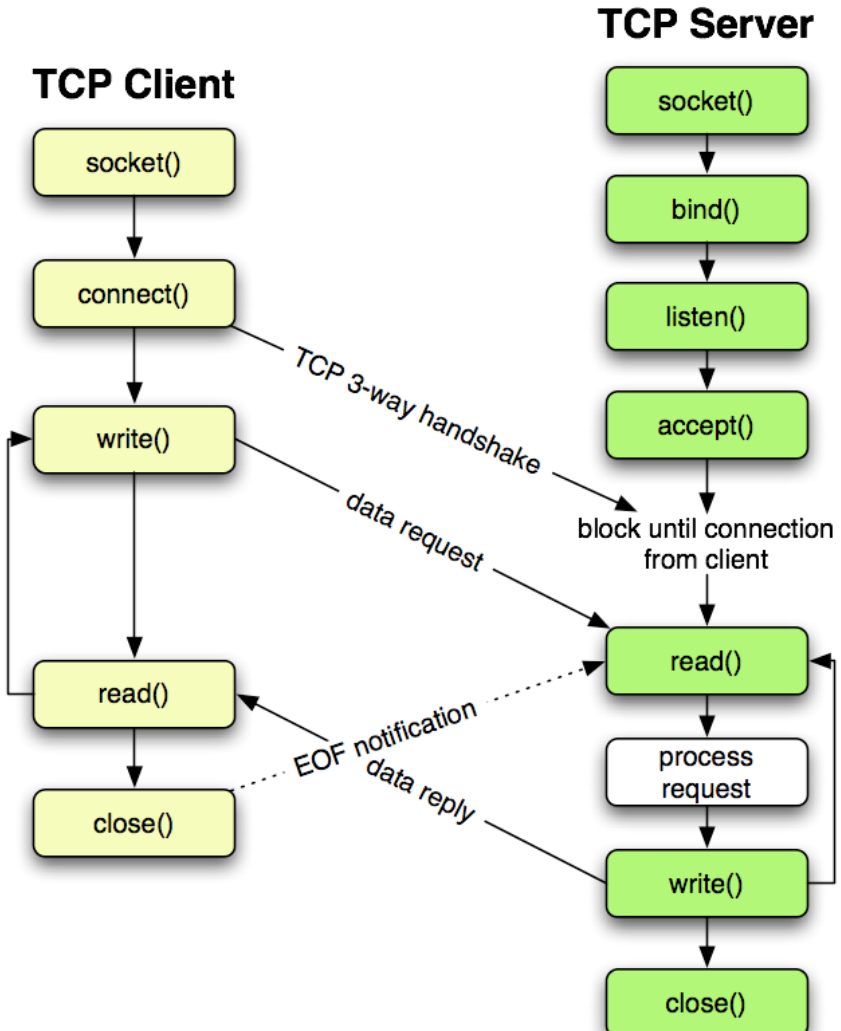
Do odbioru datagramu mogą być z powodzeniem używane zarówno **recvfrom** jak i **recv** oraz **read**

$\underbrace{\text{read} + \text{flagi}}_{\text{recv} + \text{adres nadawcy}}$
recvfrom

Do wysłania datagramu musimy użyć **sendto** (każdorazowo określamy adres odbiorcy)

Podstawowe funkcje - komunikacja połączeniowa

wspólne	<ul style="list-style-type: none"> • socket(2) – tworzenie gniazda • close(2) – usuwanie gniazda • read(2) – czytanie z gniazda • write(2) – pisanie do gniazda • recv(2) – czytanie z gniazda (dodatkowe opcje, np. z flagą MSG_PEEK nie usuwa wiadomości z kolejki) • send(2) – pisanie do gniazda
	<ul style="list-style-type: none"> • bind(2) – nadanie gniazdu serwera adresu (lokalnego lub internetowego) • listen(2) – konfigurowanie gniazda w celu przyjmowania połączeń • accept(2) – akceptowanie połączenia i tworzenie dla niego nowego gniazda
	<ul style="list-style-type: none"> • connect(2) – tworzenie połączenia między dwoma gniazdami
serwer	
klient	



Funkcja `socket` (1/2)

- `int socket(int domain, int type, int protocol);`

AF_UNIX, AF_LOCAL - komunikacja lokalna
AF_INET - protokół IPv4
AF_INET6 - protokół IPv6
...

SOCK_STREAM - komunikacja połączeniowa
SOCK_DGRAM - komunikacja bezpołączeniowa
...

Zazwyczaj dla konkretnej pary **domain** i **type** istnieje tylko jeden protokół, aby go użyć argument **protocol** ustawiamy na 0

Funkcja zwraca deskryptor nowego gniazda

Funkcja `socket` (2/2)

`test-local.c` [1/2]

```
...  
int lsfd;  
lsfd = socket (AF_UNIX, SOCK_STREAM, 0);  
...
```

`AF_UNIX`

`test-inet.c` [1/2]

```
...  
int isfd;  
isfd = socket (AF_INET, SOCK_STREAM, 0);  
...
```

`AF_INET`

Funkcja `bind` (1/3)

- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

Deskryptor utworzonego gniazda

Jeżeli przyjmujemy połączenia na każdym interfejsie to:
`sin_addr.s_addr = INADDR_ANY;`

```
struct sockaddr_un {  
    unsigned short sun_family; /* AF_UNIX */  
    char sun_path[108];  
};
```

`AF_UNIX`

```
struct sockaddr_in {  
    short    sin_family; /* AF_INET */  
    u_short  sin_port;   /* 16-bit port number */  
    struct in_addr sin_addr;  
    char     sin_zero[8]; /* unused */  
};
```

```
struct in_addr {  
    u_long    s_addr; /* 32-bit net id */  
};
```

`AF_INET`

Funkcja `bind` (2/3)

`test-local.c [2/2]`

```
...  
#define SERV_PATH "./serv.path"  
  
struct sockaddr_un serv_addr;  
  
serv_addr.sun_family = AF_UNIX;  
strcpy(serv_addr.sun_path, SERV_PATH);  
  
bind(lsf, (struct sockaddr *)&serv_addr, SUN_LEN(&serv_addr));  
...  
AF_UNIX
```

```
#define SUN_LEN(su) \  
    (sizeof(*(su)) - sizeof((su)->sun_path) + strlen((su)->sun_path))  
#endif
```

Funkcja `bind` (3/3)

`test-inet.c` [2/2]

```
...  
#define SERV_PORT 5232  
#define SERV_HOST_ADDR "82.145.73.240"  
  
struct addrinfo* res;  
  
hostinfo = getaddrinfo(Host name, IP4 or IP6 address SERV_HOST_ADDR, Service name or port SERV_PORT, hints NULL, result &res);  
  
bind (isfd, res->ai_addr, res->ai_addrlen);  
...
```

`AF_INET`

Automatycznie ustawiane pole z `AF_INET` albo `AF_INET6`

Tworzenie połączenia

- `int listen(int sockfd, int backlog);`

Określa, że gniazdo będzie oczekiwało na połączenia. Parametr `backlog` określa maksymalną długość kolejki przychodzących zgłoszeń. Zwraca `0` w przypadku sukcesu.

- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`

Wyciąga pierwsze żądanie połączenia z kolejki oczekujących połączeń, tworzy nowo podłączone gniazdo o tych samych właściwościach co `sockfd` i alokuje nowy deskryptor pliku dla gniazda (nowy deskryptor jest zwracany). Pod adres `addr` jest wpisywany adres łączącej się jednostki (przekazany przez warstwę komunikacyjną). **Jeżeli w kolejce brak połączeń funkcja blokuje proces.**

- `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

Inicjalizuje połączenie między gniazdem klienta a wskazanym adresem serwera. Zwraca `0` w przypadku sukcesu.

Konwersja adresów (string)

...

```
int main(int ac, char **av) {
    struct in_addr addr;
    char str[INET_ADDRSTRLEN];

    if(av[1]) {
        int retVal = inet_pton(AF_INET, av[1], &addr);
        if(retVal) {
            inet_ntop(AF_INET, &addr, str, INET_ADDRSTRLEN);
            printf("%s -> %s\n", av[1], str);
        }
    }
    return 0;
}
```

```
$ ./inet 127.0.0.1
```

```
127.0.0.1 -> 127.0.0.1
```

...

```
int main(int ac, char **av) {
    struct in6_addr addr;
    char str[INET6_ADDRSTRLEN];

    if(av[1]) {
        int retVal = inet_pton(AF_INET6, av[1], &addr);
        if(retVal) {
            inet_ntop(AF_INET6, &addr, str, INET6_ADDRSTRLEN);
            printf("%s -> %s\n", av[1], str);
        }
    }
    return 0;
}
```

```
$ ./inet6 0:0:0:0:0:0:0:1
```

```
0:0:0:0:0:0:0:1 -> 0::1
```

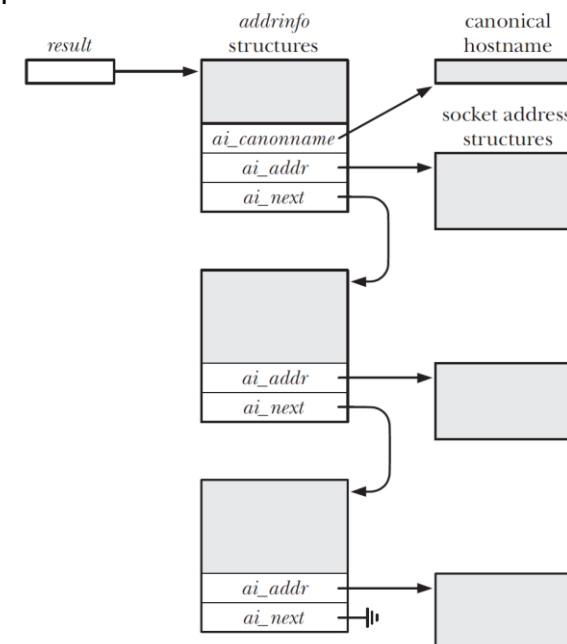

Funkcja getaddrinfo (1/2)

- `int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);`

Funkcja zwraca listę struktur typu **addrinfo** odpowiadających parametrom określonym w trzech pierwszych argumentach:

- **node** – nazwa hosta, adres IPv4 lub IPv6, może być NULL,
- **service** – nazwa usługi albo numer portu, może być NULL (ale zawsze albo node albo service musi być różny od NULL),
- **hints** – ograniczenie tworzonych struktur (), do wykorzystania pola: ai_flags, ai_family, ai_socktype i ai_protocol

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
};
```



- `void freeaddrinfo(struct addrinfo *res);`

Zwalniamy strukturę zwróconą do **res** przez **getaddrinfo**.

Funkcja getaddrinfo (2/2)

```
...
int main(int ac, char **av) {
    int passive = 0;
    char *node = NULL, *service = NULL;
    char c;

    while ((c = getopt(ac, av, "n:s:p")) != -1) {
        switch (c){
            case 'p': passive = 1; break;
            case 'n': node = optarg; break;
            case 's': service = optarg; break;
            default: abort ();
        }
    }

    struct addrinfo hints, *res, *p;
    memset (&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = passive ? AI_PASSIVE : 0; // addr fo passive socket (server)

    int retVal = getaddrinfo( node, service, &hints, &res);
    if(!retVal) {
        for (p = res; p!=NULL; p=p->ai_next){
            char str[INET_ADDRSTRLEN];
            struct sockaddr_in *x = (struct sockaddr_in *)(p->ai_addr);
            struct in_addr y = x->sin_addr;
            inet_ntop(AF_INET, &y, str, INET_ADDRSTRLEN);
            printf("%s:%d\n", str, ntohs(x->sin_port));
        }
    }

    return 0;
}
```

```
$ ./getaddrinfo -n www.wi.zut.edu.pl
82.145.72.60:0

$ ./getaddrinfo -n www.wi.zut.edu.pl -s http
82.145.72.60:80

$ ./getaddrinfo -n www.wi.zut.edu.pl -s https
82.145.72.60:443

$ ./getaddrinfo -n www.wi.zut.edu.pl -s 80
82.145.72.60:80

$ ./getaddrinfo -n 82.145.72.60 -s 80
82.145.72.60:80

$ ./getaddrinfo -s 80
127.0.0.1:80

$ ./getaddrinfo -s 80 -p
0.0.0.0:80
```

Funkcje ...sockopt

```
...
int main(int ac, char **av) {
    int sfdd = socket(AF_INET, SOCK_DGRAM, 0);
    int sfds = socket(AF_INET, SOCK_STREAM, 0);

    int optval;
    socklen_t optlen = sizeof(optval);
    getsockopt(sfdd, SOL_SOCKET, SO_TYPE, &optval, &optlen);
    printf("%s\n", optval & SOCK_DGRAM ? "datagram" : "stream");
    getsockopt(sfds, SOL_SOCKET, SO_TYPE, &optval, &optlen);
    printf("%s\n", optval & SOCK_DGRAM ? "datagram" : "stream");
    return 0;
}
```

```
$ ./sockopt
```

```
datagram
stream
```


```
...
int main(int ac, char **av) {
    int sfds = socket(AF_INET, SOCK_STREAM, 0);

    int optval = 1;
    socklen_t optlen = sizeof(optval);
    setsockopt(sfds, SOL_SOCKET, SO_REUSEADDR, &optval, &optlen);

    ...

    return 0;
}
```

Sockets API level



SO_REUSEADDR – wymagany dla większości serwerów TCP, bez tego nie możemy wiązać tego samego adresu IP i portu wielokrotnie.

„A socket is a 5 tuple (proto, local addr, local port, remote addr, remote port). SO_REUSEADDR just says that you can reuse local addresses. The 5 tuple still must be unique”

Obsługa wielu połączeń

- wątki
- procesy

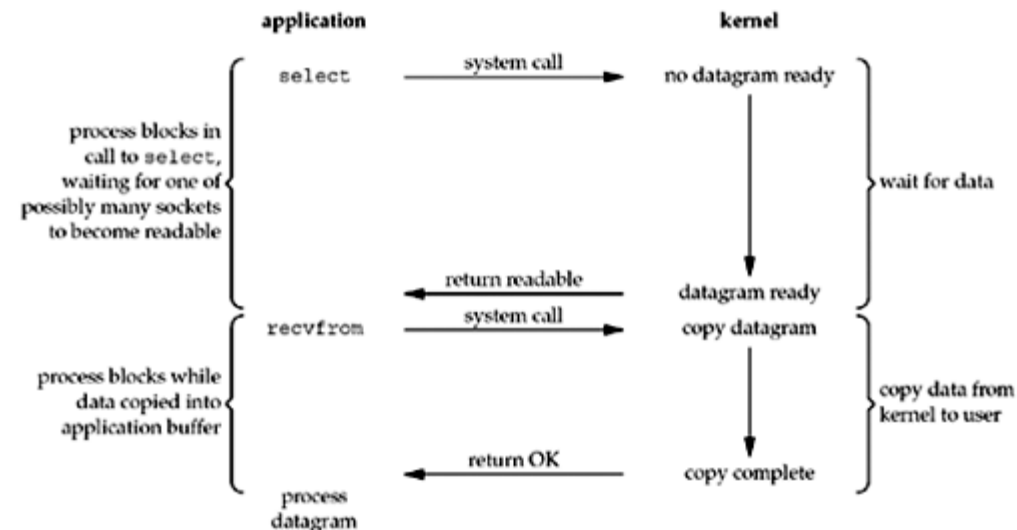


Wykorzystanie niezależnych procesów/wątków dla serwera oczekującego na połączenia oraz do obsługi poszczególnych połączeń – nieefektywne wykorzystanie zasobów!

- select (POSIX)
- poll (POSIX)
- ppoll (Linux)
- epoll (Linux)



Monitorowanie w jednym procesie/wątku wielu otwartych połączeń (deskryptorów) w oczekiwaniu na zmiany (I/O Multiplexing Model).



poll() (1/2)

```
#include <poll.h>

int poll (struct pollfd *fdarray, unsigned long nfd, int timeout);

/* Returns: count of ready descriptors, 0 on timeout, -1 on error */

struct pollfd {
    int      fd;          /* descriptor to check */
    short    events;      /* events of interest on fd */
    short    revents;     /* events that occurred on fd */
};
```

Constant	Input to <i>events</i> ?	Result from <i>revents</i> ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

poll() (2/2)

test-poll.c

```
...
struct pollfd fds[2];           // The structure for two events

fds[0].fd = sock1;
fds[0].events = POLLIN;         // Monitor sock1 for input

fds[1].fd = sock2;
fds[1].events = POLLOUT;        // Monitor sock2 for output

int ret = poll( &fds, 2, 10000 ); // Wait 10 seconds - repeat in a loop

if ( ret == -1 )                 // Check if poll actually succeed
    /* report error and abort */
else if ( ret == 0 )
    /* timeout; no event detected */
else
{
    if ( pfd[0].revents & POLLIN ) { // event detected, zero it out to reuse the structure
        pfd[0].revents = 0;
        /* input event on sock1, read data from pfd[0].fd */
    }
    if ( pfd[1].revents & POLLOUT ) {
        pfd[1].revents = 0;
        /* output event on sock2, write data to pfd[1].fd */
    }
}
...
```

Łącza strumieniowe `socketpair` (1/2)

- `int socketpair(int domain, int type, int protocol, int socket_vector[2]) ;`
- F-cja tworzy parę sprzężonych gniazd do komunikacji za pomocą tzw. łączy strumieniowych (ang. *stream pipe*). Łącze służy do komunikacji dwustronnej.
- Dostępna jest jedynie lokalna domena **AF_UNIX**. Typy to: **SOCK_STREAM**, **SOCK_DGRAM** i **SOCK_SEQPACKET**.
- Jeżeli nie wystąpił błąd (zwrócona jest wartość 0) to do wektora `socket_vector` jest zapisana para deskryptorów gniazd strumieniowych domeny UNIX.
- Komunikacja możliwa jedynie między spokrewnionymi procesami na jednej maszynie. Jedynie QNX pozwala na komunikację między procesami w różnych węzłach.

Łączy strumieniowe `socketpair` (2/2)

`test-socketpair.c`

```
#define DATA1 "abcde"
#define DATA2 "12345"

...

int sockets[2], child;
char buf[1024];

if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) == 0) {

    if ((child = fork()) > 0) { /* This is the parent. */
        close(sockets[0]);
        read(sockets[1], buf, sizeof(buf));
        printf("parent - write: %s, read: %s\n", DATA2, buf);
        write(sockets[1], DATA2, sizeof(DATA2));
        close(sockets[1]);
    } else { /* This is the child. */
        close(sockets[1]);
        write(sockets[0], DATA1, sizeof(DATA1));
        read(sockets[0], buf, sizeof(buf));
        printf("child - write: %s, read: %s\n", DATA1, buf);
        close(sockets[0]);
    }
}
```



parent - write: 12345, read: abcde
child - write: abcde, read: 12345