

Sieciowe Systemy Operacyjne – UNIX

Tomasz Surmacz

tsurmacz@ict.pwr.wroc.pl

Podstawy systemu UNIX

System UNIX

Początki – 1970: Ken Thompson, Dennis Ritchie (Bell Laboratories) na maszynie PDP-7, następnie PDP-11/20

- 1973 – pierwsza wersja z jądrem napisanym w C
- 1979 – BSD Unix (University of California, Berkeley)
- 1983 – Unix System V
- 1990 – Unix System V Release 4
- 1987 – Minix (A. Tanenbaum)
- 1991 – Linux 0.02 (Linus Torvalds)

Brian Kernighan – język C, powstał specjalnie w celu zapewnienia przenośności pisanego oprogramowania

Cechy systemu:

- system wieloużytkownikowy
- system wielozadaniowy (wiele procesów)
- podział czasu
- wywłaszczanie procesów
- podsystem plików
- zarządzanie pamięcią
- biblioteki systemowe
- wszystkie urządzenia dostępne przez pliki specjalne

3 podstawowe wersje: (1988)

- Version 7
- Berkeley – BSD 4.1 vs. BSD 4.2 (BSD4.4)
- System V – SysV Rel.3 vs. Sysv Rel. 2 (SVR4)

Podstawowe różnice:

- parametry wykonania komend (np. `ps -ef` lub `ps -aux`)
- różne koncepcje dostępu do terminali (strumienie w SysV)
- inna obsługa przerw systemowych (sygnałów)
- inne umiejscowienie standardowych programów

Standardy:

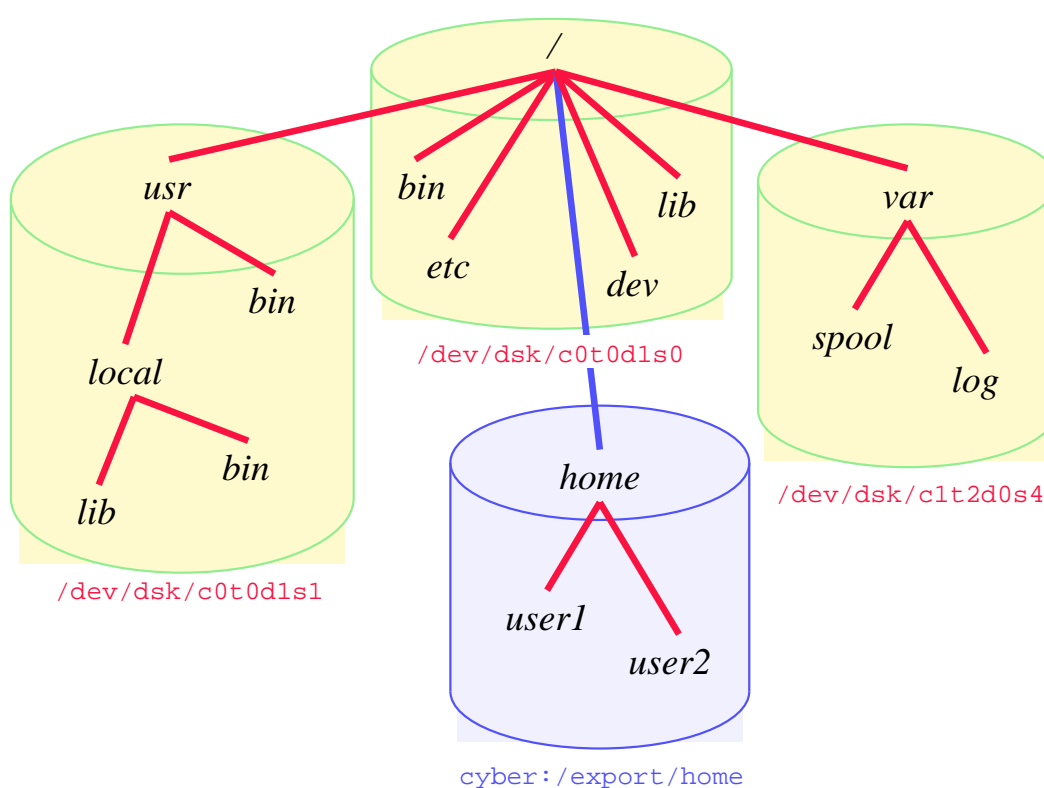
- ANSI C
- POSIX
- X/Open
- „Jedną z najwspanialszych zalet związanych ze standaryzacją jest to, że obowiązujących standardów jest tak wiele, że zawsze znajdzie się taki, który nam odpowiada.”

Licencje:

- Systemy komercyjne: SysV, SCO, Solaris, HPUX, OSF, ...
- FreeBSD, Linux
- FSF & GNU
- Licencja GPL

System plików

- Separatorem ścieżki jest znak „/”
- Katalog / to „korzeń” (*root*) systemu plików
- Wszystkie katalogi/dyski są podłączone jako gałęzie drzewa katalogów – zarówno lokalne, jak i zdalne.



- Różne rodzaje systemów plików: UFS, SysV, minix, ext2, MSDOS, NFS
- Komendy związane z plikami: *ls*, *cd*, *pwd*, *mkdir*, *rmdir*, *cat*, *mv*, *cp*, *rm*, *mount*, *umount*, *df*, *du*, *quota*
- Pliki: */etc/fstab*, */etc/mntab*, */etc/exports*

Zarządzanie pamięcią

- Pamięć fizyczna systemu podzielona jest na *segmenty*
- Rozmiar segmentu – z reguły 4-8-32 kB
- Dostęp do segmentów pamięci realizowany jest przez tablicę deskryptorów pamięci, z pomocą sprzętowych układów MMU
- Pamięć wirtualna
 - Swoboda adresowania
 - Możliwość zaadresowania większego obszaru pamięci niż dostępny
 - Stronicowanie pamięci/plik wymiany, dokonywane automatycznie

Podział pamięci w systemie UNIX:

- Pamięć jądra systemu
- Bufory dyskowe
- Plik(i) wymiany (*swap*)
- Pamięć procesów użytkowników
 - Kod programu (TEXT)
 - Dane
 - Stos
 - Stos wywołań systemowych

Biblioteki

- Język C określa konstrukcje sterujące wykonaniem programu
- Funkcje *nie są* częścią języka C, lecz bibliotek systemowych lub programów – nawet najbardziej podstawowe, jak *printf()*
- Funkcje systemowe znajdują się w bibliotekach dynamicznych, takich jak *libc.so*
- Biblioteki statyczne (np. */usr/lib/libc.a*)
 - dołączane w całości lub częściowo do kodu kompilowanego programu, w trakcie kompilacji
 - programy linkowane statycznie są większe, lecz nie wymagają działającego linkera *ld*
- Biblioteki dynamiczne (np. */usr/lib/libsocket.so.2*)
 - dołączane dynamicznie do kodu programu w trakcie jego uruchamiania
 - zajmują region pamięci współdzielony z innymi procesami
 - mniejsze zapotrzebowanie na pamięć
 - mniejsze programy
 - dynamiczny linker – programy *ld* i *ldd*
- Zmienna środowiskowa *LD_LIBRARY_PATH*
- Podstawowe biblioteki systemowe:
 - *libc.so* – standardowe funkcje wejścia/wyjścia, zarządzania pamięcią, itp.
 - *libm.so* – funkcje matematyczne
 - *libsocket.so* – funkcje sieciowe (SysV)
 - *libnsl.so* – „name server library” – tłumaczenie nazw komputerów na adresy
 - *libX11.so* – funkcje systemu okienkowego X11

Użytkownicy

- System wielozadaniowy (*multitasking*)
- System wieloużytkownikowy (*multiuser*)
- Konieczność ochrony użytkowników przed sobą nawzajem

Atrybuty użytkownika – */etc/passwd*:

```
root:x:0:1:Super-User:/root:/sbin/sh
ts:x:138:10:Tomasz Surmacz:/home/ts:/usr/local/bin/tcsh
nobody:x:60001:60001:Nobody:/:
```

- Nazwa użytkownika
- Identyfikator (uid)
- Grupa (nazwa i identyfikator – gid)
- Hasło dostępu
- Katalog domowy
- Interpreter poleceń (shell)

Dla jądra systemu istotne są:

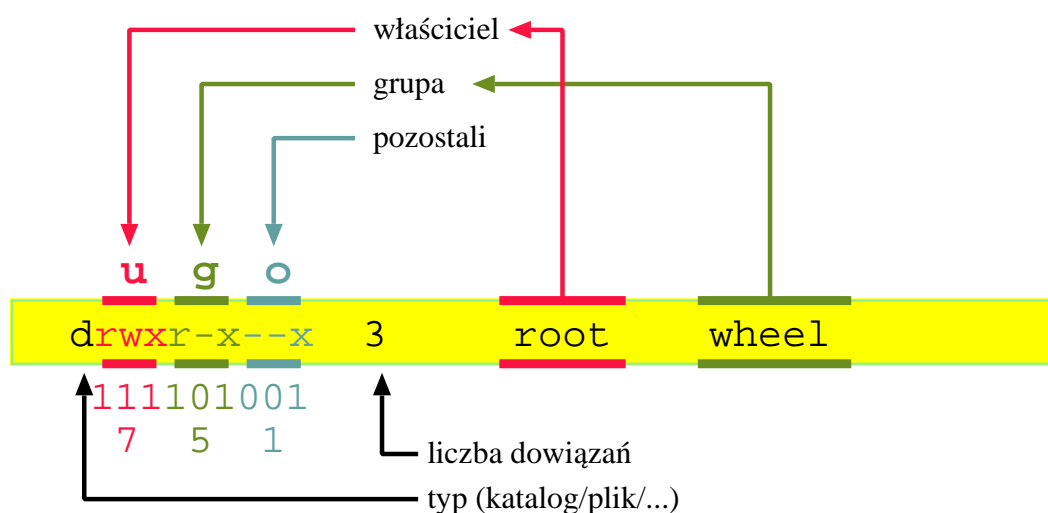
- uid procesu
- gid procesu
- prawa dostępu do pliku
- uprawnienia specjalne (uid==0)

Polecenia operujące na uid/gid:

- newgrp, su, login, id

Prawa dostępu

- Każdy plik w systemie ma swojego właściciela i właściciela grupowego
- Każdy plik opisany jest przez 3 grupy praw dostępu:
 - dla właściciela (user)
 - dla grupy (group)
 - dla pozostałych użytkowników (other)
- Podstawowe prawa dostępu to
 - (r)ead – odczyt
 - (w)rite – pisanie
 - e(x)ecute – wykonanie



Przykładowa zawartość katalogu domowego:

```
drwxr-x--x  44 ts      users      2048 Feb 27 16:39 .
drwxr-xr-x  47 ts      users      3072 Feb 28 00:28 ..
-rw-----   1 ts      users      3476 May 25  1998 .cshrc
-rw-r--r--   1 ts      users          16 Jun 28  1999 .forward
```



```
drwx----- 2 ts      mail      1024 Feb 27 03:44 Mail
drwxr-xr-x  3 ts      users     1024 Jun  2  1998 News
drwxr-xr-x  2 ts      users     1024 Feb 19  1998 Done
drwxr-xr-x  3 ts      users     1024 Dec  1 14:31 hitch
-rw-r--r--  1 ts      wheel     4029 Feb 27 03:13 back.tar.gz
drwxr-xr-x  3 ts      users     1024 Feb 27 03:44 pwis
drwxr-xr-x  3 ts      users     1024 Feb 27 03:19 src
```

Komendy pozwalające zmieniać prawa dostępu:

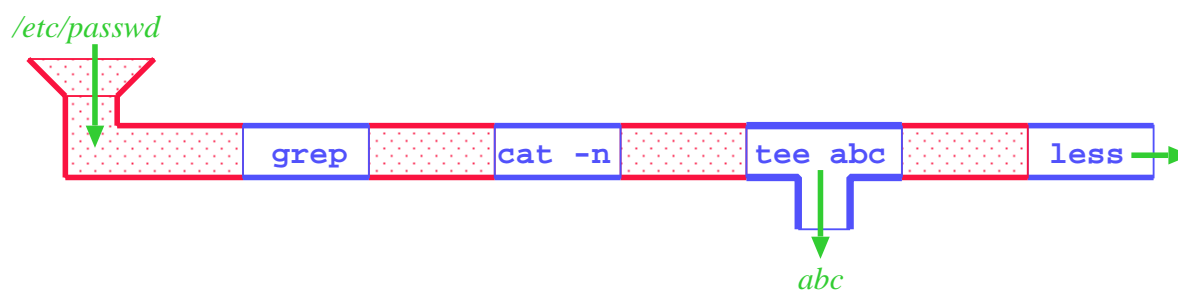
- `chmod`, `chgrp`
- `chown` (tylko użytkownik `root`!)
- `umask`

Bity specjalne:

- `-rwsr-xr-x` – `set-user-id`
- `-rwxr-sr-x` – `set-group-id`
- `-rw-----T` – `sticky bit`
- `drwxr-sr-x` – `set-group-id`
- `drwxrwxrwt` – `sticky bit`
- `prw-rw-rw-` – strumień (*named pipe*)
- `srw-rw-rw-` – gniazdo w domenie UNIX (*UNIX-domain socket*)
- `brw-rw----` – urządzenie blokowe (*block device*)
- `crw-rw----` – urządzenie znakowe (*character device*)
- `lrwxrwxrwx` – dowiązanie symboliczne (*symbolic link*)

Wejście/wyjście

- standardowe wejście – `stdin`
- standardowe wyjście – `stdout`
- wyjście błędów – `stderr`
- strumień pipe
- przekierowanie we/wy: `<` `>` `<<` `>>` `|`



```
grep :0: < /etc/passwd | cat -n | tee abc | less
```

- system plików
- katalogi, podkatalogi, pliki

Interpreter poleceń – shell

- `/bin/sh`
- `/bin/csh`
- `bash`
- `tcsh`

Środowisko programistyczne

Edytory tekstu:

- `vi`
- `emacs`
- `jed`
- `joe`

Kompilatory:

- `cc`
- `gcc`
- `g++`
- Program `make` wywołujący odpowiednie kompilatory i linker
- Linkowanie – `ld` lub `gcc`
- Pliki nagłówkowe: `/usr/include`, `/usr/local/include`
- Biblioteki: `/usr/lib`, `/usr/local/lib`

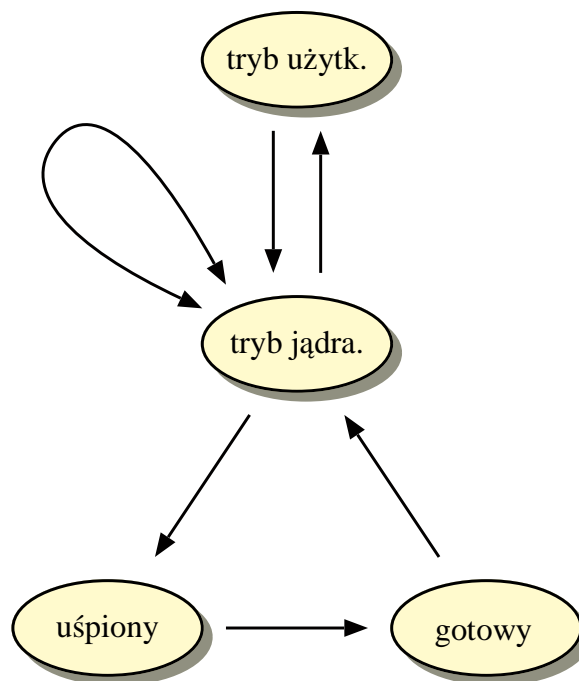
Uruchamianie programów:

- Debugger systemowy `adb`
- Debugger GNU `gdb`
- Ścieżka wykonania – katalog `.` (kropka)

Procesy

- tworzone funkcją *fork()*
- unikalny pid
- przodek (rodzic)—potomek (dziecko)
- grupa procesów i przewodnik grupy

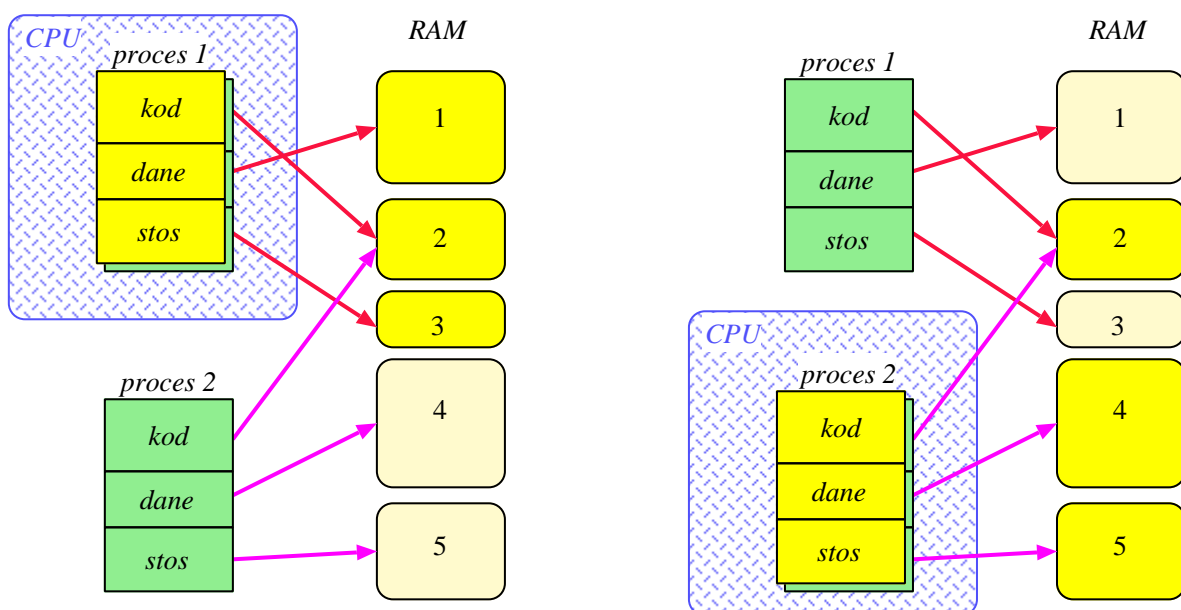
Stany procesów:



- przełączanie procesów/przełączanie kontekstu
- tablica procesów
- priorytety
- `nice`

Kontekst procesu

- Każdy proces posiada własne segmenty kodu, danych i stosu.
- Jądro systemu, poprzez system obsługi pamięci wirtualnej, w chwili wykonywania procesu ma dostęp do *jednego* segmentu kodu, jednego segmentu danych i jednego segmentu stosu.
- Tablica procesów zawiera wskaźniki do zapamiętanych *kontekstów* poszczególnych procesów – zawierających ich prywatne mapy odwzorowujące wszystkie trzy segmenty programu na segmenty pamięci fizycznej.



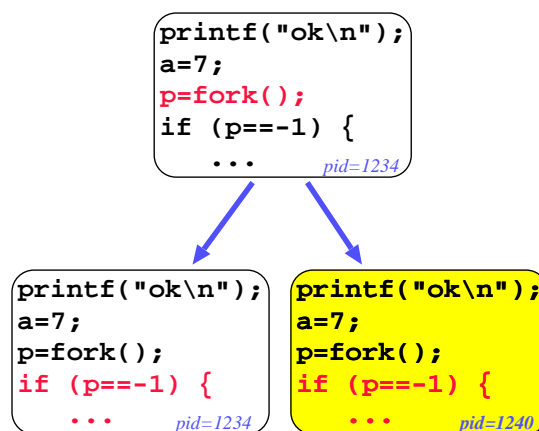
- *Przełączenie kontekstu* polega na zapamiętaniu aktualnego kontekstu procesu w jego obszarze kontekstu, po czym wczytaniu do procesora kontekstu innego procesu.

Sterowanie procesami z powłoki użytkownika

- Każdy proces zwraca wartość: 0 – ok, >0 – błąd
- Separator komend: znak ; lub & lub znak nowej linii
- Uruchamianie w tle: `proces &`
- Łączenie procesów strumieniem: `proces1 | proces2`
- Warunkowe wykonanie drugiego procesu:
 - jeśli pierwszy zakończył się poprawnie: `proces1 && proces2`
 - jeśli pierwszy zakończył się błędem: `proces1 || proces2`
- Przekierowanie wyjścia błędów:
 - csh: `proces >& plik` oraz `proces1 |& proces2`
 - sh: `proces > plik 2>&1`
- Zatrzymanie procesu: `stop %nazwa`, Klawisz Ctrl-Z, `kill -STOP`, `kill -SUSP`
- Zabicie procesu lub wysłanie sygnału: `kill`, `kill -WINCH`, itp.
- Ponowne uruchomienie zatrzymanego procesu: `fg`
- Ponowne uruchomienie procesu w tle: `bg`
- Sprawdzenie listy działających procesów: `jobs`, a także `ps`

Tworzenie nowych procesów

- Do tworzenia nowych procesów służy funkcja *fork()* ze standardowej biblioteki *libc.so*.
- Nowy proces, utworzony funkcją *fork()* jest dokładną kopią rodzica i wykonuje się w tym samym miejscu – za wywołaniem funkcji *fork()*



- Oba procesy nie różnią się niczym, poza wartością zwróconą przez funkcję *fork()*:
 - rodzic: numer procesu potomnego
 - dziecko: wartość 0
- Sposobem na rozróżnienie obu procesów jest zbadanie tej wartości:

```
pid=fork();
switch (pid) {
    case -1: printf("Błąd fork!\n");
            exit(1);

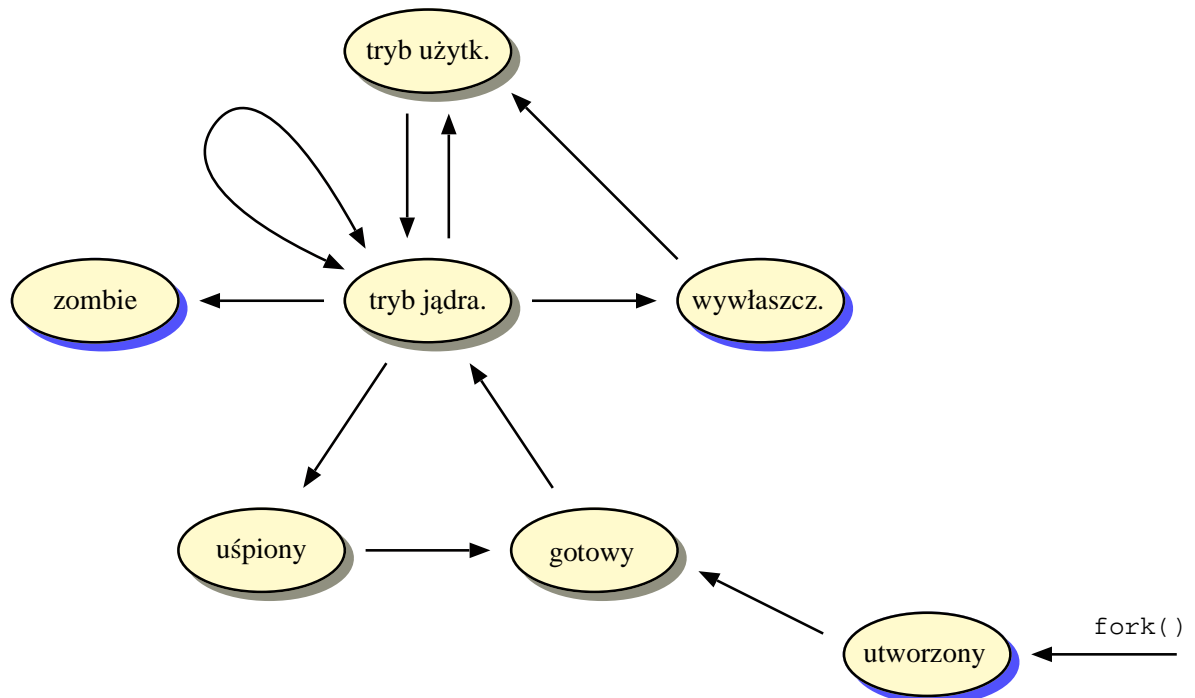
    case 0: /* dziecko */
            printf("Tu pisze dziecko. pid rodzica=%d\n", getppid());
            break;

    default: /* rodzic */
            printf("To drukuje rodzic. pid dziecka=%d\n", pid);
}
```

- Inne funkcje: *getpid()*, *getppid()*, *wait()*, *kill()*, *exec()*

Funkcja *fork()*

- przydziela nowemu procesowi pozycję w tablicy procesów
- przydziela nowemu procesowi nowy, unikalny identyfikator
- tworzy kopię kontekstu procesu macierzystego, kopiując segmenty pamięci lub zwiększając licznik odwołań do segmentu (np. współdzielonego segmentu kodu)
- otwarte pliki rodzica pozostawia otwarte w dziecku, a więc zwiększa liczniki w tablicy plików i i-węzłów
- przekazuje rodzicowi *pid* dziecka, a dziecku – wartość 0.
- nowy proces, choć nie był jeszcze nigdy wykonywany, „budzi się” tak, jakby zasnął w oczekiwaniu na zasób (wychodzi ze stanu uśpienia)



Funkcje *exec()*

- Rodzina funkcji: *execl()*, *execv()*, *execle()*, *execve()*, *execlp()*, *execvp()*
- 1. argument – ścieżka programu lub skryptu
- 2. argument i dalsze – parametry wywołania
- Wywołanie nadpisuje w aktualnym kontekście procesu segmenty kodu, danych i stosu nowym programem i wykonanie funkcji *main()*

```
if ((pid=fork())==0) {  
    execl("/bin/ls", "ls", "-la", "/tmp", NULL);  
    printf("Błąd!!! Ta instrukcja nie ma prawa się wykonać!\n");  
} else {  
    wait(NULL);  
}
```

Nowy proces dziedziczy:

- numer procesu i numer procesu rodzica oraz przynależność do grupy procesów
- wartość *nice*
- wartość *umask*
- priorytet
- *uid*, *gid* i przynależność do grup użytkowników
- katalog bieżący
- limity zasobów
- otwarte pliki¹
- kilka innych wartości dotyczących blokad na plikach, semaforów i obsługi sygnałów (*man exec*)

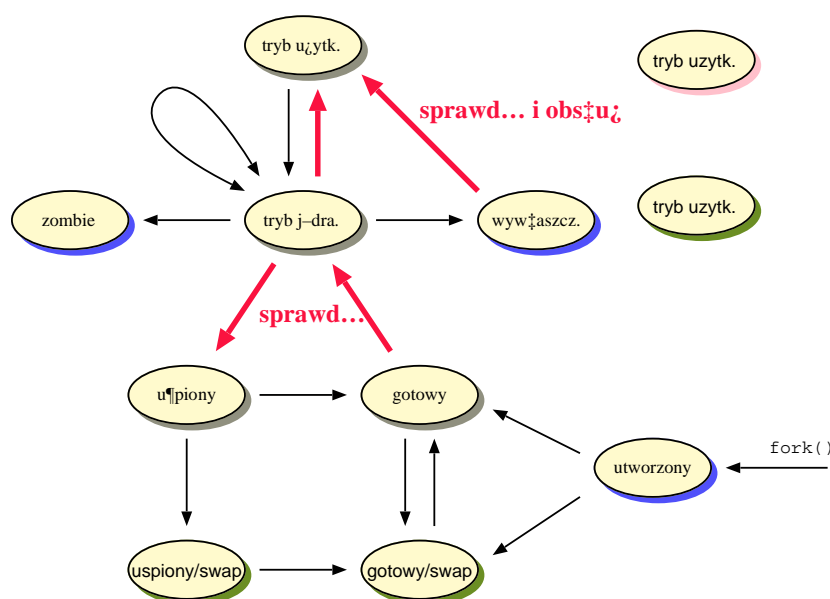
¹niekoniecznie! – uwaga na *fcntl()* i *FD_CLOEXEC*

Sygnały

Pozwalają zasygnalizować wystąpienie sytuacji specjalnej, takiej jak:

- śmierć potomka (przez wywołanie `exit()` lub w inny sposób)
- wyjątek – np. próba dostępu do pamięci poza przyznany zakres adresów, próba zapisu do pamięci z atrybutem read-only, itp.
- niespodziewany błąd podczas wykonywania programu (np. pisanie do łącza, którego już nikt nie czyta)
- błąd, z którym system nie potrafi sobie poradzić, np. brak pamięci na wykonanie `exec()` gdy już zostały zwolnione stare segmenty programu
- pobudka, czyli SIGALARM wysyłany na życzenie procesu przez system
- interakcja z terminalem – klawisz BREAK, SUSPEND, itp.
- wysłanie sygnału przez inny proces
- wykonywanie programu krok po kroku przez debugger

Sygnały są obsługiwane tylko przy przejściu między trybem jądra systemu a trybem użytkownika



- Listę dostępnych sygnałów można sprawdzić pisząc `kill -list` lub przeglądając plik `/usr/include/sys/signal.h` lub `/usr/include/unistd.h`

```
#define SIGHUP 1      /* hangup */
#define SIGINT 2      /* interrupt (rubout) */
#define SIGQUIT 3     /* quit (ASCII FS) */
...
#define SIGFPE 8      /* floating point exception */
#define SIGKILL 9      /* kill (cannot be caught or ignored) */
#define SIGBUS 10     /* bus error */
#define SIGSEGV 11     /* segmentation violation */
#define SIGSYS 12     /* bad argument to system call */
#define SIGPIPE 13     /* write on a pipe with no one to read it */
#define SIGALRM 14     /* alarm clock */
#define SIGTERM 15     /* software termination signal from kill */
#define SIGUSR1 16     /* user defined signal 1 */
#define SIGUSR2 17     /* user defined signal 2 */
...
#define SIGLOST 37     /* resource lost (eg, record-lock lost) */
...
```

- Część sygnałów jest domyślnie ignorowana, część powoduje zakończenie procesu
- Każdy sygnał z wyjątkiem `SIGKILL` i `SIGSTOP` można przechwycić, rejestrując odpowiednią funkcję obsługi sygnału za pomocą `signal()` lub `sigset()`
- Jeśli funkcja przechwytyująca ma ignorować sygnał, wystarczy wywołać `signal(SIG_IGN)` lub `sigignore()`
- Funkcją `sigpause(numer-sygnału)` można zawiesić proces aż do momentu otrzymania żadanego sygnału
- Na czas obsługi sygnału przyjmowanie sygnałów tego samego typu zostaje zablokowane automatycznie (a dodatkowo inne sygnały można blokować i odblokowywać wołając `sighold()` i `sigrelse()`)

Grupy procesów

- grupę może stanowić np. interpreter poleceń (shell) i wszystkie procesy przez niego uruchamiane – naciśnięcie BREAK itp. wysyła sygnał nie tylko do wykonywanego procesu, ale i do interpretera.
- ustawienie grupy – funkcja *setpgrp()*

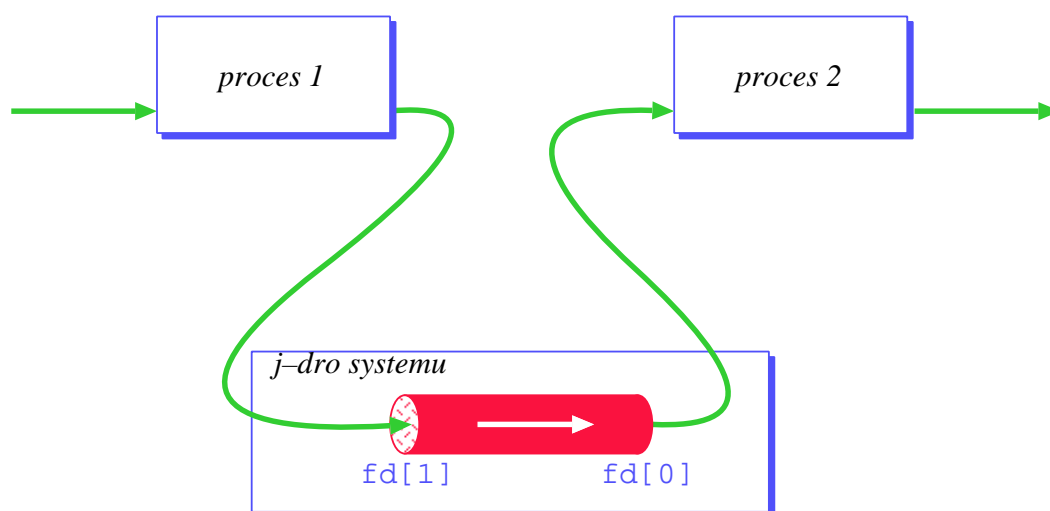
Kończenie procesu

- *wait()*
- *exit()*
- Zakończony proces staje się *zombie* i powoduje wysłanie do rodzica sygnału SIGCLD
- Wywołanie przez rodzica funkcji *wait()* pozwala odebrać status zwrócony przez *exit()*
- „uwolniony” *zombie* ostatecznie znika z systemu
- wywołanie *wait()*, gdy nie ma żadnego *zombie*, zawiesza proces, do momentu gdy któreś z dzieci zakończy żywot lub gdy już nie będzie żadnego.

Strumienie pipe i FIFO

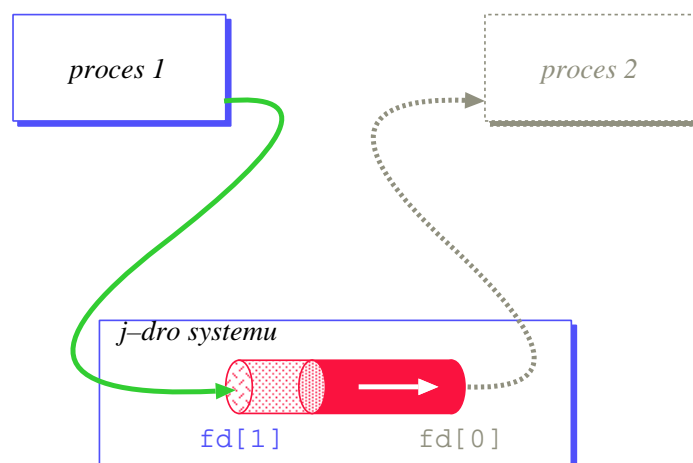
Strumienie pipe

- pozwalają łączyć spokrewnione ze sobą procesy
- transmisja jednokierunkowa
- z reguły łączą wyjście `stdout` jednego procesu z wejściem `stdin` innego



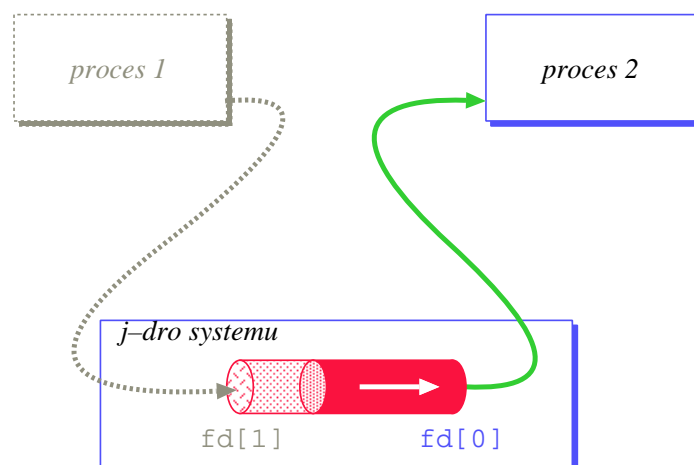
- tworzone funkcją `pipe(int fd[2])` – wypełnia 2-elementową tablicę deskryptorów
- `fd[1]` jest końcem otwartym do pisania
- `fd[0]` jest końcem otwartym do czytania
- oba końce to „zwykłe” deskryptory plików – można na nich operować takimi funkcjami jak `read()`, `write()`, `fprintf()`, czy `close()`.
- jądro systemu gwarantuje, że operacje zapisu nie przekraczające rozmiaru strumienia są niepodzielne

Pisanie do strumienia pipe



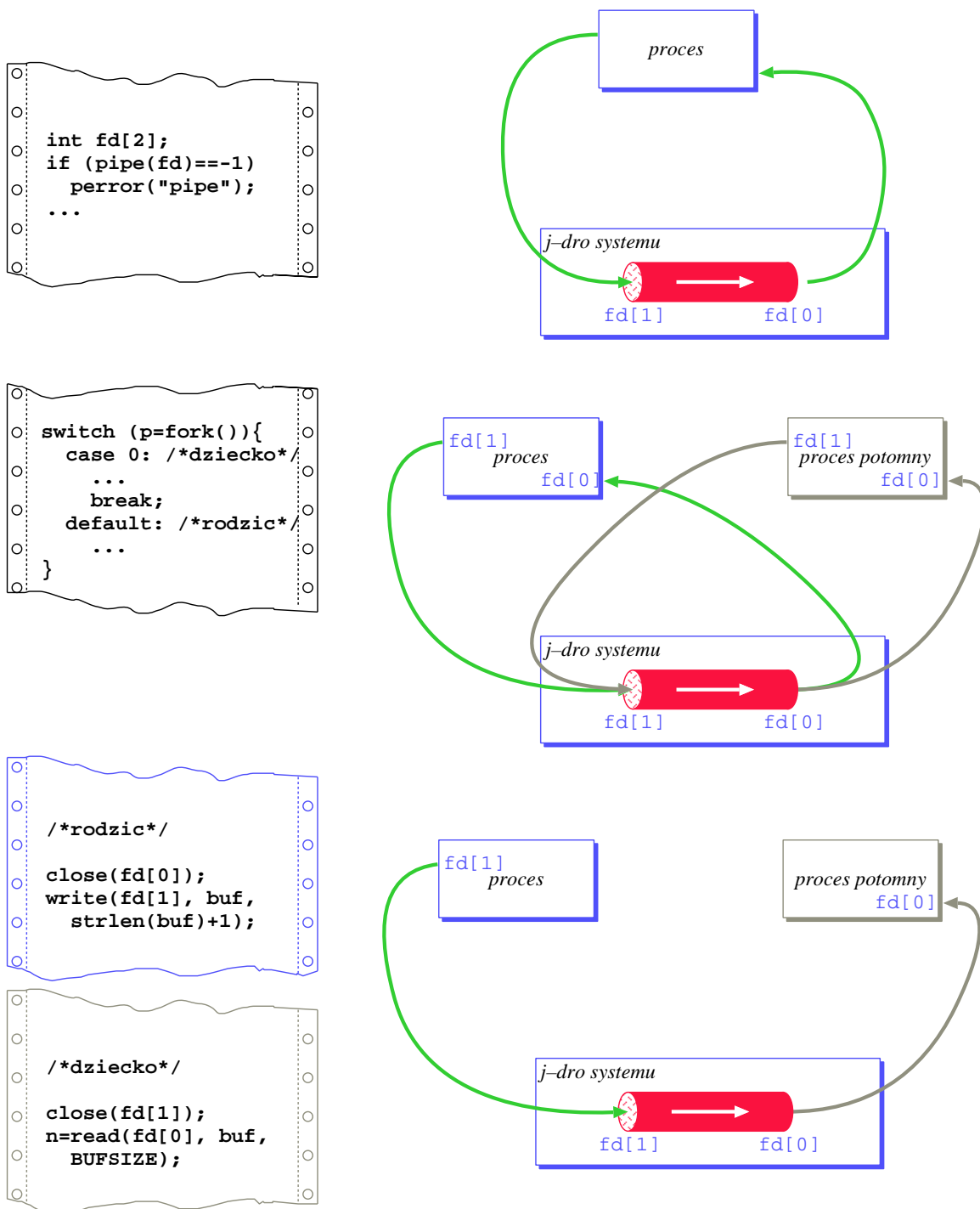
- strumień pipe pełni jednocześnie rolę bufora o pojemności 4kB
- zapisanie danych, gdy jest miejsce w strumieniu, powoduje natychmiastowy powrót z funkcji *write()* itp.
- zapisanie danych powyżej maksymalnego rozmiaru powoduje zablokowanie procesu, do czasu aż zwolni się miejsce w strumieniu (ktoś te dane przeczyta)
- za pomocą odpowiednich funkcji *fcntl()* lub *ioctl()* można ustawić opcję *O_NDELAY*, lecz wówczas proces piszący musi być przygotowany na to, że może mu się udać zapis tylko części wysyłanych danych
- pisanie do strumienia, którego nikt nie czyta (jego drugi koniec został zamknięty) powoduje wysłanie do procesu piszącego sygnału **SIGPIPE**
- zamknięcie zapisywanego końca powoduje, że proces czytający w wyniku wywołania funkcji *read()* otrzyma wartość 0

Czytanie ze strumienia pipe



- otrzymywane dane stanowią strumień nie podzielony na żadne pakiety lub inne fragmenty
- czytanie ze strumienia może zwrócić mniej danych, niż zażądano
- jeśli strumień jest pusty i zamknięty do zapisu, funkcja `read()` zwraca wartość 0, oznaczającą EOF.
- jeśli strumień jest pusty, ale otwarty do zapisu, funkcja `read()` zostaje zablokowana, do momentu gdy można będzie coś przeczytać
- blokowania można uniknąć stosując opcję `O_NDELAY` w funkcji `fcntl()` lub korzystając z funkcji `select()`
- zamknięcie strumienia do odczytu, gdy ciągle znajdowały się w nim nie przeczytane dane, generuje sygnał `SIGPIPE` wysyłany do procesu piszącego
- zamknięcie pustego strumienia danych nie powoduje żadnych konsekwencji dla procesu piszącego – dopóki nie spróbuje czegoś zapisać.
- dzięki blokowaniu możliwe jest wykorzystanie strumieni do wzajemnej synchronizacji procesów

Łączenie dwóch procesów strumieniem pipe



Tworzenie strumieni stdout-stdin

- Do kopiowania otwartych deskryptorów służą funkcje *dup()* i *dup2()*
- *dup(n)* kopiuje deskryptor *n* na najniższy wolny numer deskryptora
- Poniższy program:

```
int fdes;  
fdes=open("/tmp/test", "rt");  
close(0);  
dup(fdes);
```

spowoduje przepisanie deskryptora *fdes* do deskryptora nr 0.

- to samo można osiągnąć za pomocą *dup2(fdes, 0)*

Najczęstszym zastosowaniem jest zastąpienie standardowego wejścia (lub wyjścia) strumieniem pipe, przed wykonaniem funkcji *exec()*:

```
int fd[2];  
  
pipe(fd);  
if ((f=fork()) == 0) {  
    /* dziecko -- będzie czytać */  
    close(fd[1]);  
    close(0);  
    dup(fd[0]);  
    execlp("cat", "cat", "-n", NULL);  
    fprintf(stderr, "exec się nie udał!\n");  
    exit(1)  
} else {  
    /* rodzic lub błąd fork() */  
    ...  
    wait();  
}
```

Funkcja *popen()*

Podobny efekt można osiągnąć za pomocą funkcji *popen()*:

```
#include <stdio.h>
FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream);
```

- *popen()* tworzy strumień pipe oraz proces potomny, w którym wykonuje `/bin/sh`, przekazując mu jako parametr nazwę programu do wykonania
- jeśli `*type=='r'`, to standardowe wyjście uruchomionego podprocesu jest połączone ze strumieniem, a proces główny może czytać ze zwróconego deskryptora
- jeśli `*type=='w'`, zwrócony koniec jest otwarty do zapisu i połączony ze standardowym wejściem uruchomionego podprocesu
- zakończenie komunikacji może wymagać użycia *pclose()*, zawsze też konieczna jest synchronizacja za pomocą *wait()*.

Wady stosowania *popen()*:

- za każdym razem niepotrzebnie uruchamiany jest `/bin/sh`
- brak pełnej kontroli nad wejściem i wyjściem z podprocesu
- nie jest możliwe jednoczesne przechwycenie wejścia i wyjścia, ani rozdzielenie standardowego wyjścia od wyjścia błędów
- funkcja *popen()* NIGDY i pod żadnym pozorem nie powinna być stosowana w programach typu `set-user-id` (zbyt łatwo przeoczyć złe dane przekazywane do interpretera `sh`).

Strumienie FIFO

Zasadnicze różnice w stosunku do strumieni pipe:

- Posiadają dowiązanie w systemie plików – tworzone funkcją *mknod()* lub komendą *mknod*.
- Mogą być używane przez procesy nie spokrewnione ze sobą, a nawet procesy różnych użytkowników.
- Funkcja *open()* używana do otwierania kolejki FIFO do zapisu blokuje proces, aż do momentu otwarcia kolejki do odczytu, i odwrotnie. Dopiero gdy kolejka otwarta jest jednocześnie do zapisu i odczytu obie funkcje *open()* powracają ze stanu uśpienia;
- „Sprzątanie” tymczasowych kolejek może wymagać usunięcia ich z systemu plików funkcją *unlink()*.
- Kolejki FIFO mają z reguły większy rozmiar bufora (4–16 kB)

Podobieństwa:

- Zamknięcie końca używanego do pisania (dokładniej: wszystkich końców) generuje u czytelników EOF;
- Zamknięcie końca używanego do czytania generuje sygnał SIGPIPE wysyłany do wszystkich piszących do strumienia;
- Zapis i odczyt przez standardowe funkcje I/O, takie jak *write()* czy *read()*;
- Niepodzielność zapisów mniejszych niż rozmiar strumienia;
- Blokowanie procesów w przypadku przepełnienia lub pustego strumienia (i możliwość zastosowania *O_NDELAY*).

Korzystanie ze strumieni FIFO

```
#include <stdio.h>

main()
{
    int fd;
    int err;

    err=mknod("/tmp/fifo1", S_IFIFO | 0660, 0)
    if (err<0 && err!=EEXIST) {
        fprintf(stderr, "Nie mogę utworzyć FIFO\n");
        exit (1);
    }

    fd=open("/tmp/fifo1, O_RDONLY); /* blokada? */
    if (fd<0) {
        fprintf(stderr, "Nie mogę otworzyć FIFO do czytania\n");
        exit(2);
    }

    read(fd, ... ..);

    ...

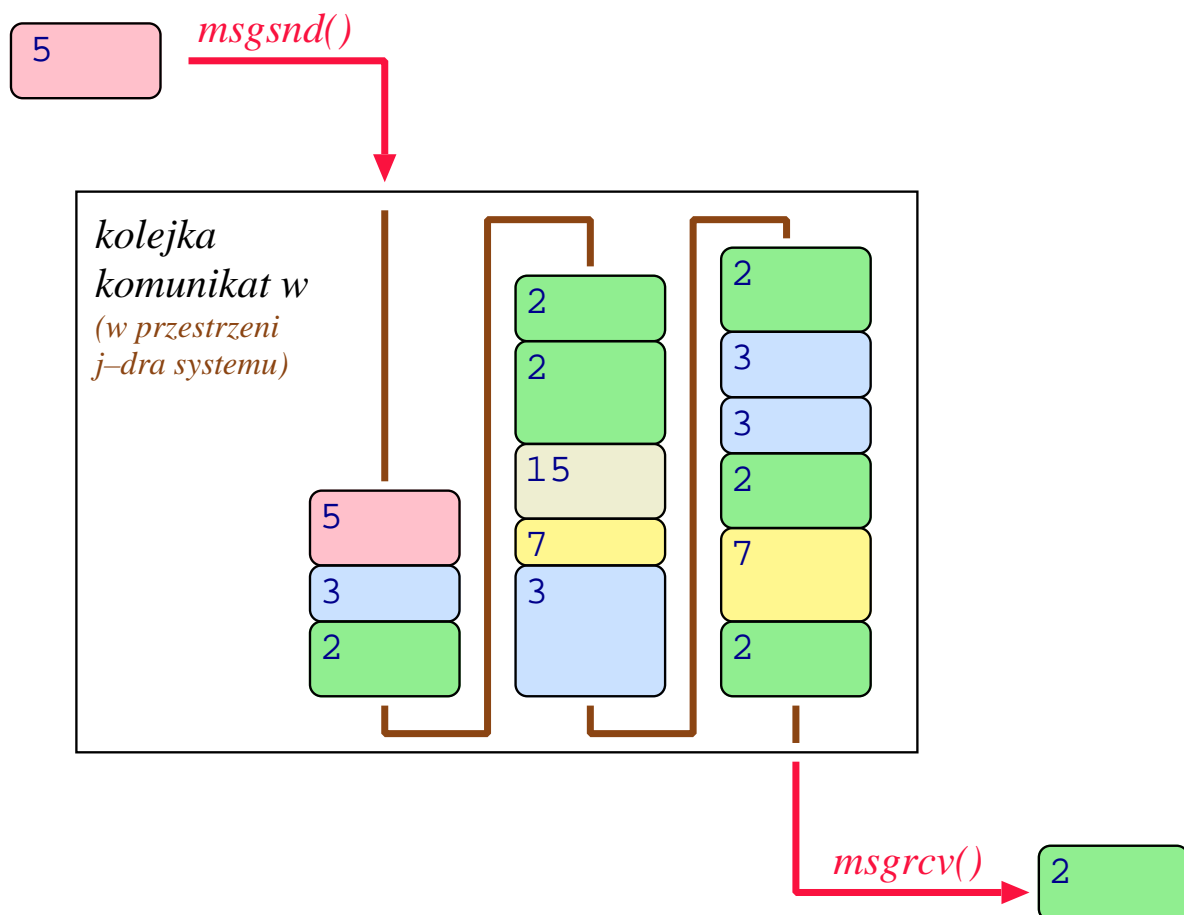
    close(fd);
    unlink("/tmp/fifo1");
}
```

Kolejki komunikatów

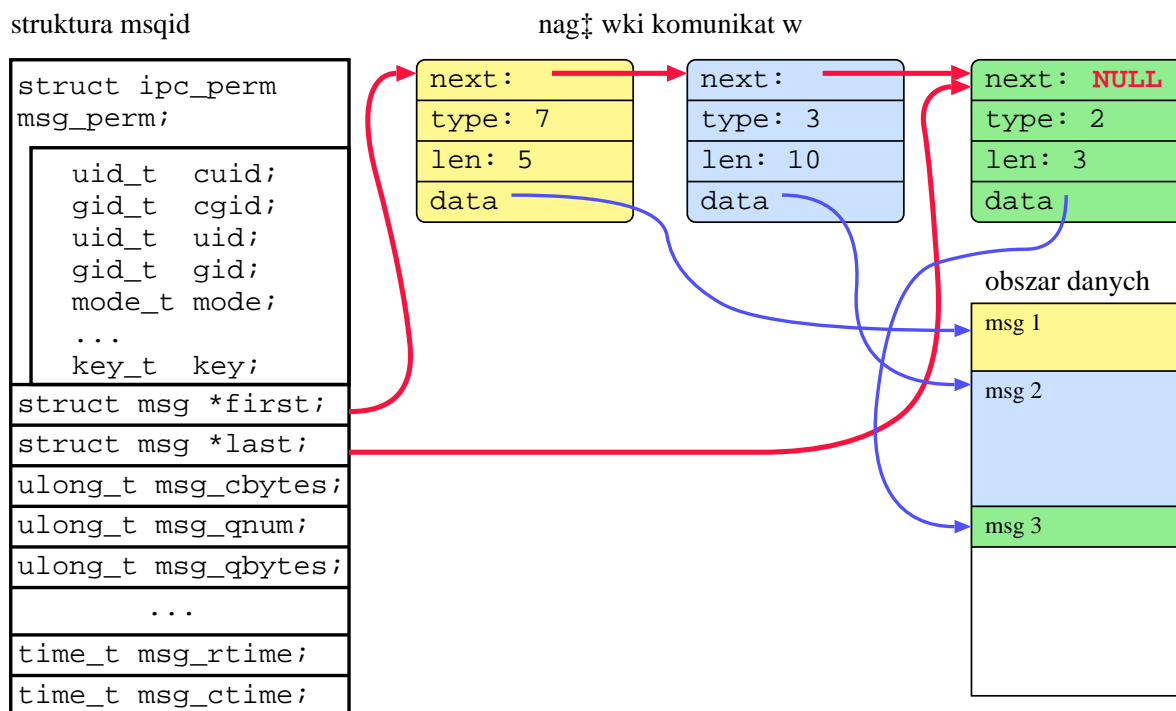
- lista (jednokierunkowa) zawierająca komunikaty o określonym maksymalnym rozmiarze;
- nowe komunikaty dodawane są na końcu listy, zachowując kolejność

ich wysyłania

- każdy komunikat ma dodatkowy parametr zwany *typem*, co pozwala na obsługę kilku „strumieni” komunikatów w ramach jednej kolejki (poprzez selektywne odbieranie komunikatów wybranego typu).



Struktura kolejki w jądrze systemu



- `cuid` i `cgid` – proces, który utworzył kolejkę (*creator uid*, *creator gid*);
- `uid` i `gid` – aktualny właściciel i właściciel grupowy kolejki;
- `mode` – prawa dostępu do kolejki;
- `key` – klucz identyfikujący kolejkę;
- `first`, `last` – wskaźniki na pierwszy i ostatni komunikat w kolejce;
- wielkość całej kolejki nie może przekraczać `msg_qbytes`, a aktualna wielkość to `msg_cbytes`;
- aktualna liczba komunikatów w kolejce to `msg_qnum`.

Tworzenie kolejek

```
int msgget(key_t key, int msgflg);
```

Funkcja *msgget()* tworzy nową kolejkę lub znajduje już istniejącą

- Jeśli klucz ma wartość `IPC_PRIVATE`, zawsze jest tworzona nowa kolejka.
- Jeśli kolejka o podanym kluczu już istnieje, zostaje zwrócony jej identyfikator, o ile użytkownik ma odpowiednie prawa dostępu, w przeciwnym razie zwracany jest (poprzez zmienną `errno`) błąd `ENOENT`;
- Jeśli kolejka nie istnieje, a wśród opcji `msgflg` ustawiona była `IPC_CREAT`, zostaje utworzona nowa kolejka i zwrócony jej identyfikator;
- Użycie opcji `IPC_EXCL` wymusza utworzenie nowej kolejki – jeżeli były ustawione obie opcje: `IPC_CREAT` i `IPC_EXCL`, a kolejka już istniała, zostaje zwrócony błąd `EEXIST`.

Jeśli tworzona jest nowa kolejka, najmłodsze 9 bitów w opcjach oznacza prawa dostępu do tworzonej kolejki, np. 0644 albo 0666.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define KEY ((key_t) 12345L)
#define PERM 0600

main()
{
    int msqid;

    if ( (msqid=msgget(KEY, PERM | IPC_CREAT)) < 0) {
        fprintf(stderr, "Nie można utworzyć kolejki\n");
        exit (1);
    }
    ...
}
```

Usuwanie kolejek

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Funkcja *msgctl()* pozwala usunąć kolejkę, ale także sprawdzić jej stan lub zmienić pewne wartości związane z kolejką. Sposób działania zależy od parametru *cmd*:

- **IPC_STAT** – umieszcza dane o kolejce w strukturze wskazywanej przez *buf*;
- **IPC_SET** – zmienia parametry kolejki na parametry przekazane w strukturze wskazywanej przez *buf*. Zmienione mogą zostać: właściciel i grupa (o ile wołający proces ma *uid=0* lub taki sam, jak proces, który utworzył kolejkę), prawa dostępu do kolejki i maksymalny rozmiar kolejki;
- **IPC_RMID** – usuwa kolejkę z systemu.

Polecenia systemowe związane z kolejkami komunikatów:

- **ipcrm** – usuwa kolejkę
- **ipcs -q** – sprawdza status kolejki

Wysyłanie komunikatów

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,
           int msgflg);

struct mymsg {           /* przykładowa struktura komunikatu */
    long  mtype;          /* typ komunikatu */
    char  mtext[1];       /* zawartość komunikatu */
}
```

- Wysyłając komunikat należy określić jego długość i adres początku;
- `msgflg` może mieć ustawiony bit `IPC_NOWAIT`, oznaczający, że funkcja nie będzie blokowana, jeśli komunikatu nie można wysłać natychmiast;
- Wysyłany komunikat musi mieć określony typ – liczba dodatnia na samym początku struktury przechowującej komunikat;
- Struktura zawierająca komunikat może być dowolna, ale pierwsze pole musi być liczbą typu `long`, zawierającą typ komunikatu.

```
struct komunikat {
    long type;
    int x;
    int y;
    char opis[30];
}

main()
{
    int mq;
    struct komunikat msg={12, 1, 2, "abc"};

    mq=msgget(123, IPC_CREAT | 0644);
    msgsnd(mq, &msg, sizeof(msg), 0);
}
```

Odbieranie komunikatów

```
int msgrcv(int msqid, void *msgp, size_t msgsz,  
           long msgtyp, int msgflg);
```

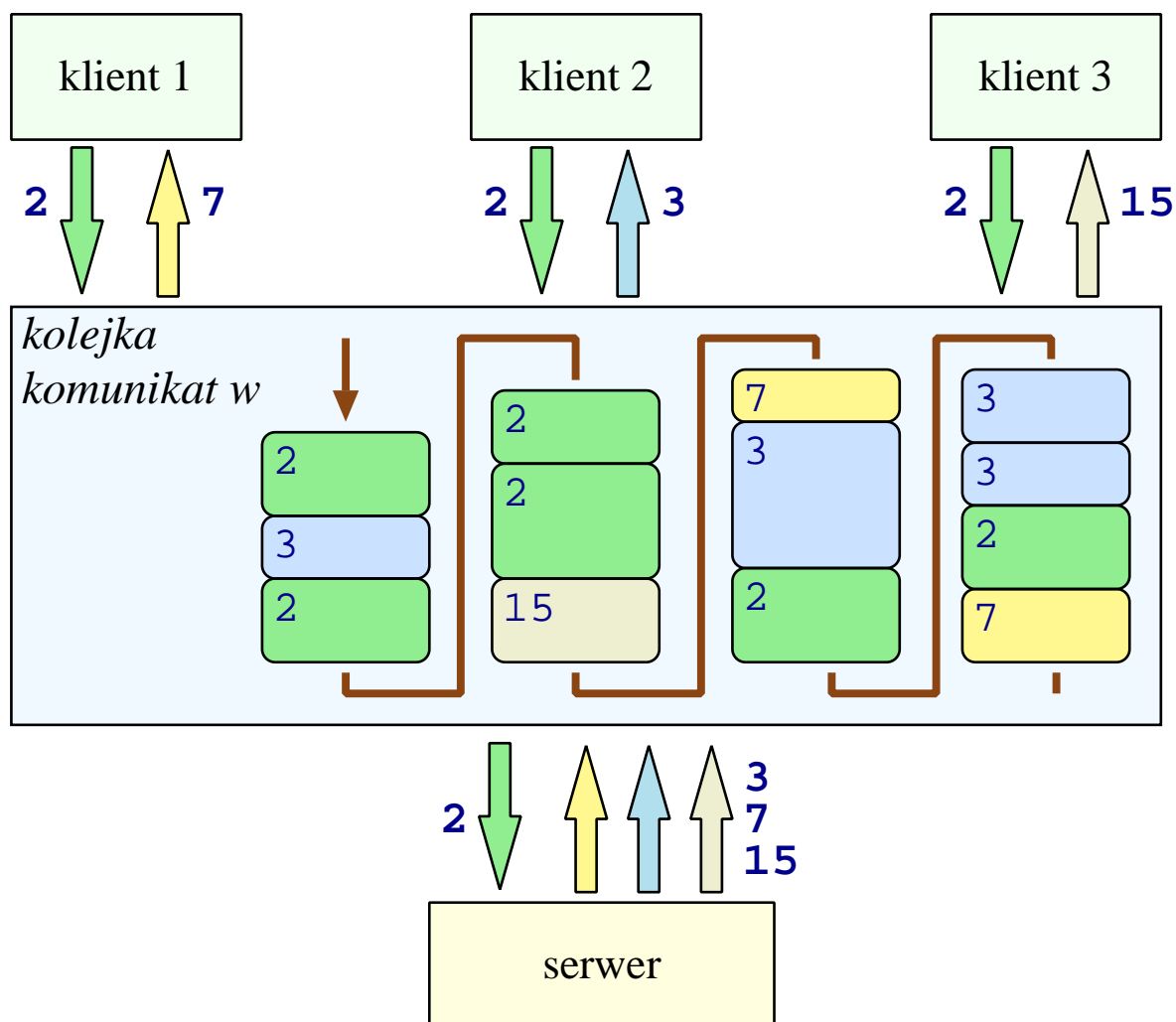
Wartość `msgtyp` określa jakie komunikaty mogą zostać odebrane:

- 0 – pierwszy komunikat z kolejki, dowolnego typu;
- > 0 – pierwszy komunikat o podanym typie;
- < 0 – spośród komunikatów, których typ jest mniejszy lub równy wartości bezwzględnej `msgtyp`, wybierany jest komunikat o najniższym typie (a jeśli jest ich kilka – pierwszy z nich).

Wartość `msgflg` określa sposób zachowania funkcji:

- jeśli ustawiony jest znacznik `IPC_NOWAIT`, a w kolejce nie mażądanego komunikatu, funkcja wróci natychmiast, zwracając wartość -1 i ustawiając zmienną `errno` na `ENOMSG`;
- jeśli znacznik `IPC_NOWAIT` nie jest ustawiony, a w kolejce nie mażądanego komunikatu, funkcja powoduje uśpienie procesu, do czasu gdy:
 - zostanie wysłany odpowiedni komunikat;
 - kolejka zostanie usunięta z systemu (błąd `EIDRM`);
 - zostanie odebrany sygnał, który nie jest ignorowany. Komunikat nie jest pobierany z kolejki, a funkcja `msgrcv()` zwraca błąd `EINTR` lub jest automatycznie restartowana – zależnie od opcji ustawionych przez `sigaction()` (parametr `SA_RESTART`).
- jeśli ustawiony jest znacznik `MSG_NOERROR`, funkcja nie będzie sygnalizować błędu, jeśli odbierany komunikat jest większy niż długość przekazanego bufora, przycinając go do żądanego rozmiaru, w przeciwnym razie komunikat nie zostanie odebrany, a zmienna `errno` zostanie ustawiona na `E2BIG`.

Selektywne przesyłanie komunikatów



- klienci przesyłają komunikaty do serwera oznaczając je typem 2 (lub np. 1 dla danych o wyższym priorytecie, 2 – normalnym);
- serwer odbiera wyłącznie komunikaty o typie 2 (lub 1–2), a wysyła komunikaty innych typów;
- każdy klient ma osobny numer dla komunikatów, które odbiera (np. swój numer procesu).

Funkcja *ftok()*

```
#include <sys/ipc.h>
```

```
key_t ftok(const char *path, int id);
```

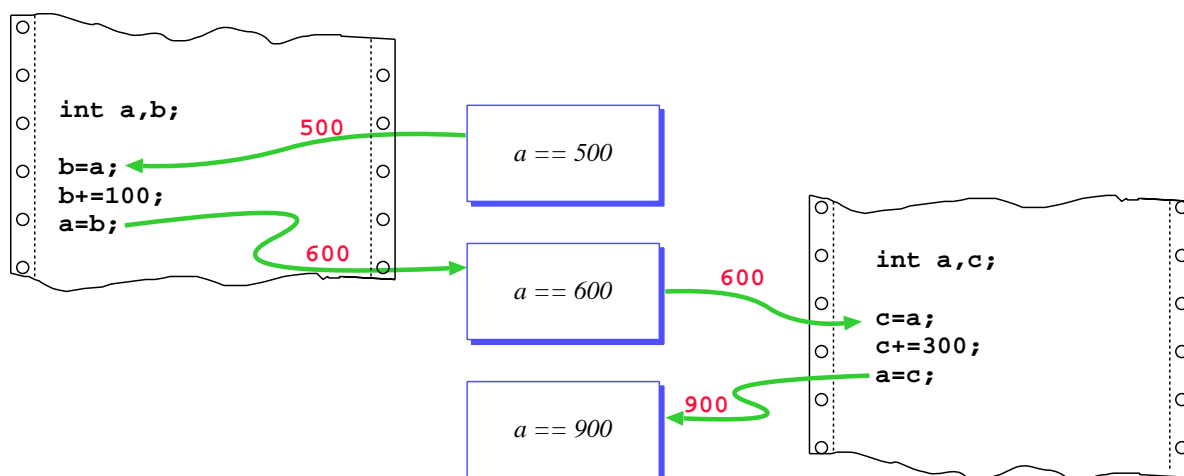
Funkcja *ftok()* zwraca identyfikator, którego można użyć w celu utworzenia lub dostępu do istniejącej kolejki komunikatów, semafora lub segmentu pamięci wspólnej.

- Użycie w dwóch wywołaniach *ftok()* takich samych parametrów *path* i *id* powoduje utworzenie identycznych kluczy;
- Użycie różnych się parametrów powoduje wygenerowanie różnych (unikalnych) kluczy.
- Klucz tworzony jest na podstawie numeru i-węzła podanego pliku, więc plik musi istnieć, w przeciwnym razie *ftok()* zwraca wartość -1.
- Numery i-węzłów są unikalne tylko w obrębie jednego systemu plików, dlatego w celu zapewnienia unikalności klucza, używany jest także „mały numer” urządzenia (*device minor number*), na którym znajduje się plik (odpowiada to numerowi partycji dysku);

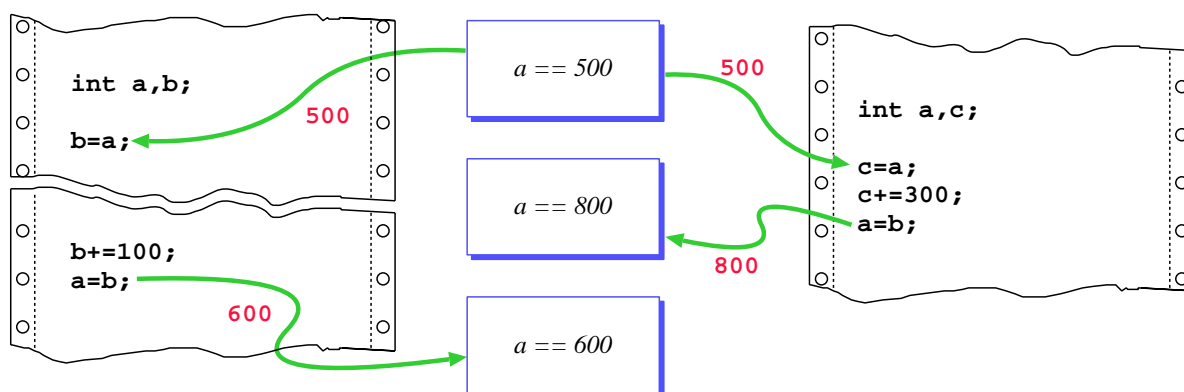
Semafor

Sekcja krytyczna

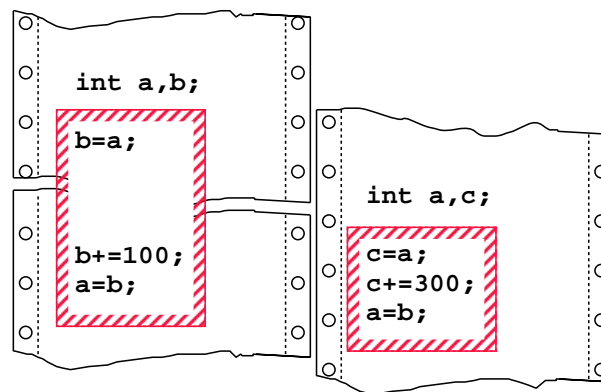
Dwa procesy niezależnie modyfikują wspólną zmienną „a”, jeden po drugim:



Dwa procesy niezależnie modyfikują wspólną zmienną „a”, lecz system operacyjny powoduje chwilowe wstrzymanie pierwszego procesu w trakcie operacji:



- Od momentu pobrania wartości zmiennej „a” do momentu jej odeśnięcia żaden inny proces nie powinien mieć do niej dostępu
- Fragment programu potrzebujący wyłączności działania nazwiemy *sekcją krytyczną*.



Rozwiązanie?

```
static int sem=1;
```

```
Zajmij(int *wolny)
{
    while (*wolny<=0)
        sleep(1);
    *wolny--;
}
```

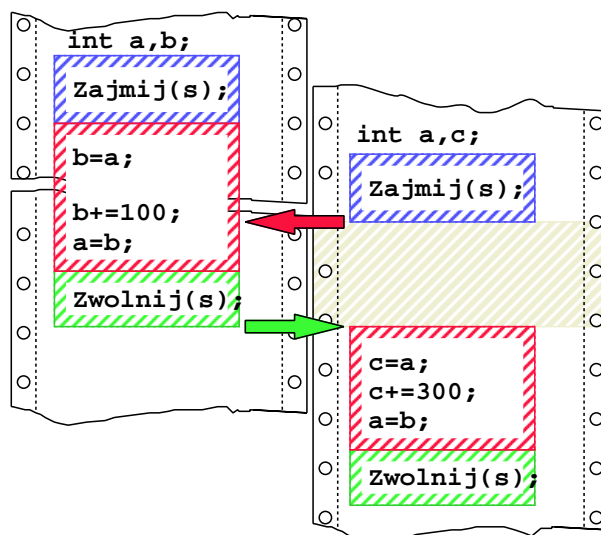
```
Zwolnij(int *wolny)
{
    *wolny++;
}
```

```
main()
{
    int a, b;

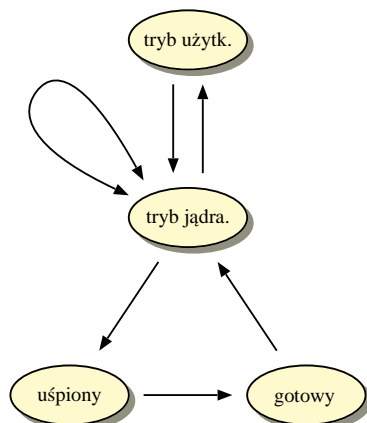
    Zajmij(&sem);
    b=a;
    b+=100;
    a=b;
    Zwolnij(&sem);
}
```

Dlaczego
to nie
działa ?

To będzie działać, jeśli operacje *Zajmij()* i *Zwolnij()* będą niepodzielne.



- Operacja *Zajmij()* jest *wejściem* do sekcji krytycznej – usypia proces, jeśli zasób już jest zajęty, aż do momentu, gdy inny proces go zwolni. Proces zostaje umieszczony w kolejce procesów oczekujących na zwolnienie zasobu (semafora);
- Operacja *Zwolnij()* stanowi *wyjście* z sekcji krytycznej – nigdy nie usypia procesu, za to może spowodować obudzenie innego (przeniesienie go z kolejki oczekujących).
- Obie operacje są niepodzielne (atomowe) i działają na poziomie jądra systemu.



Semafor możemy traktować jak abstrakcyjny typ danych składający się ze zmiennej i kolejki procesów:

- *Zajmij*(S)

Jeśli $S > 0$, to $S \leftarrow S - 1$ i idź dalej.

Jeśli $S = 0$, to umieść proces w kolejce oczekujących na zwolnienie semafora i zaśnij.

- *Zwolnij*(S)

Jeśli kolejka oczekujących jest pusta, to $S \leftarrow S + 1$:

Jeśli ktoś czeka, to usuń go z kolejki oczekujących i obudź go, nie zmieniając wartości S .

Semafor binarny – przyjmujący jedynie wartości 0 i 1.

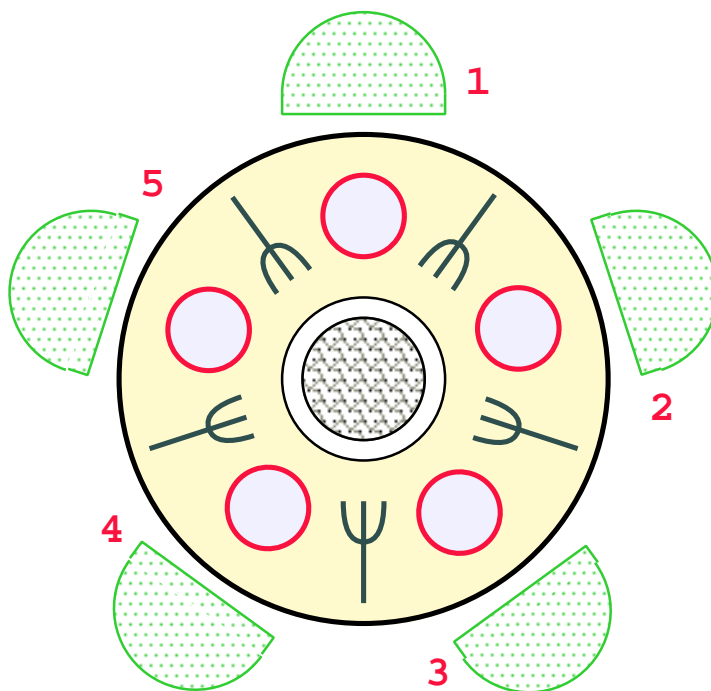
Semafor ogólny – przyjmujące dowolne (lub wybrane) wartości nieujemne

- Bieżąca wartość semafora odpowiada liczbie „wolnych” zasobów;
- Osiągnięcie wartości 0 oznacza, że kolejny proces żądający zasobu zostanie uśpiony;
- Możliwe jest zmniejszenie lub zwiększenie wartości semafora o wartość inną niż 1 (o ile zmniejszenie nie doprowadziłoby do osiągnięcia wartości ujemnej).

Inne przykłady wymagające ochrony dostępu do sekcji krytycznej:

- modyfikacja wspólnej zmiennej w pamięci lub pliku (konto bankowe?);
- zapis pliku przez kilka procesów;
- modyfikacja rekordu w bazie danych;
- dostęp do urządzeń zewnętrznych (terminale, modemy, itp.);

Problem pięciu filozofów



- Życie filozofa to medytacje, ale to wyczerpujące zajęcie, więc filozof czasami robi się głodny;
- Głodny filozof udaje się do jadalni i zajmuje miejsce przy stole;
- Filozof je tylko wtedy, gdy ma dwa widelce;
- Dwóch filozofów nie może jednocześnie trzymać tego samego widelca;
- Najedzony filozof wychodzi z jadalni i wraca medytować.

Problemy:

- Żaden filozof nie powinien zostać zagłodzony.
- Nie można dopuścić do blokady.
- Problem higieny pomijamy, jako nie wnoszący nic do rozwiązania.

Tworzenie semaforów

```
int semget(key_t key, int semflg);
```

Funkcja *semget()* tworzy nowy zbiór semaforów lub znajduje już istniejący:

- Jeśli klucz ma wartość `IPC_PRIVATE`, zawsze jest tworzony nowy zbiór semaforów;
- Jeśli semafor o podanym kluczu już istnieje, zostaje zwrócony jego identyfikator, o ile użytkownik ma odpowiednie prawa dostępu do niego, w przeciwnym razie zwracany jest (poprzez zmienną `errno`) błąd `ENOENT`;
- Jeśli semafor nie istnieje, a wśród opcji `semflg` ustawiona była `IPC_CREAT`, zostaje utworzony nowy semafor i zwrócony jego identyfikator;
- Użycie opcji `IPC_EXCL` wymusza utworzenie nowego semafora – jeżeli były ustawione obie opcje: `IPC_CREAT` i `IPC_EXCL`, a semafor już istniał, zostaje zwrócony błąd `EEXIST`.

Jeśli tworzony jest nowy semafor, najmłodsze 9 bitów w opcjach oznacza prawa dostępu do tworzonego semafora, np. 0644 albo 0666.

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

#define KEY ((key_t) 12345L)
#define PERM 0600

main()
{
    int semid;
    if ( (semid=semget(KEY, PERM | IPC_CREAT)) < 0) {
        fprintf(stderr, "Nie można utworzyć semafora\n");
        exit (1);
    }
    ...
}
```

Usuwanie semaforów

```
int semctl(int semid, int semnum, int cmd, union semun arg);

union semun {
    int          val;
    struct semid_ds *buf;
    ushort_t     *array;
} arg ;
```

Funkcja *semctl()* pozwala usunąć semafor, ale także wykonać na nim różne operacje. Sposób działania zależy od parametru *cmd*, a parametr *semnum* określa numer semafora w grupie, którego dotyczy operacja (przy pierwszych 3 operacjach, dotyczących całej grupy, ten parametr jest nieistotny):

- **IPC_STAT** – umieszcza dane o semaforze (takie jak właściciel, prawa dostępu, itp.) w strukturze wskazywanej przez *buf*;
- **IPC_SET** – zmienia parametry grupy semaforów na parametry przekazane w strukturze wskazywanej przez *buf*. Zmienione mogą zostać: właściciel i grupa (o ile wołający proces ma *uid=0* lub taki sam, jak proces, który utworzył semafor) i prawa dostępu do semafora.
- **IPC_RMID** – usuwa grupę semaforów z systemu.
- **GETVAL** – pobiera i zwraca wartość semafora *semnum*;
- **SETVAL** – ustawia wartość semafora *semnum* na wartość *val* unii *semun* przekazanej jako argument *arg*;

Polecenia systemowe związane z semaforami

- **ipcrm** – usuwa semafor (grupę semaforów) z systemu;
- **ipcs -s** – sprawdza status semafora.

Zajmowanie i zwalnianie semaforów

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

```
struct sembuf {  
    short  sem_num;    /* numer semafora */  
    short  sem_op;     /* operacja na semaforze */  
    short  sem_flg;    /* opcje */  
}
```

Ogólnie:

- `sem_op < 0` – zajęcie semafora
- `sem_op > 0` – zwolnienie semafora
- `sem_op == 0` – synchronizacja z semaforem (oczekiwanie, aż jego wartość osiągnie 0)

Opcje `sem_flg`:

- `IPC_NOWAIT`

Jeżeli nie można wykonać żądanej operacji, proces nie zostaje uśpiony, lecz funkcja wraca z błędem `EAGAIN`

- `SEM_UNDO`

Jeżeli semafor zostaje przydzielony, liczba, o jaką została zmniejszona jego wartość, zostaje dodana do zmiennej `semadj` procesu (a przy zwalnianiu semafora – odjęta od tej zmiennej).

Jeśli proces zostaje zakończony z niezerową wartością `semadj` (np. po otrzymaniu sygnału itp.), funkcja `exit()` spowoduje dodanie tej zmiennej do wartości semafora (czyli zwolnienie zajętych zasobów).

Zajęcie semafora

```
struct sembuf op_zajm={
    0,    /* semafor nr 0 */
    -1,   /* zajmij == odejmij wartość N (tutaj: 1) */
    0     /* opcje */
};
int semid;

semid=semget(IPC_PRIVATE, 1, IPC_CREAT);
if (semid== -1)
    ... (błąd) ...
semop(semid, &op_zajm, 1);
/* sekcja krytyczna */
```

Algorytm zajmowania semafora ($N < 0$):

- jeśli wartość semafora jest większa lub równa wartości bezwzględnej N , to zmniejsz ją o tyle (a jeśli użyto opcji `SEM_UNDO`, to dodatkowo zwiększ o $|N|$ zmienną `semadj`).
- jeśli nie, i nie została użyta opcja `IPC_NOWAIT`, zwiększ licznik procesów oczekujących na semafor i zawieś proces, aż do czasu, gdy nastąpi jedno ze zdarzeń:
 - zostanie spełniony ten warunek. Wówczas wartość semafora jest pomniejszana o $|N|$ (i ew. zwiększana o $|N|$ wartość `semadj`), zmniejszany jest licznik procesów oczekujących, a proces zostaje obudzony.
 - semafor zostanie usunięty z systemu;
 - proces otrzyma sygnał (funkcja zwróci błąd `EINTR` lub zostanie automatycznie wznowiona).
- jeśli nie było możliwe zajęcie semafora, ale użyto opcji `IPC_NOWAIT`, wróć natychmiast z błędem `EAGAIN`.

Zwolnienie semafora

```
struct sembuf op_tab[2]={
    0,    /* semafor nr 0 */
    -1,   /* zajmij == odejmij wartość N (tutaj: 1) */
    0     /* opcje */
}, {
    0,    /* semafor nr 0 */
    1,    /* zwolnij == dodaj wartość N (tutaj: 1) */
    0     /* opcje */
};
int semid;

semid=semget(IPC_PRIVATE, 1, IPC_CREAT);
if (semid==-1)
    ... (błąd) ...
semop(semid, op_tab, 1);
    /* sekcja krytyczna */
    /* ... */
semop(semid, &(op_tab[1]), 1);
```

Algorytm zwalniania semafora przez system operacyjny ($N > 0$):

- wartość semafora zostaje zwiększona o N ,
- jeśli ustawiona jest opcja `SEM_UNDO`, wartość ta jest również odejmowana od zmiennej `semadj` procesu związanej z tym semaforem,
- jeśli licznik procesów oczekujących na semafor ma wartość większą od zera, zostaje obudzony pierwszy proces z kolejki.

Oczekiwanie na zajęcie semafora przez inny proces

Jeśli wartość N jest równa 0, proces oczekuje aż semafor zostanie zajęty przez inny proces:

- jeśli wartość semafora wynosi 0, funkcja *semop()* wraca natychmiast;
- jeśli wartość semafora jest większa od 0, ale użyto opcji *IPC_NOWAIT*, funkcja natychmiast wraca z błędem *EAGAIN*;
- jeśli wartość semafora jest większa od 0 i nie użyto opcji *IPC_NOWAIT*, zostaje zwiększona wartość *semzcnt* semafora, a proces zostaje uśpiony, do czasu, gdy:
 - wartość semafora (*semval*) osiągnie zero;
 - semafor zostanie usunięty z systemu;
 - proces otrzyma sygnał, który ma zostać przechwycony. Wówczas wartość *semzcnt* jest zmniejszana i wołana procedura obsługi sygnału zarejestrowana wcześniej wywołaniem funkcji *signal()*.

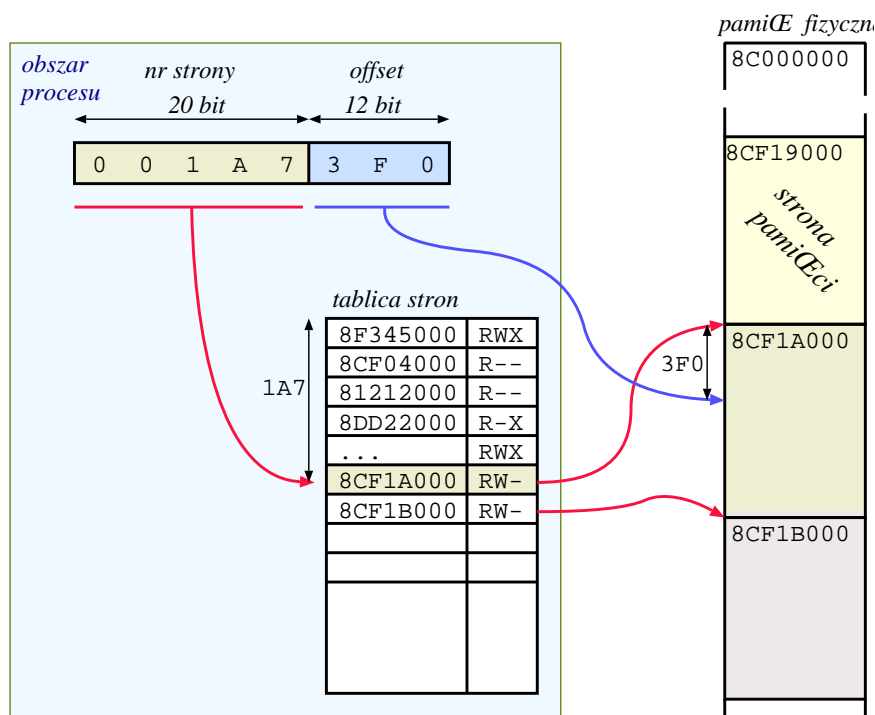
Zastosowania dla tego typu postępowania:

- Rozwiązanie problemu wyścigu podczas inicjowania wartości semafora.
- Synchronizacja z procesem żądającym dostępu do zasobu (jednoczesna operacja typu „czekaj, aż osiągnie 0, a gdy się to stanie, zwolnij”).

Pamięć wspólna

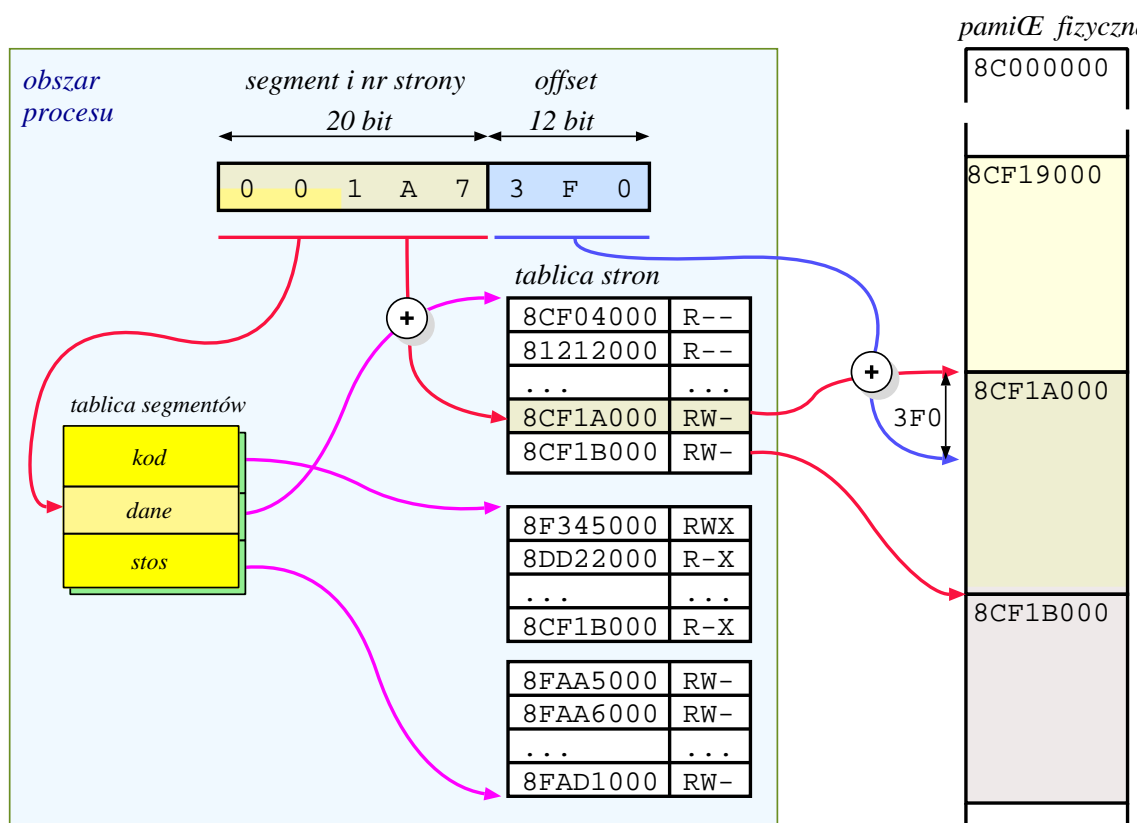
Stronicowanie pamięci

- Adresy wkompileowane w programy nie mogą być adresami pamięci fizycznej, bo nie byłoby możliwe współbieżne wykonywanie kilku procesów.
- Kompilatory generują kod operujący w virtualnej przestrzeni adresowej (ograniczonej z góry, np. do 4GB), a tłumaczenia adresów wirtualnych na rzeczywiste dokonuje jednostka MMU procesora.
- System może wykonywać kilka kopii tego samego programu, operującego tymi samymi adresami pamięci wirtualnej, więc tłumaczenie adresów wirtualnych na fizyczne jest ściśle związane z kontekstem procesu.
- Pamięć wirtualna uzyskiwana jest przez *segmentację* i *stronicowanie*. Jednostkowym obszarem pamięci jest *strona*, zwykle wielkości od 8 do 32 kB (stały rozmiar zależy od systemu).



Segmentacja pamięci

W rzeczywistości dostęp do pamięci jest trochę bardziej skomplikowany. Dostęp do pamięci fizycznej odbywa się w pierwszym rzędzie poprzez segmenty (danych, kodu, stosu), a dopiero potem – strony:



- Każdy segment posiada własną tablicę stron (adres tablicy i jej długość);
- segment reprezentuje ciągły obszar pamięci;
- przy każdym dostępie do pamięci starsza część adresu wyznacza używany segment, a jednocześnie jest porównywana z jego wielkością, w celu wykrycia dostępu poza przydzielony segment;
- starsza część adresu wyznacza indeks w tablicy stron segmentu i pozwala odczytać adres strony pamięci fizycznej;
- ostateczny adres powstaje przez zsumowanie adresu strony i offsetu.

Tworzenie segmentu pamięci wspólnej

```
int shmget(key_t key, int size, int shmflg);
```

Funkcja *shmget()* tworzy nowy segment pamięci wspólnej lub znajduje już istniejący:

- Jeśli klucz ma wartość `IPC_PRIVATE`, tworzony jest nowy segment;
- Jeśli segment o podanym kluczu już istnieje, zostaje zwrócony jego identyfikator, o ile użytkownik ma odpowiednie prawa dostępu do niego, w przeciwnym razie zwracany jest (poprzez zmienną `errno`) błąd `ENOENT`;
- Jeśli segment pamięci wspólnej nie istnieje, a wśród opcji `shmflg` ustawiona była `IPC_CREAT`, zostaje utworzony nowy blok pamięci i zwrócony jego identyfikator;
- Użycie opcji `IPC_EXCL` wymusza utworzenie nowego segmentu pamięci – jeżeli były ustawione obie opcje: `IPC_CREAT` i `IPC_EXCL`, a segment już istniał, zostaje zwrócony błąd `EEXIST`.

Jeśli tworzony jest nowy segment pamięci, najmłodsze 9 bitów w opcjach oznacza prawa dostępu do tworzonego bloku pamięci, np. 0644 albo 0666.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define KEY ((key_t) 12345L)
#define PERM 0600
#define SIZE 4096

main()
{
    int shmid;

    if ( (shmid=shmget(KEY, SIZE, PERM | IPC_CREAT)) < 0 ) {
        fprintf(stderr, "Nie można utworzyć bloku pamięci wspólnej\n");
        exit (1);
    }
    ...
}
```

Usuwanie pamięci wspólnej

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Funkcja *shmctl()* pozwala usunąć segment pamięci wspólnej, ale także wykonać na nim różne operacje. Sposób działania zależy od parametru *cmd*:

- **IPC_STAT** – umieszcza dane o bloku pamięci (takie jak właściciel, prawa dostępu, itp.) w strukturze wskazywanej przez *buf*;
- **IPC_SET** – zmienia parametry segmentu pamięci wspólnej na parametry przekazane w strukturze wskazywanej przez *buf*. Zmienione mogą zostać: właściciel i grupa (o ile wołający proces ma *uid==0* lub taki sam, jak proces, który utworzył segment pamięci wspólnej) i prawa dostępu do segmentu.
- **IPC_RMID** – usuwa segment pamięci wspólnej z systemu.
- **SHM_LOCK** – zablokowanie w pamięci segmentu pamięci wspólnej (operacja może być wykonana wyłącznie przez użytkownika posiadającego *euid==0*);
- **SHM_UNLOCK** – odblokowanie segmentu.

Polecenia systemowe związane z pamięcią wspólną:

- **ipcrm** – usuwa segment pamięci wspólnej z systemu;
- **ipcs -m** – sprawdza status pamięci wspólnej w systemie.

Korzystanie z pamięci wspólnej

Zanim proces będzie mógł odczytać coś z pamięci wspólnej lub do niej zapisać, musi zażądać od systemu dołączenia segmentu pamięci wspólnej do własnej przestrzeni adresowej (widząc jakby przez „okienko” swojej pamięci, segment pamięci wspólnej). Po zakończeniu korzystania z pamięci wspólnej, proces powinien odłączyć ją od swojej przestrzeni adresowej. Służą do tego odpowiednio funkcje *shmat()* i *shmdt()*.

```
void *shmat(int shmid, const void *shmaddr, int shmflg);  
int shmdt(const void *shmaddr);
```

Funkcja *shmat()* zwraca adres przyłączonego segmentu. Adres ten zależy od użytych parametrów *shmaddr* i *shmflg*:

- Jeśli *shmaddr*==NULL (czyli (void*)0), to przydzielany jest pierwszy wolny adres znaleziony przez system;
- Jeśli *shmaddr* jest niezerowy, wówczas system podłącza segment pamięci wspólnej pod podany adres, chyba że została użyta także opcja SHM_RND, wówczas adres zostaje wyrównany w dół, do najbliższego adresu podzielonego przez SHMLBA.
- Użycie opcji SHM_RDONLY powoduje podłączenie pamięci w trybie wyłącznie do odczytu, w przeciwnym razie dostęp jest do czytania i pisania.

Funkcja *shmdt()* odłącza od przestrzeni adresowej procesu segment pamięci wspólnej wskazywany przez *shmaddr*.

Zarządzanie segmentami pamięci przez system

Przydzielanie segmentu pamięci:

Nowy segment może zostać przydzielony programowi tylko w kilku wypadkach:

- Tworzenie nowego procesu za pomocą funkcji *fork()*
- Zwolnienie wszystkich segmentów procesu i przydział nowych w chwili wykonania *exec()*
- Przydział segmentu pamięci wspólnej za pomocą *shmget()*

Przyłączenie przydzielonych segmentów do procesu:

- Nowo tworzony proces: *fork()*, *exec()*
- Segment pamięci wspólnej: *shmdt()*

Zmiana rozmiaru segmentu:

- Segmenty prywatne: automatycznie, funkcją *sbrk()*
- Segmenty wspólne: nigdy

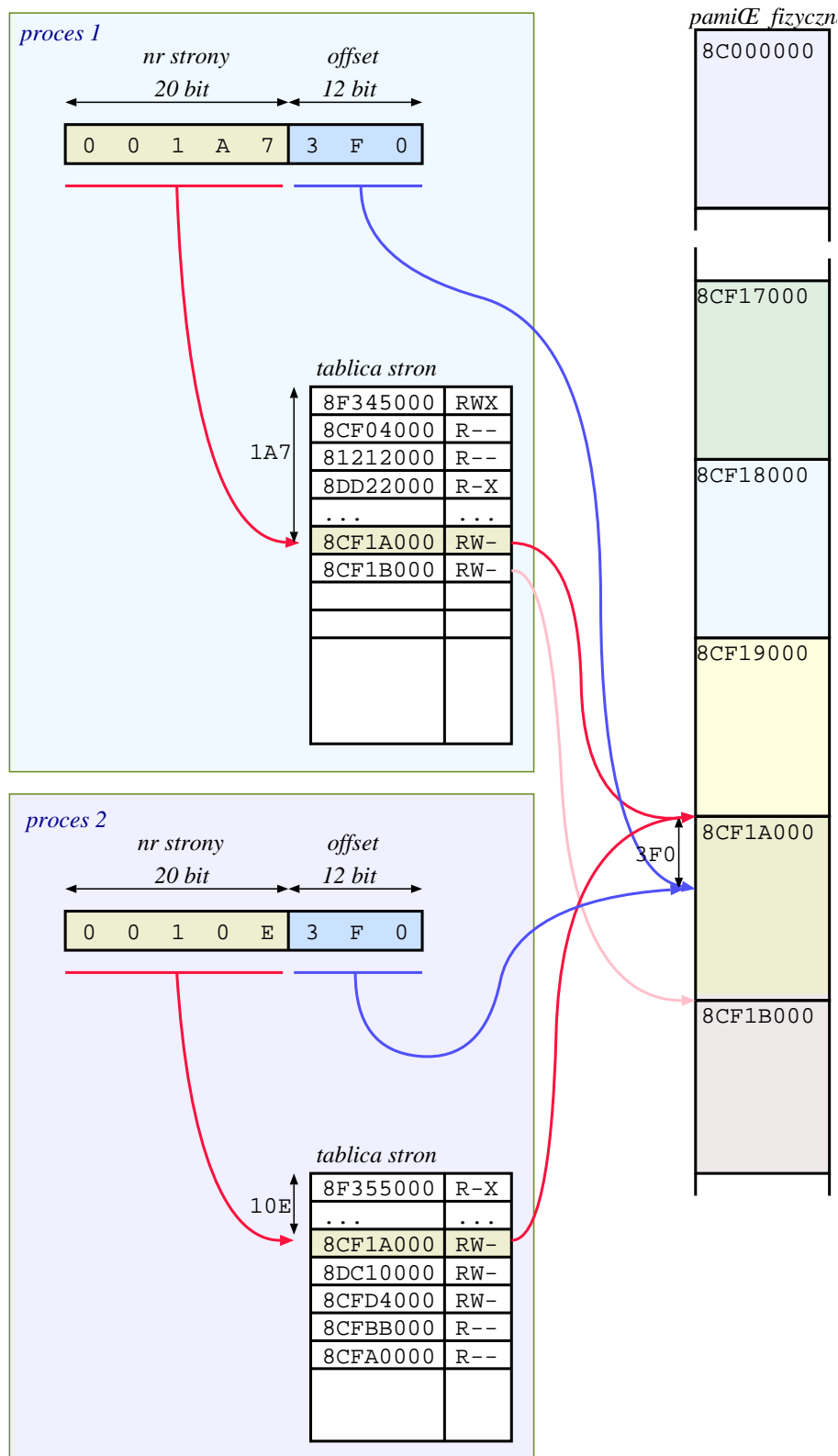
Odlączanie segmentów od procesu:

- Segmenty prywatne: *exit()* i *exec()*
- Segmenty wspólne: *shmdt()*

Zwalnianie segmentów:

- Segmenty prywatne: dokonywane automatycznie w chwili, gdy segment przestaje być dołączony do jakiegokolwiek procesu (o ile nie był ustawiony *sticky bit* – *rwxr-xr-t*).
- Segmenty wspólne: po wykonaniu *shmctl(IPC_RMID)*

Pamięć wspólna widziana przez 2 procesy



Wspólny dostęp do plików – *mmap()*

```
#include <sys/mmap.h>
```

```
void *mmap(void *addr, size_t len, int prot, int flags,  
           int fildes, off_t off);
```

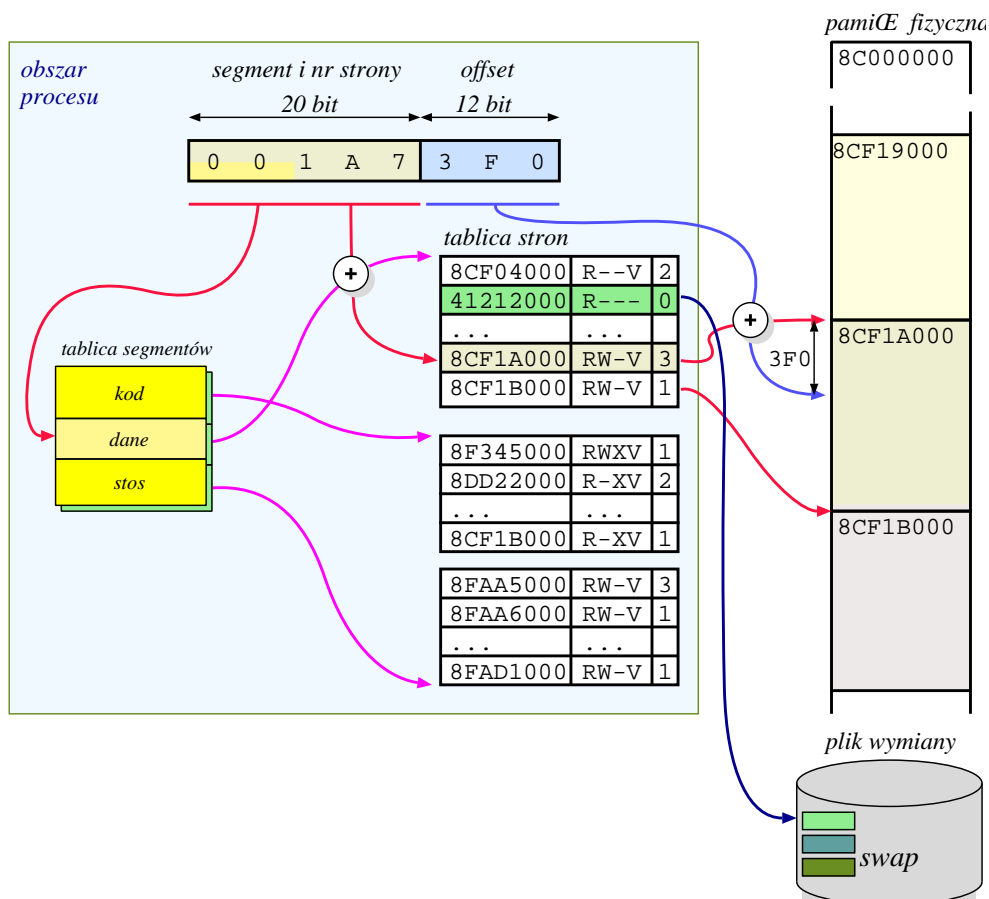
Funkcja *mmap()* pozwala na dostęp do otwartych plików tak, jakby znajdowały się w pamięci RAM – zamiast poruszać się po pliku za pomocą *fseek()* i pisać/czytać przez *fread()* i *fwrite()*, można używać wskaźników i zwykłych operacji przypisania.

- **addr** – preferowany adres w pamięci (jeśli NULL – system przydzieli adres nie ekolidujący z dotychczas przydzielonymi segmentami);
- **len** – wielkość obszaru;
- **prot** – jedna lub kilka opcji połączonych logicznym „or”: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE (w niektórych systemach część z kombinacji jest niedostępna, np. write+exec);
- **flags** – opcje dotyczące mapowanych danych, m.in.:
 - MAP_SHARED – zmiany są dokonywane wspólnie z innymi procesami;
 - MAP_PRIVATE – zmiany są dokonywane na prywatnej kopii danych (pierwsza próba zapisu powoduje utworzenie kopii);
- **fildes** – deskryptor otwartego uprzednio pliku (musi to być „zwykły” plik, nie może być gniazdko, FIFO, urządzenie itp.);
- **off** – przesunięcie mapowanego obszaru względem początku pliku.

Inne funkcje: *munmap()*, *msync()*.

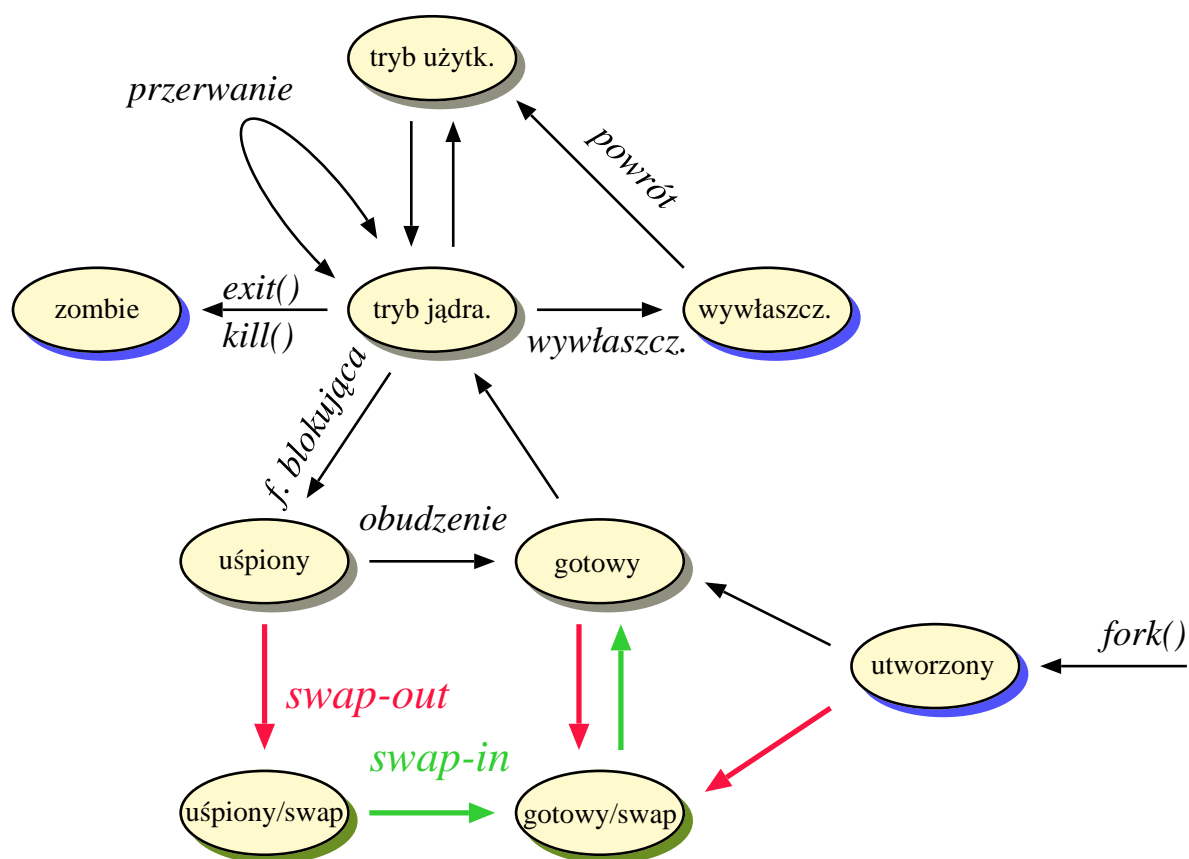
Pamięć wirtualna

W rzeczywistości dostęp do pamięci jest trochę bardziej skomplikowany. (*Deja Vu* – to takie dziwne uczucie, ...)



- Tablica stron może zawierać informacje o stronach, których *nie ma* w pamięci fizycznej;
- w chwili dostępu do takiej strony generowany jest wyjątek powodujący zachowanie kontekstu procesu, ew. przeniesienie do kolejki procesów oczekujących na sprowadzenie strony z pamięci swap, oraz sprowadzenie strony, po czym wznowienie procesu tak, jakby nic się nie stało;
- Dodatkowo tablica zawiera licznik odwołań, automatycznie zmniejszany, a służący do szukania stron, które mogą być wyrzucone do pamięci swap.

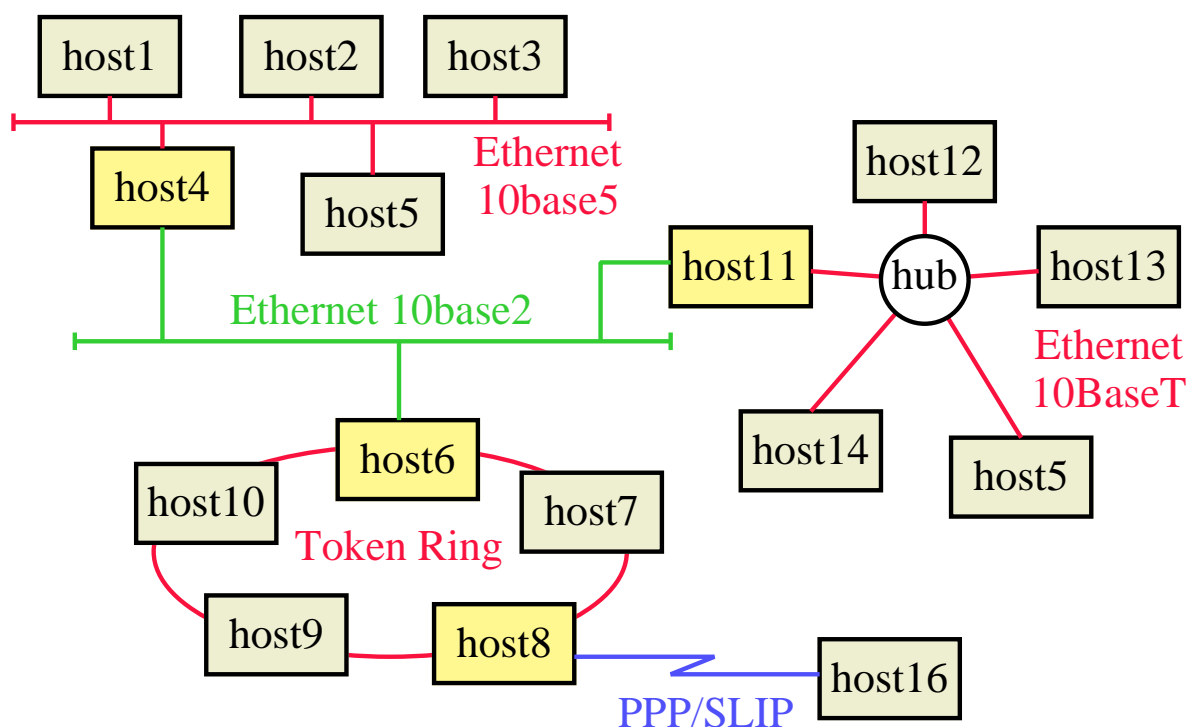
Pełny diagram stanów procesów



Komunikacja sieciowa

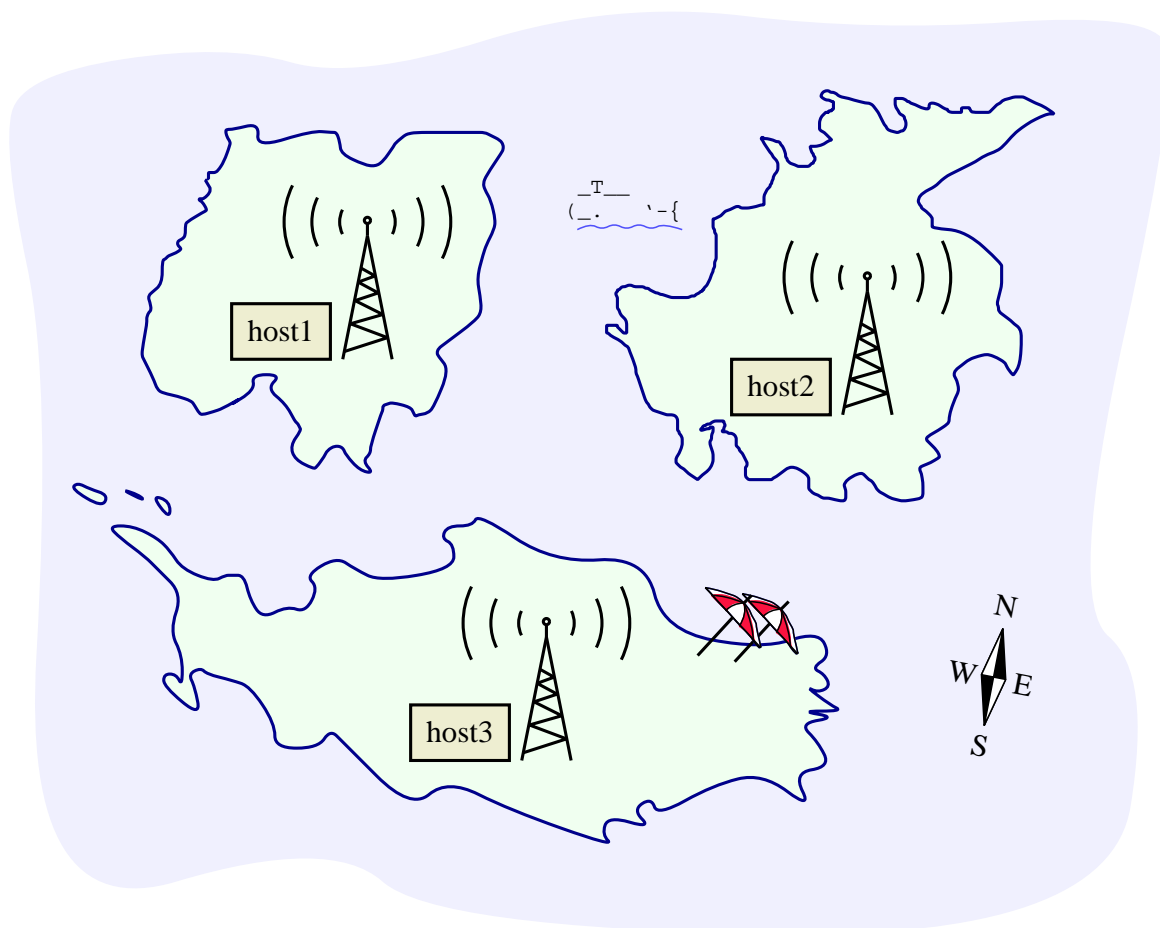
Różne rodzaje sieci:

- sieci broadcastowe
- sieci token ring i FDDI
- połączenia punkt-punkt (point-to-point)



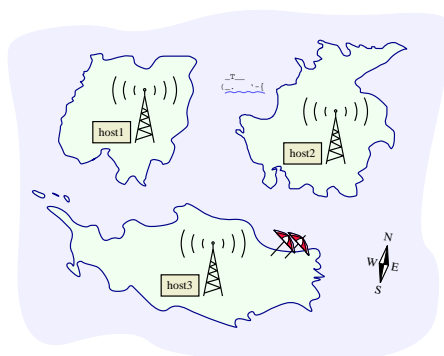
- nawet w obrębie sieci lokalnych stosowane są różne technologie, np. Ethernet 10baseT (skrętka), 10base5 (kabel koncentryczny 50Ω), 10base2 („gruby” ethernet)
- pojedyncze sieci lokalne łączone są ze sobą albo przez komputery z dwoma interfejsami sieciowymi, pełniącymi zwykle rolę routerów, albo specjalne urządzenia – huby, switchy, routery.
- sieć ethernet jest siecią typu broadcast, sieci komórkowe też;

- zaczęło się od „Aloha network” – pakietowej sieci radiowej na Hawajach w latach '70:



Dostarczanie pakietów:

- wszyscy słuchają *rozgłaszanych* pakietów, więc każdy pakiet musi mieć adres docelowy i adres nadawcy, np. unikalny numer przypisany komputerowi;
- jeśli pakiet do kogo innego – wyrzucić (ethernet – dokonywane na poziomie sprzętu, bez generowania niepotrzebnego przerwania);
- problem z poufnością – każdy może podsłuchiwać!



Dostęp do medium:

- wspólny zasób – co będzie, gdy 2 stacje zaczną nadawać jednocześnie? – trzeba wykrywać jakoś nałożenie się pakietów, czyli kolizje
- nadawaj w ciemno, ale dodaj sumę kontrolną
- suma kontrolna i tak się przyda – na wypadek nisko przelatujących samolotów
- jeśli dostałeś nieuszkodzony pakiet – odeślij potwierdzenie
- co będzie, jeśli potwierdzenie nie dotrze?
- nadawca czeka jakiś czas i retransmituje to, co wysłał wcześniej
- jeśli nastąpi kolizja, każdy z nadawców spróbuje po chwili ponownie wysłać pakiet

Skalowalność:

- co będzie, gdy zwiększy się ruch lub liczba stacji?
- ograniczony zasięg – konieczność retransmisji przez niektóre stacje pakietów dla innych stacji

Model OSI/ISO

W celu umożliwienia współdziałania wielu różnych implementacji protokołów sieciowych i wykorzystania różnych mediów transmisyjnych, stworzono warstwowy model OSI/ISO:



Sieć ethernet – warstwa 2

- Metoda dostępu – CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*)
- Adresowanie – każda karta sieciowa ma swój adres – 6-bajtową liczbę (pierwsze 3 bajty przypisane do producenta, np. 08:00:20 – Sun, 00:e0:63 – Cabletron, itd.)
- Adres FF:FF:FF:FF:FF:FF jest adresem specjalnym – broadcast
- Karta sieciowa odbiera pakiety tylko wtedy, gdy zawierają jej adres lub adres broadcast (pozostałe ignoruje).



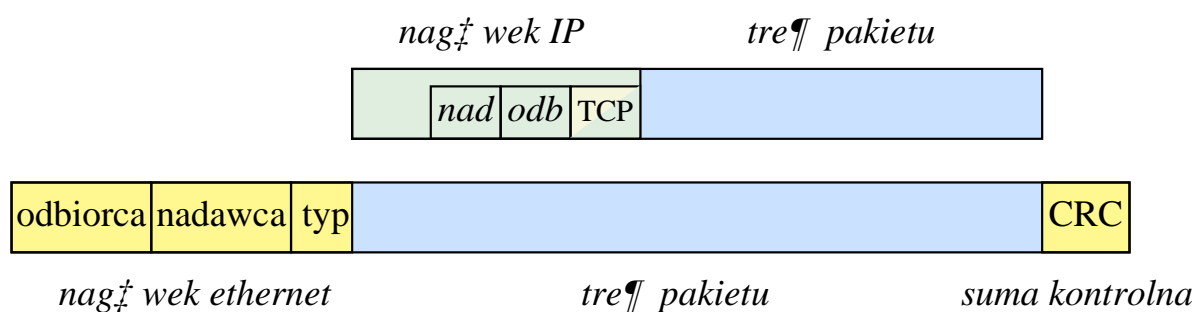
nagłówek ethernet

treść pakietu

suma kontrolna

Protokoły sieciowe IP – warstwa 3

- wewnątrz ramek ethernet zawarte są pakiety IP:



- nagłówek IP zawiera: adres IP nadawcy i odbiorcy, typ pakietu, TTL, numer protokołu, identyfikator pakietu, opcje, długość, itp. – łącznie 20 bajtów.
- adresy: 32-bitowe (4 bajty)
- typ protokołu: wersja 4 (IPv4)
- numer protokołu: TCP, UDP, ICMP
- TTL – *Time To Live* – maksymalny czas życia pakietu
- identyfikator pakietu – pozwala wykryć duplikaty lub zgubienie pakietu
- fragmentacja – czy występuje, czy ostatni fragment, offset od początku

TCP/UDP – warstwa 4

- Numer portu nadawcy i odbiorcy
- Długość pakietu
- Suma kontrolna TCP/UDP
- Opcje TCP
- Numer pakietu TCP i potwierdzenie pakietu drugiej strony

Przykładowy pakiet

```

ETHER:  ----- Ether Header -----
ETHER:  Packet 16 arrived at 0:58:9.26
ETHER:  Packet size = 106 bytes
ETHER:  Destination = ff:ff:ff:ff:ff:ff, (broadcast)
ETHER:  Source       = 0:e0:63:4:40:c0, Cabletron
ETHER:  Ethertype = 0800 (IP)
ETHER:
IP:  ----- IP Header -----
IP:  Version = 4
IP:  Header length = 20 bytes
IP:  Type of service = 0x00
IP:  xxx. .... = 0 (precedence)
IP:  ...0 .... = normal delay
IP:  .... 0... = normal throughput
IP:  .... .0.. = normal reliability
IP:  Total length = 92 bytes
IP:  Identification = 35629
IP:  Flags = 0x0
IP:  .0.. .... = may fragment
IP:  ..0. .... = last fragment
IP:  Fragment offset = 0 bytes
IP:  Time to live = 1 seconds/hops
IP:  Protocol = 17 (UDP)
IP:  Header checksum = a551
IP:  Source address = 156.17.40.65, hop.ict.pwr.wroc.pl
IP:  Destination address = 156.17.40.95, 156.17.40.95
IP:  No options
IP:
UDP:  ----- UDP Header -----
UDP:  Source port = 520
UDP:  Destination port = 520 (RIP)
UDP:  Length = 72
UDP:  Checksum = 1099
UDP:
RIP:  ----- Routing Information Protocol -----
RIP:  Opcode = 2 (route response)
RIP:  Version = 1
RIP:
RIP:  Address                      Port  Metric
RIP:  156.17.42.160  NET-42-160.ict.pwr.wroc.pl 0    1
RIP:  156.17.40.192  156.17.40.192    0    1
RIP:  156.17.30.64   156.17.30.64    0    1

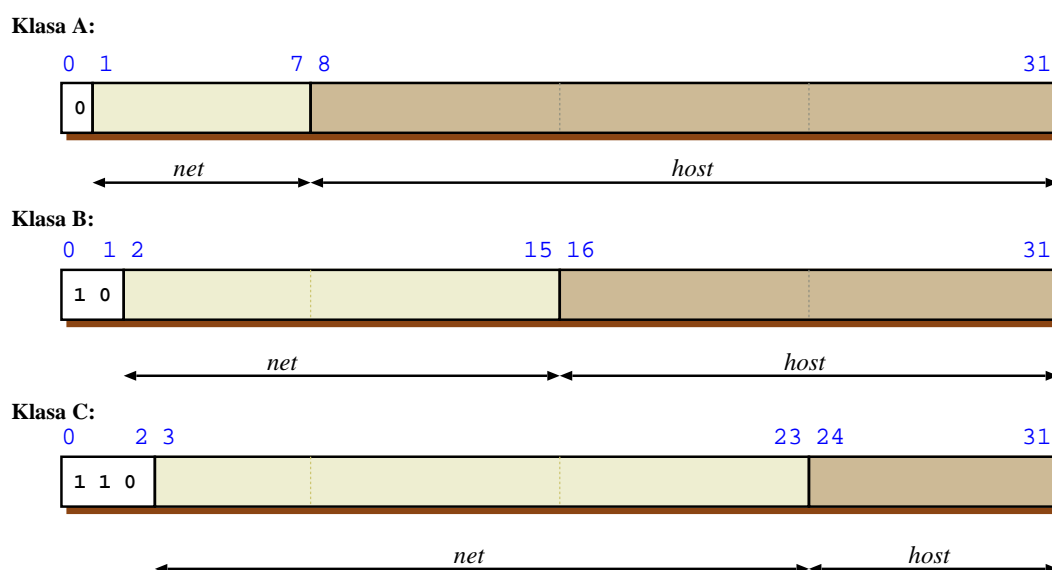
      0: ffff ffff ffff 00e0 6304 40c0 0800 4500  ....c.@...E.
      16: 004c 8b2d 0000 0111 a551 9c11 2841 9c11  ...-....Q..(A..
      32: 285f 0208 0208 0098 1099 0201 0000 0002  (.....
      48: 0000 9c11 2aa0 0000 0000 0000 0000 0000  ....*.....
      80: 0001 0002 0000 9c11 28c0 0000 0000 0000  .....(.....
      96: 0000 0000 0001 0002 0000 9c11 1e40 0000  .....@...
     112: 0000 0000 0000 0000 0001  .....

```

Adresy w sieciach IP

Każdy komputer musi mieć swój adres w sieci, jeśli ma być osiągalny

Adres internetowy to 32-bitowa liczba, podzielona zwykle na oktety i zapisywana w notacji xxx.xxx.xxx.xxx. Każdy adres należy do jakiejś *klasy* – A, B, C, D lub E:



- Klasa A – Adresy 0.x.x.x–127.x.x.x, maska 255.0.0.0
- Klasa B – Adresy 128.0.x.x–191.255.x.x, maska 255.255.0.0
- Klasa C – Adresy 192.0.0.x–223.255.255.x, maska 255.255.255.0
- Klasa D – Adresy 224.0.0.x–239.255.255.x – multicasting
- Klasa E – Adresy 240.0.0.x–255.255.255.x – eksperymentalne

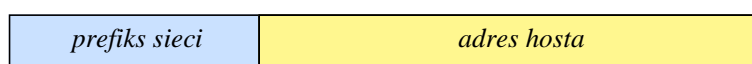
Adresy do prywatnego wykorzystania: (RFC 1918)

- 10.0.0.0 - 10.255.255.255 (10/8)
- 172.16.0.0 - 172.31.255.255 (172.16/12)
- 192.168.0.0 - 192.168.255.255 (192.168/16)

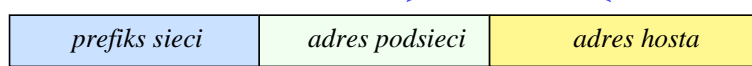
Maski sieciowe i routing

W obrębie klasy można dokonywać dodatkowych podziałów na podsieci, np. przyjmując zawężoną maskę. (RFC 950)

Dwupoziomowa hierarchia podziału na klasy:



Trzypoziomowa hierarchia podziału bezklasowego:



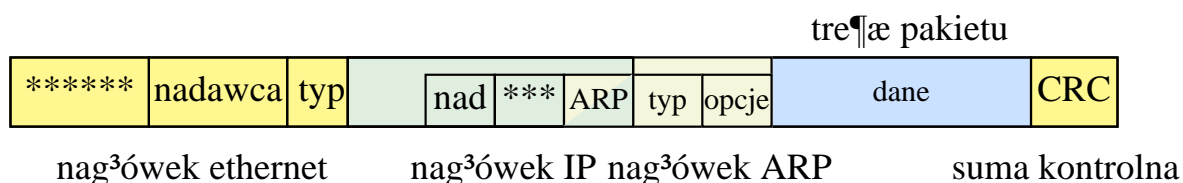
- taki podział na podklasy (ang: subnetting) pozwala na logiczną i fizyczną segmentację sieci „lokalnej”
- Przykład: w sieci WASK o adresie klasy B 156.17.x.x przyjęto maskę podsieci 255.255.255.224, czyli 27 bitów na adres sieci, 5 bitów na adres hosta.

	Prefiks klasy B			
Maska klasy B:	11111111.11111111.00000000.000 00000			
Maska podsieci:				
255.255.255.224:	11111111.11111111.11111111.111 00000			
156.17.33.34	10011100.00010001.00100001.001 00010			
	<-----> <--->			
	podsieć		host	

- W jednej podsieci mieszczą się 32 adresy IP.
- Zawsze istnieje 8 podsieci o takich samych 3 pierwszych oktetach (podsieci 0, 32, 64, 96, 128, 160, 192 i 224).
- Adres hosta z samymi zerami jest adresem zarezerwowanym (adres sieci)
- Adres hosta z samymi jedynekami to adres „broadcast”
- Efektywnie pozostaje 30 adresów do wykorzystania. W kolejnych podsieciach będą to odpowiednio: 1-30, 33-62, 65-94, 97-126, 129-158, 161-190, 193-222, 225-254.

Odwzorowanie adresów warstw 2 i 3

- Chcąc wysłać pakiet IP do komputera znajdującego się w sieci lokalnej, stacja wysyłająca musi w nagłówku ramki ethernet umieścić 6-bajtowy adres karty sieciowej odbiorcy.
- Skąd dowiedzieć się jaki to adres, jeśli znamy tylko adres IP?
- Tłumaczeniem adresów warstwy 2 (ethernet) na adresy warstwy 3 (IP) i odwrotnie zajmują się specjalne protokoły sieciowe – ARP (*Address Resolution Protocol*) i RARP (*Reverse ARP*);
- stacja wysyła zapytanie ARP-Request będące pakietem broadcast z zapytaniem ARP o adres IP:



- pakiet odbierają wszystkie komputery w segmencie lokalnym, a właściwy odpowiada na niego, ze swoim adresem karty ethernet w polu nadawcy (typ pakietu: ARP-response);
- odpowiedzieć mogą także inne urządzenia (np. switch), „podszywając się” z adresem właściwego nadawcy – proxy ARP;
- inicjator zapytania zapamiętuje poznane tłumaczenie IP-ethernet na jakiś czas (15-30 minut), by nie wysyłać ciągle zapytań ARP;
- RARP – inicjowany w celu dowiedzenia się z serwera własnego adresu IP (np. podczas bootowania stacji bezdyskowych).

Abstrakcyjny model komunikacji sieciowej

UDP

- Bezpołączeniowa transmisja datagramów
- Pakiety mogą zaginać
- Mogą pojawiać się duplikaty
- Warstwa sieciowa gwarantuje, że pakiet dotrze w postaci nieuszkodzonej lub nie dotrze w ogóle
- Program sam musi zadbać o poprawne dostarczenie pakietu poprzez odpowiednie potwierdzenia i ew. retransmisję
- Brak fazy nawiązywania połączenia, a więc krótszy niż w TCP czas przesłania pierwszych pakietów

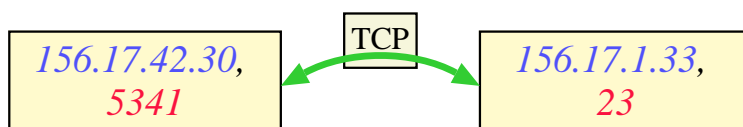
TCP

- Transmisja połączeniowa
- Nawiązanie połączenia zajmuje trochę czasu (trzyetapowa konwersacja: wysłanie żądania, potwierdzenie gotowości, ustanowienie kanału komunikacyjnego)
- Warstwa sieciowa gwarantuje, że wszystkie pakiety dotrą na miejsce i we właściwej kolejności (a jeśli nie – zostanie zwrócony odpowiedni błąd)
- Brak wyraźnego podziału na pakiety – po nawiązaniu połączenia dane widzimy jako strumień, w którym można wyróżnić jedynie początek i koniec
- Raz nawiązane połączenie musi zostać poprawnie zamknięte (znów trzyetapowa konwersacja), chyba że zostanie zerwane (skomplikowane protokoły wykrywania zerwanych połączeń)

Identyfikacja połączeń

- Aby rozpocząć komunikację między dwoma komputerami (a następnie odpowiednio kierować nadchodzące pakiety do właściwego procesu) potrzebne są ich adresy sieciowe, ale nie tylko, bo takich sesji komunikacyjnych może być wiele.
- Na tej samej maszynie może być wiele procesów odpowiedzialnych za różne połączenia², więc sam adres nie wystarczy – potrzebne jeszcze coś: *numer portu* – 16-bitowa liczba typu `int`;
- Numer portu przypisany jest do usługi. Ale co będzie, jeśli mamy nawiązanych kilka połączeń z tą samą usługą? Po czym je rozróżnić?

Każde połączenie IP jest identyfikowane przez *PIĘĆ* parametrów:



- Adres IP jednej strony
- Numer portu pierwszej strony
- Adres IP drugiej strony
- Numer portu drugiej strony
- Typ protokołu: TCP lub UDP

Jeśli nawiązujemy połączenie z usługą TELNET (port TCP/23), odbywa się to z losowo wybranego portu po stronie nadawcy. Po zaakceptowaniu połączenia (w chwili jego nawiązywania) może także ulec zmianie numer portu u strony odbierającej połączenie.

²w przypadku UDP jest to tylko bezpołączeniowa komunikacja między 2 procesami, ale dla uproszczenia nazwijmy to połączeniem

Problem kolejności bajtów

- Procesory takie jak Zilog Z-80, Intel 80x86, Pentium itp., przechowują dane w formacie *little-endian*: najpierw najmniej znaczący bajt, potem bardziej znaczące;
- Procesory Motorola 680x0, SPARC, RISC, MIPS – kolejność *big-endian*: najpierw najbardziej znaczący bajt, potem coraz mniej znaczące.

Przesyłając dane pomiędzy różnymi typami procesorów (w szczególności – przy konstruowaniu nagłówek zawierających adresy, numery portów, długości, itp.) musimy zdecydować się na jeden rodzaj – wybrany został porządek *big-endian*.

W celu konwersji z porządku hosta do porządku sieciowego należy użyć jednej z funkcji (makr):

- *htons()* – dla danych typu `short int` (np. numery portów)
- *htonl()* – dla danych typu `int` (np. adresy IP)

i analogicznie, z porządku sieciowego do porządku hosta:

- *ntohs()*
- *ntohl()*

W systemach z procesorem typu *little-endian* makra te dokonują odpowiedniej konwersji, a w systemach typu *big-endian* – są to puste definicje.

Model komunikacji klient–serwer

- Z reguły komunikacja sieciowa odbywa się z użyciem *serwerów* oczekujących non-stop na przyjęcie zgłoszenia od *klientów*, czyli programów nawiązujących połączenie.
- W przypadku transmisji UDP także klient musi zarejestrować chęć odbierania pakietów adresowanych na konkretny numer portu, więc z sieciowego punktu widzenia – po części staje się także serwerem.
- Serwer oczekuje na zgłoszenia pod *znanym* numerem portu, a po przyjęciu zgłoszenia uruchamia swoją kopię (*fork()*), która będzie obsługiwać tego klienta lub robi to równolegle z obsługą innych klientów w głównym wątku, możliwe jest więc wielokrotne jednoczesne korzystanie z tej samej usługi.
- Trzymanie „w pogotowiu” serwerów wszystkich usług, także tych bardzo rzadko używanych, to marnowanie pamięci – dlatego większość może być uruchamiana przez program *inetd* – *Internet Super-daemon*.
- „Znane” lub „zarezerwowane” numery portów to numery wymienione w pliku */etc/services*, np.:
 - 23/tcp – TELNET
 - 21/tcp – FTP
 - 520/udp – RIP
 - 6667/tcp – IRC
 - itp.

Uproszczony model OSI:

- 5-7 warstwa procesów (procesy/daemony)
- 3-4 warstwa transportowa (UDP, TCP)
- 2-3 warstwa sieciowa (IP)
- 1-2 warstwa łącza (interfejs sprzętowy)

Protokoły sieciowe

Standardowe usługi, takie jak zdalny dostęp, transfer plików, poczta elektroniczna itp. są przypisane do stałych numerów portów.

- Usługi zdalnego dostępu
 - TELNET, RLOGIN
 - RSH, REXEC
 - SSH, STELNET
- Usługi tłumaczenia nazw i katalogowe
 - DNS (Domain Name System)
 - NIS (Network Information System)
 - NIS+ (NISPLUS – zaawansowana wersja NIS)
 - FNS (Federated Naming System)
- Usługi zdalnej autoryzacji
 - RADIUS
 - TACACS, TACACS+
- Usługi poczty elektronicznej i Usenet news
 - SMTP (Simple Mail Transfer Protocol)
 - POP2, POP3
 - IMAP
 - NNTP, NNRP
- Synchronizacja czasu
 - NTP (Network Time Protocol)
 - RDATE

- Zdalny dostęp do plików
 - NFS, RFS
 - LOCK, STAT
 - AFS
 - SAMBA
- Zarządzanie siecią i usługi wspomagające
 - ECHO, CHARGEN, DISCARD – testowanie sieci
 - SNMP (Simple Network Management Protocol)
 - DHCP
 - BOOTP
 - TFTP (Trivial File Transfer Protocol)
 - RIP, OSPF, IGRP, BGP – routing
- Usługi systemów UNIX w sieciach lokalnych
 - FINGER, RWHO
 - LPR
 - TALK
 - SYSLOG
- Usługi informacyjne i transfer plików
 - HTTP (WWW)
 - GOPHER
 - WHOIS
 - FTP
 - ARCHIE

i wiele innych...

Powiązania między usługami a numerami portów

Każdy protokół związany jest także z konkretnym typem komunikacji, np. SMTP zawsze używa TCP, RIP zawsze używa komunikacji UDP, jednak niektóre usługi mogą być oferowane w obu typach komunikacji:

- RDATE – zamiennie, pełna funkcjonalność w TCP i UDP;
- DNS – UDP używany do zapytań o pojedyncze adresy, TCP do transferów pełnych stref pomiędzy serwerami DNS.

Przypisanie usług do konkretnych numerów portu można znaleźć w pliku `/etc/services` lub za pomocą funkcji systemowych `getservbyname()`, `getservbyport()`, `getservent()` i innych.

```
#
# Network services, Internet style
#
echo          7/tcp
echo          7/udp
daytime       13/tcp
daytime       13/udp
ftp-data      20/tcp
ftp           21/tcp
ssh           22/tcp
telnet        23/tcp
smtp          25/tcp      mail
domain        53/udp
domain        53/tcp
http          80/tcp      www
pop2          109/tcp      pop-2      # Post Office Protocol - V2
pop3          110/tcp      # Post Office Protocol - Version 3
sunrpc        111/udp      rpcbind
sunrpc        111/tcp      rpcbind
...
irc           6667/tcp
```

Tłumaczenie nazw usług w systemie UNIX

```
#include <netdb.h>
```

```
struct servent *getservbyname(const char *name, const char *proto);  
struct servent *getservbyport(int port, const char *proto);
```

```
int setservent(int stayopen);  
struct servent *getservent(void);  
int endservent(void);
```

- *getservbyname()* i *getservbyport()* pozwalają przetłumaczyć pojedyncze nazwy usług na numery portów lub odwrotnie;
- Dane zwracane są w strukturze `struct servent`:

```
struct servent {  
    char *s_name;          /* oficjalna nazwa usługi */  
    char **s_aliases;      /* lista aliasów, zakończona NULL */  
    int s_port;            /* numer portu */  
    char *s_proto;         /* nazwa protokołu */  
};
```

- *getservent()* pozwala sekwencyjnie przeglądać listę usług, korzystając z lokalnego pliku */etc/services* lub serwerów NIS/NIS+;
- *getservent()*, *getservbyname()* i *getservbyport()* zwracają wskaźnik na statyczną strukturę danych w pamięci systemowej, dlatego nie można tych funkcji wykorzystywać w programach wielowątkowych. Zamiast nich należy korzystać z *getservent_r()*, *getservbyname_r()*, *getservbyport_r()*.

Przykład protokołu – SMTP

Przykładowa konwersacja z demonem poczty elektronicznej wykonana przy pomocy programu `telnet`:

```
% telnet cyber 25
Trying 156.17.41.30...
Connected to cyber.
Escape character is '^]'.
220 cyber.ict.pwr.wroc.pl ESMTP Sendmail 8.9.3 Mon, 17 Apr 2000 21:00:00 +0200 (MET DST)
ehlo okapi.ict.pwr.wroc.pl
250-cyber.ict.pwr.wroc.pl Hello ts@okapi [156.17.42.30], pleased to meet you
250-8BITMIME
250-SIZE 1500000
250-ETRN
250 HELP
help
214-This is Sendmail version 8.9.3
214-Commands:
214-   HELO   EHLO   MAIL   RCPT   DATA   EXPN
214-   RSET   NOOP   QUIT   HELP   VRFY   VERB
214-For more info use "HELP <topic>".
214-To report bugs in the implementation send email to
214-   sendmail@CS.Berkeley.EDU.
214-For local information send email to Postmaster at your site.
214 End of HELP info
mail from: <santa@heaven.org>
250 <santa@heaven.org>... Sender ok
rcpt to: <ts@cyber.ict.pwr.wroc.pl>
250 <ts@cyber.ict.pwr.wroc.pl>... Recipient ok
rcpt to: <postmaster@ict.pwr.wroc.pl>
250 <postmaster@ict.pwr.wroc.pl>... Recipient ok
rcpt to: <zxcvbnm@cyber>
550 <zxcvbnm@cyber>... User unknown
data
354 Enter mail, end with "." on a line by itself
From: Swiety Mikolaj <santa@heaven.org>
To: Wszystkie grzeczne dzieci (undisclosed recipients at cyber)
Subject: Swieta...

Wprawdzie to nie te swieta, ale i tak -- Wesołych Świąt!
.
250 VAA08247 Message accepted for delivery
quit
221 cyber.ict.pwr.wroc.pl closing connection
Connection closed by foreign host.
```

Protokoły sieciowe większości usług z reguły stosują się do następujących zasad:

- komunikacja odbywa się w sposób całkowicie tekstowy (z wyjątkiem fragmentów wymagających przesłania plików/danych binarnych);
- klient wydaje tekstowe polecenia zakończone znakiem końca linii;
- serwer odpowiada w sposób tekstowy, rozpoczynając linię 3-cyfrowym kodem, po którym następuje dodatkowa informacja dla użytkownika;
 - Trzy cyfry i minus oznaczają, że kod odpowiedzi będzie kontynuowany w następnej linii;
 - Trzy cyfry i spacja oznaczają, że jest to ostatnia linia.
- Pierwsza cyfra określa, czy wykonanie operacji się udało:
 - Kody **2xx** oznaczają poprawne zakończenie operacji;
 - **3xx** – poprawne zakończenie wydanej komendy, ale wymagane jest kontynuowanie rozpoczętej operacji zgodnie z przyjętym protokołem (np. po podaniu komendy „`user abc`” może się pojawić odpowiedź „`331 Password required for user abc`”);
 - **4xx** – tymczasowy błąd, np. brak wolnej pamięci, zajęte zasoby, itp. Próba może być ponowiona za jakiś czas;
 - **5xx** – błąd, np. zła nazwa użytkownika, złe parametry, itp. Nie ma sensu ponawiać prób, bo w tej postaci zawsze będzie generowany błąd.
- dzięki tekstowemu charakterowi transmisji (i komentarzom po kodach numerycznych) możliwe jest „ręczne” testowanie połączeń nawet bez specjalnego programu klienta;
- programy klienta z reguły interpretują wyłącznie numeryczne kody zakończenia operacji. Ścisła interpretacja pierwszej (i drugiej) cyfry pozwala na poprawną reakcję nawet w przypadku rozbieżności w implementacji protokołu po obu stronach połączenia.

Tworzenie gniazdek

Komunikacja sieciowa w systemie UNIX odbywa się za pomocą *gniazd sieciowych* (zwanym też *gniazdkami* lub *gniazdkami BSD*).

- Gniazdko jest przez system UNIX traktowane tak, jak deskryptor otwartego pliku – można z niego czytać funkcją *read()*, można do niego pisać przez *write()*, zamknąć je za pomocą *close()*, a także wykonać operacje możliwe tylko na deskryptorach będących gniazdkami;
- Gniazdko istnieje w określonej *domenie adresowej*, co implikuje w jaki sposób zapisywany będzie *adres gniazdka* – stosowane obecnie domeny to:
 - *AF_UNIX* – gniazdko systemu UNIX, związane z systemem plików;
 - *AF_INET* – gniazdko sieciowe, IPv4;
 - *AF_INET6* – nowe gniazdko sieciowe, IPv6;
 - Mniej popularne: *AF_ISO*, *AF_NS* (Xerox).
- Oprócz *domeny adresowej* na sposób komunikacji ma wpływ *typ gniazdka*:
 - *SOCK_STREAM* – gniazdko strumieniowe (TCP);
 - *SOCK_DGRAM* – gniazdko datagramowe (UDP);
 - *SOCK_RAW* – gniazdko operujące w warstwie 2 modelu OSI/ISO (np. ARP, pakiety ICMP i inne);
 - inne.
- Nowe gniazdko można utworzyć za pomocą *socket()* lub *socketpair()* (ta druga wyłącznie w domenie *AF_UNIX*);
- Zanim gniazdko będzie nadawało się do użycia, należy mu nadać *adres* za pomocą funkcji *bind()*.

Funkcja socket()

```
#include <sys/types.h>
#include <sys/socket.h>
```

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

- Domena – AF_UNIX lub AF_INET
- Typ – SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET
- Protokół – ściśle związany z typem gniazdka, np. gniazdka AF_INET SOCK_STREAM korzystają zawsze z protokołu IPPROTO_TCP, a internetowe gniazdka typu SOCK_DGRAM – protokołu IPPROTO_UDP.

Rozróżnienie może być istotne w przypadku gniazdek SOCK_RAW, gdzie możemy wybrać IPPROTO_ICMP lub IPPROTO_RAW.

Podanie wartości 0 spowoduje wybranie protokołu właściwego dla podanej kombinacji domeny i typu gniazdka.

Funkcja socketpair()

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socketpair(int domain, int type, int protocol, int sv[2]);
```

Tworzy dwa nie połączone ani nie nazwane deskryptory gniazd, z których każdy może służyć do dwukierunkowej komunikacji (inaczej niż w strumieniach FIFO czy deskryptorach zwracanych przez *pipe()*).

- Domena AF_UNIX
- Typ SOCK_STREAM lub SOCK_DGRAM
- Protokół – wartość 0 wybiera właściwy.

Nadanie nazwy – funkcja `bind()`

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int bind(int s, const struct sockaddr *name, socklen_t *namelen);
```

- `sockfd` – deskryptor utworzonego wcześniej gniazda;
- `name` – odpowiednia do domeny adresowej struktura zawierająca adres nadawany gniazdku;
- `namelen` – długość struktury adresowej.

W przypadku gniazdek sieciowych IPv4 adres określa struktura `sockaddr_in` zdefiniowana w pliku `/usr/include/netinet/in.h`:

```
struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr  sin_addr;
    unsigned char   sin_zero[8];
};
```

Sktruktora `in_addr` oraz typ `in_port_t` są określonym sposobem zdefiniowania adresu/portu w sposób przenośny, by nadawały się do zastosowania w wielu architekturach procesorów i różnych systemach. W rzeczywistości są to odpowiednio 32- i 16-bitowe liczby typu `unsigned int` – ZAWSZE zapisane w porządku sieciowym.

Nadanie nazwy jest konieczne:

- w przypadku *serwerów* – zarówno połączeniowych, jak i bezpołączeniowych – pozwala odnotować w systemie ogólnie znany adres serwera i powiązać nadchodzące pakiety z procesem, który ma je otrzymywać;
- w przypadku transmisji bezpołączeniowej – zarówno po stronie serwera, jak i klienta. Pakiety wysyłane z systemu będą miały ustawiony określony adres nadawcy, co umożliwia odesłanie mu odpowiedzi;

Nadanie nazwy jest opcjonalne:

- jeśli w systemie znajduje się kilka interfejsów sieciowych³ i chcemy wymusić skorzystanie z wybranego. Możemy też użyć pseudo-adresu `INADDR_ANY`, który zezwala na użycie dowolnego adresu/interfejsu.
- jeśli klient chce uzyskać przed uzyskaniem połączenia wybrany numer portu. Domyślnie stosowany port 0 pozwala na wybranie dowolnego numeru portu przez system, w chwili łączenia z serwerem.

Porty poniżej 1024 (`IPPORT_RESERVED`) są portami *zarezerwowanymi* lub *uprzywilejowanymi* – taki numer może przypisać gniazdku funkcja `bind()` wywołana wyłącznie przez użytkownika z `euid==0` (root).

Pierwszy wolny (od góry) numer portu z zakresu zarezerwowanego (512-1023) można przydzielić funkcją `rresvport(int* port)`, która przydziela gniazdko i nadaje mu nazwę (zwracając jego deskryptor).

Porty powyżej 5000 (`IPPORT_USERRESERVED`) są zarezerwowane na usługi serwerów uruchamianych przez użytkowników (bez uprawnień użytkownika root).

Jeśli numer portu zostanie ustawiony na 0, w chwili uzyskania połączenia zostanie przydzielony pierwszy wolny (unikalny) numer z zakresu 1024-5000.

³`eth, lo...`

Usługi tłumaczenia nazw – DNS/NIS/NIS+

```
#include <netdb.h>
```

```
struct hostent *gethostbyname(const char *name);  
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

To, w jaki sposób system UNIX tłumaczy nazwy na adresy i odwrotnie, zależy od konfiguracji systemu:

- Plik */etc/nsswitch.conf* określa z jakich usług sieciowych należy korzystać – lokalnych plików, NIS, NIS+ czy DNS (i w jakiej kolejności);
- Plik */etc/hosts* zawiera nazwy hostów i adresy konieczne podczas uruchamiania systemu (np. do skonfigurowania kart sieciowych, gdy jeszcze nie można skorzystać z DNS);
- Plik */etc/resolv.conf* określa adresy serwerów DNS oraz domyślne domeny doklejane do zapytań.

Korzystanie z funkcji systemowych pozwala uniezależnić się od różnorodności stosowanych systemów tłumaczenia nazw (pliki lokalne, DNS, NIS, NIS+)

Korzystając z tych funkcji należy pamiętać, że:

- Jednej nazwie hosta może być przyporządkowanych wiele adresów (np. w przypadku routerów lub hostów mających kilka interfejsów sieciowych) – należy przejrzeć je wszystkie;
- Każdy host może oprócz nazwy *kanonicznej* (głównej) posiadać i inne nazwy, zwyczajowe, zwane aliasami, np. `ftp.pwr.wroc.pl` jest aliasem nazwy kanonicznej `panorama.wcss.wroc.pl`;
- Wszystkie adresy zwracane są w porządku sieciowym i może być konieczne użycie odpowiednich makr *htonl()* lub *ntohl()*.

Adresy i nazwy zwracane są w strukturze `hostent`

```
struct hostent {
    char    *h_name;          /* canonical name of host */
    char    **h_aliases;      /* alias list */
    int     h_addrtype;       /* host address type */
    int     h_length;         /* length of address */
    char    **h_addr_list;    /* list of addresses */
};
```

Struktura ta zawiera pola pozwalające przekazać adres niezależnie od stosowanej domeny adresowej.

Oprócz funkcji szukających adresu/nazwy wybranego hosta istnieją funkcje pozwalające sekwencyjnie przeglądać zawartość pliku `/etc/hosts` lub map NIS/NIS+:

```
int sethostent(int stayopen);
struct hostent *gethostent(void);
int endhostent(void);
```

Wszystkie z funkcji: `gethostbyname()`, `gethostbyaddr()`, `gethostent()` zwracają wskaźnik na statyczną strukturę danych w pamięci systemowej, dlatego nie można tych funkcji wykorzystywać w programach wielowątkowych. Zamiast nich należy korzystać z `gethostbyname_r()`, `gethostbyaddr_r()` i `gethostent_r()`, które dodatkowo wymagają podania wskaźnika na bufor, gdzie zostaną zwrócone szukane dane, jego długości, oraz adresu zmiennej pełniącej funkcję zmiennej `errno` (w której zostanie umieszczony kod ewentualnego błędu).

Inne przydatne funkcje:

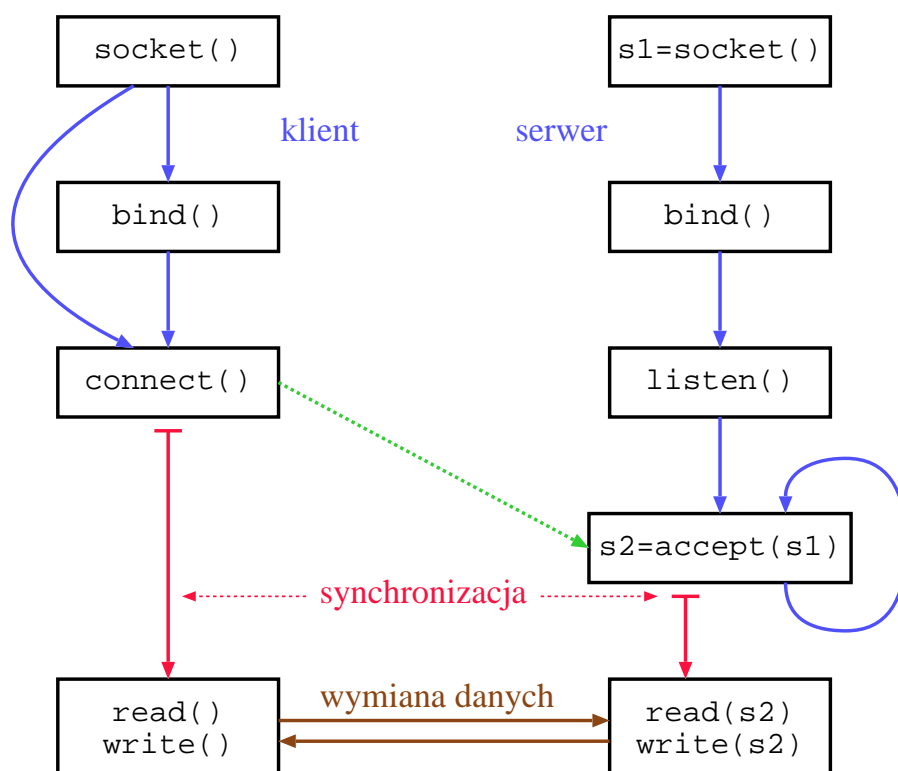
```
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(const struct in_addr in);
unsigned long inet_addr(const char *cp);
unsigned long inet_network(const char *cp);
struct in_addr inet_makeaddr(const int net, const int lna);
```

Komunikacja TCP

Nawiązywanie połączeń TCP

- Klient wywołuje funkcję *connect()*, podając adres serwera, z którym chce się połączyć;
- Serwer rejestruje swoją usługę w systemie za pomocą funkcji *listen()*, a następnie oczekuje na połączenia za pomocą *accept()*;
- Funkcje *connect()* i *accept()* są funkcjami *blokującymi*, tzn. powodują uśpienie procesu do momentu uzyskania połączenia z drugą stroną.

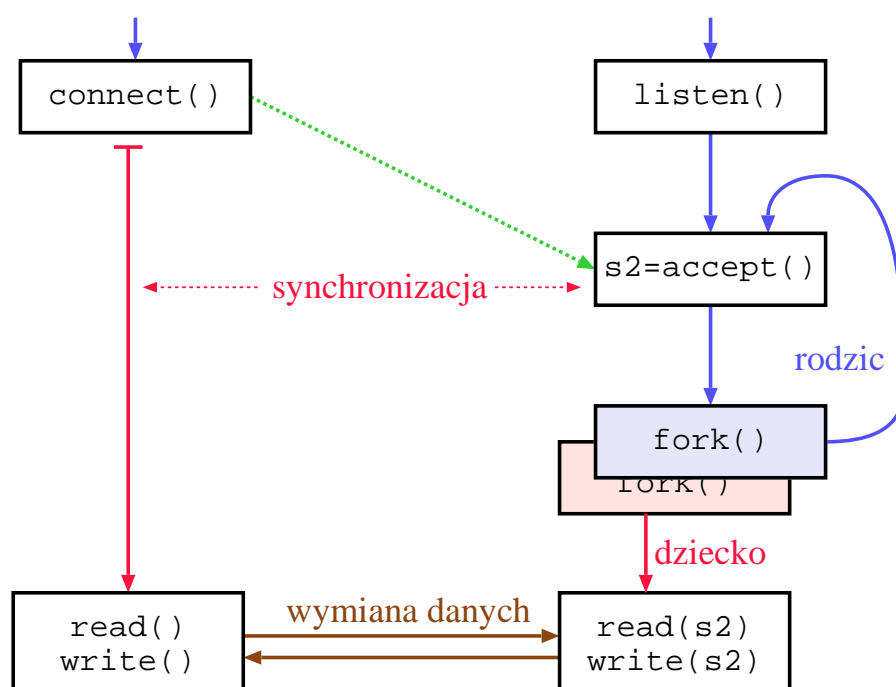


- Zaakceptowanie połączenia powoduje utworzenie przez proces serwera *nowego* gniazdka, które jest gniazdkiem *połączonym*, podczas gdy główne gniazdko (*s1*) może nadal służyć do przyjmowania nowych połączeń.

Serwery współbieżne

Po zaakceptowaniu połączenia funkcją `accept()` serwer może niezależnie obsługiwać klienta, który się z nim połączył, oraz nawiązywać nowe połączenia.

Wykorzystując funkcje nieblokujące lub inne mechanizmy, można to robić w jednym procesie, z reguły jednak wygodniej jest do obsługi połączonego klienta utworzyć nowy proces:



- Po wykonaniu `accept()` proces wykonuje `fork()`, rozdzielając się na 2 procesy;
- Rodzic zamyka gniazdko `s2` (połączone z klientem) i wraca w pętli do funkcji `accept()`, oczekując na nowe połączenia, a także od czasu do czasu „sprzątając” po procesach potomnych za pomocą `wait()`;
- Dziecko zamyka gniazdko `s1`, służące do akceptowania nowych połączeń, oraz inne niepotrzebne deskryptory przekazane mu przez rodzica, do komunikacji z klientem używając jedynie gniazdka `s2`.

Funkcja *connect()*

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int s, const struct sockaddr *name, struct _t namelen);
```

- Funkcję można zastosować zarówno w przypadku gniazd strumieniowych (TCP) jak i datagramowych (UDP).
- TCP – nawiązuje połączenie z serwerem
- UDP – tworzy asocjację (przypisanie adresów do gniazdka)
- *name* określa adres drugiego gniazdka, z którym ma nastąpić połączenie (lub w przypadku UDP – asocjacja)
- jeśli do gniazdka *s* nie jest przypisany funkcją *bind()* żaden adres, to zostaje on nadany automatycznie przez warstwę transportową w chwili nawiązania połączenia.

Funkcja *listen()*

```
int listen(int s, int backlog);
```

- Parametr *backlog* określa maksymalną wielkość kolejki połączeń oczekujących na zaakceptowanie funkcją *accept()*.
- Jeśli kolejka zostanie przepełniona klienci zaczną otrzymywać błąd *ECONNREFUSED* w gniazdkach typu *AF_UNIX*, a w gniazdkach *AF_INET* – połączenia będą ignorowane po stronie serwera, a retransmitowane po stronie klienta (aż do *ETIMEDOUT*).

Funkcja *accept()*

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

- Funkcja powoduje zablokowanie programu do czasu, gdy jest przynajmniej jedno zgłoszenie w kolejce oczekujących połączeń.
- Pierwsze zgłoszenie z kolejki zostaje zaakceptowane, tworząc nowe gniazdko, a w zmiennej `addr` zostają umieszczone dane na temat drugiej strony połączenia, o ile są znane.
- Wykonalność funkcji *accept()* można sprawdzić także za pomocą *select()* lub *poll()*.

Funkcja *shutdown()*

```
int shutdown(int s, int how);
```

- Zamyka dwukierunkowe połączenie związane z gniazdkiem `s` całkowicie, lub tylko jednostronnie.
- 0: z gniazdko nic już nie będzie czytane;
- 1: do gniazdko już nic nie będzie pisane;
- 2: gniazdko zostaje zamknięte zarówno do pisania jak i do czytania.

Serwery samodzielne i uruchamiane przez inetd

Typowy mechanizm działania serwerów sieciowych:

- Serwer główny rejestruje gniazdko przyjmujące połączenia;
- nadchodzące połączenia przejmowane są przez procesy potomne;
- zakończone połączenie z klientem oznacza koniec pracy serwera potomnego;
- proces główny oczekuje na śmierć procesów potomnych wołając regularnie *wait()* lub przechwytyując sygnały SIGCLD;
- proces główny nigdy się nie kończy.

Skoro mechanizm działania jest identyczny w przypadku większości serwerów sieciowych (TELNET, RLOGIN, TALK, FINGER itp.), to po co wielokrotnie marnować pamięć na proces główny dla każdej z usług z osobna?

- Jeden wspólny proces akceptujący połączenia dla wielu usług sieciowych i uruchamiający odpowiednie serwery potomne;
- Problemy:
 - różnorodność protokołów (TCP, UDP);
 - różne uprawnienia serwerów (działające z konta root, sys lub innego);
 - elastyczność:
 - * różne serwery w różnych katalogach dyskowych;
 - * niektóre usługi obsługiwane przez ten sam serwer;
 - * możliwość przekazania parametrów, opcji, plików konfiguracyjnych;
 - efektywność – możliwość przejęcia przyjmowania nowych połączeń przez „usamodzielniony” proces potomny.

Serwer inetd

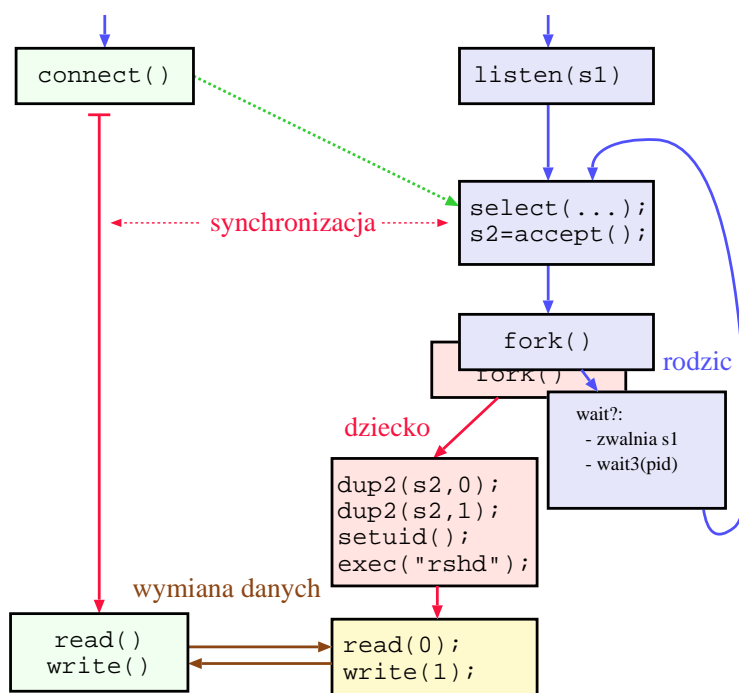
`inetd` lub `in.inetd` – „Internet-superdaemon”, program uruchamiający inne serwery na żądanie – wtedy, gdy są potrzebne, np:

- `in.telnetd`
- `in.ftpd`
- `tftpd`
- `in.fingerd`
- `in.rshd`
- `in.rlogind`
- `in.rshd`
- `in.rexecd`
- `in.identd` (`pidentd`)
- `pop3d`

Konfiguracja programu `inetd` znajduje się w pliku `/etc/inetd.conf` i określa sposób obsługi każdego z wymienionych w nim portów:

```
ftp      stream  tcp      nowait  root    /usr/sbin/in.ftpd    in.ftpd -l
telnet   stream  tcp      nowait  root    /usr/sbin/in.telnetd in.telnetd
talk     dgram   udp      wait    root    /usr/sbin/in.talkd   in.talkd
#
# Time service is used for clock synchronization.
#
time     stream  tcp      nowait  root    internal
time     dgram   udp      wait    root    internal
#
printer  stream  tcp      nowait  root    /usr/lib/print/in.lpd in.lpd
ident    stream  tcp      wait    sys     /usr/sbin/in.identd  in.identd -w -t180
```


Procesy uruchamiane przez `inetd` nie martwią się o nawiązywanie połączenia, lecz komunikują się z klientem przez swoje standardowe wejście i wyjście:

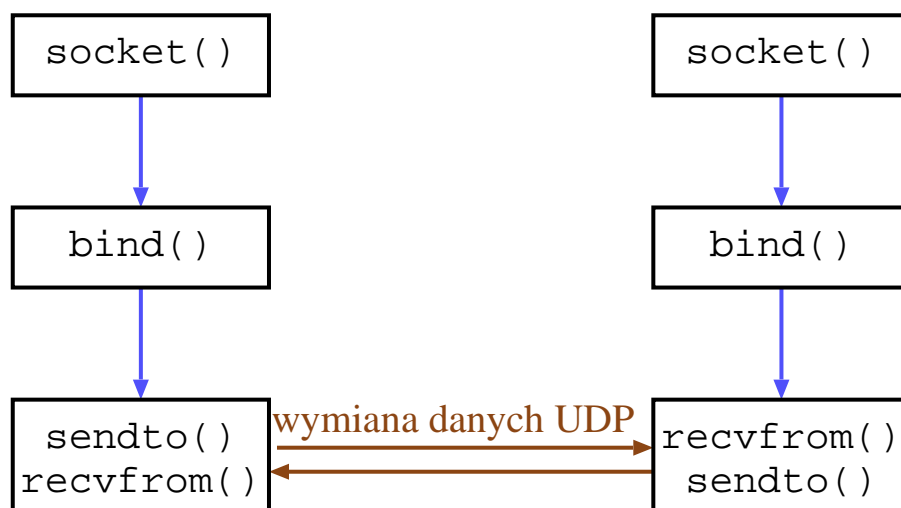


- `inetd` dla każdej usługi znalezionej w `/etc/inetd.conf` przydziela osobne gniazdko, przypisuje mu za pomocą `bind()` właściwy numer portu, wykonuje `listen()`, by zacząć nasłuchiwać na połączenia, po czym za pomocą `select()` czeka, aż na którymś z nich można będzie wykonać `accept()`;
- Po przyjęciu zgłoszenia i wykonaniu `fork()`, proces macierzysty:
 - zamyka połączone gniazdko (`s2`);
 - jeśli w konfiguracji wystąpiła opcja `wait`, odłącza się od głównego gniazdka, aż do chwili, gdy proces potomny się zakończy;
 - dalej nasłuchuje na gniazdku głównym, w przypadku `nowait`;
- Proces potomny kopiuje połączone gniazdko `s2` do deskryptorów 0, 1 i 2, zamyka wszystkie pozostałe otwarte deskryptory, jeśli trzeba – wykonuje `setuid()` i `setgid()`, po czym uruchamia odpowiedni program za pomocą `exec()`.

Komunikacja UDP

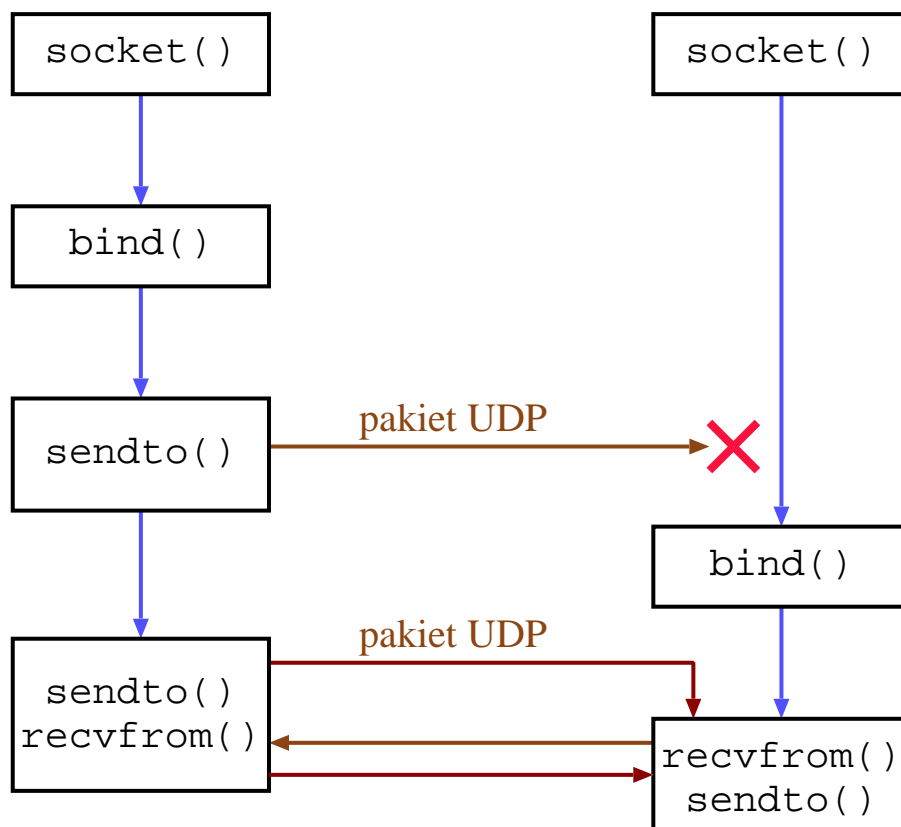
- Gniazdko `SOCK_DGRAM`, używane w komunikacji UDP, są *bezpołączeniowe*, więc nie trzeba nawiązywać połączenia funkcjami takimi, jak `connect()` i `accept()`.
- Konieczne jest zarejestrowanie w systemie chęci otrzymywania pakietów wysyłanych na żądany numer portu, co wymaga *nadania nazwy* gniazdku funkcją `bind()`.

Typowa kolejność wołania funkcji systemowych w komunikacji UDP:



- Funkcja `bind()` nadaje nazwę gniazdku (lokalny adres i lokalny numer portu), tworząc *pólasocjację*.
- `sendto()` wymaga każdorazowego podania adresu drugiej strony, do której jest wysyłany pakiet.
- `recvfrom()` wymaga każdorazowo dostarczenia zmiennej, w której zostanie zwrócony adres strony, która przysłała pakiet.

- Nie ma żadnej synchronizacji pomiędzy procesami.
- Po zarejestrowaniu chęci otrzymywania pakietów (funkcją *bind()*) tworzona jest kolejka, do której są zapisywane przychodzące pakiety.
- Warstwa transportowa nie zapewnia niezawodności przesyłanych danych nawet po zarejestrowaniu gniazdka – pakiety mogą ginać lub przychodzić w kilku kopiach.



- Pakiety przychodzące przed wykonaniem funkcji *bind()* są odrzucane (bez potwierdzenia).
- Pakiety przychodzące po wykonaniu funkcji *bind()* są kolejgowane i dostarczane do aplikacji, po wywołaniu przez nią funkcji *recvfrom()*.

Funkcje *recvfrom()* i *sendto()*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);

ssize_t sendto(int s, const void *msg, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);

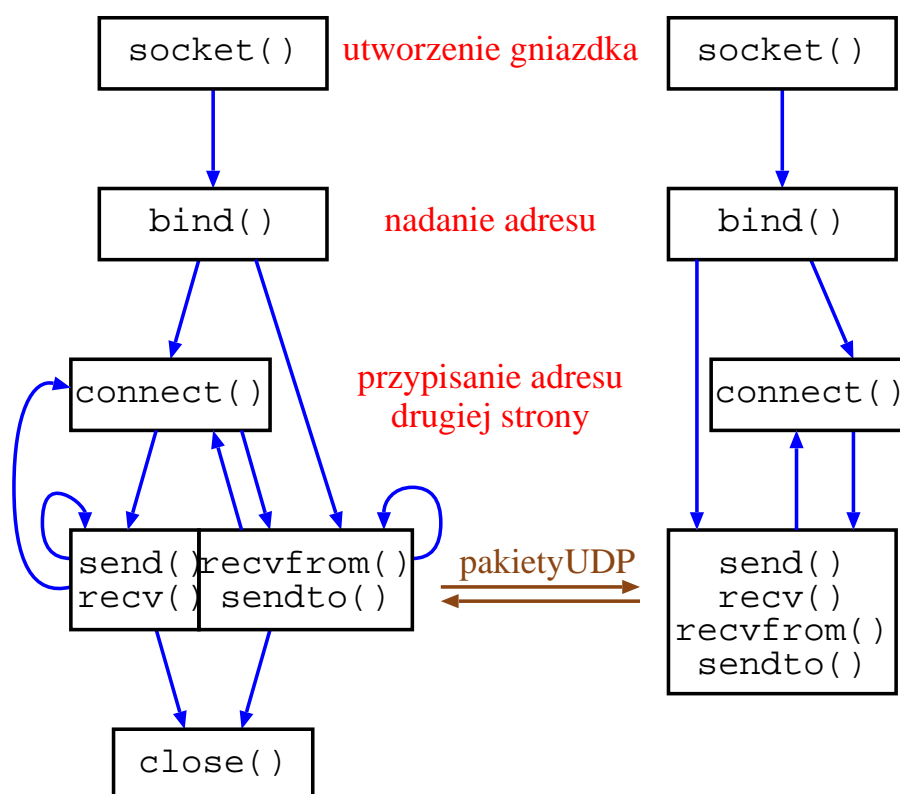
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t send(int s, const void *msg, size_t len, int flags);
```

- *sendto()* wysyła dostarczony pakiet na adres przekazany w zmiennej wskazywanej przez *from*;
- *recvfrom()* w zmiennej wskazywanej przez *from* umieszcza adres nadawcy pakietu, a długość tego adresu w zmiennej wskazywanej przez *fromlen*, chyba że *from*==NULL.
- *send()* i *recv()* wymagają wcześniejszego utworzenia pełnej asocjacji (za pomocą funkcji *connect()*), w której zawarty jest adres drugiej strony.
- Podanie w opcjach *recvfrom()* i *recv()* stałej MSG_PEEK powoduje podglądnięcie dostępnych danych, bez ich „konsumowania”.
- *recvfrom()* i *recv()* są funkcjami *blokującymi*.

Jeśli pożądanego jest, by funkcja nie blokowała aplikacji przy braku komunikatu, można skorzystać z funkcji *select()* lub przestawić gniazdko w tryb nieblokujący, za pomocą *fcntl()*. W takim przypadku *recvfrom()* i *recv()* mogą zwrócić wartość `-1` i ustawić zmienną `errno` na `EWOULDBLOCK`.

Korzystanie z `connect()`

- Działanie funkcji `connect()` w przypadku gniazdek typu `SOCK_DGRAM` ogranicza się do zarejestrowania w systemie pełnej asocjacji związanej z komunikacją, a więc także adresu drugiej strony.
- Po wykonaniu `connect()` z podanym adresem drugiej strony, w dalszej komunikacji można używać także funkcji `recv()` i `send()`.
- W trakcie komunikacji można wielokrotnie wołać `connect()` z różnymi adresami, zmieniając istniejącą asocjację.
- Po wywołaniu `connect()` z pustym adresem, ustanowiona wcześniej asocjacja zostaje usunięta i dalsza komunikacja możliwa jest wyłącznie za pomocą funkcji `recvfrom()` i `sendto()`.



Przykład serwera protokołu TCP

```
#define SERVPOR 8000

main (int argc, char ** argv)
{ int sock, newsock, pid, clen, port;
  struct sockaddr_in cl_addr, serv_addr;

  if (argc > 1)
    port = atoi(argv[1]);
  else
    port = SERVPOR;

  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    sys_err("serwer: nie można utworzyć gniazdka");
  memset(&serv_addr, 0, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(port);

  if (bind(sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
    sys_err("serwer: nie można nadać gniazdka adresu");

  listen(sock, 5);

  while(1) {
    clen = sizeof(cl_addr);
    newsock = accept(sock, (struct sockaddr*) &cl_addr, &clen);
    if (newsock < 0)
      sys_err("serwer: błąd w funkcji accept()");
    else {
      switch (pid = fork()) {
        case 0: /* dziecko */
          close (sock);
          ...
          read(newsock, ... ); write(newsock, ... );
          ...
          close(newsock);
          exit(0);
        case (-1):
          sys_err("błąd podczas fork()"); break;
      } /* switch */
    } /* else */
    close (newsock);
  } /* while */
}
```

Przykład klienta protokołu TCP

```
#define SERVPOR 8000
#define SERVADDR "127.0.0.1"

main (int argc, char ** argv)
{
    int sock, n, nw;
    struct sockaddr_in serv_addr;
    char *serv_ip;
    int serv_port;
    char buf[1024];

    if (argc>=3) {
        serv_ip = argv[1]; serv_port = atoi(argv[2]);
    } else {
        serv_ip = SERVADDR; serv_port = SERVPOR;
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(serv_ip);
    serv_addr.sin_port = htons(serv_port);

    if ((sock=socket(AF_INET, SOCK_STREAM, 0)) < 0)
        sys_err("klient: nie można utworzyć gniazdka");
    if (connect(sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
        sys_err("klient: nie można połączyć się z serwerem");

    while ( (n=read(sock, buf, sizeof(buf))) >0) {
        ...
        nw = write(sock, ..., ... );
        ...
        if ( /* koniec połączenia inicjowany przez klienta: */ ) {
            write(sock, "koniec\n", 8);
            shutdown(sock, 1); /* nie będziemy już pisać */
        }
    }

    if (n==0) { /* EOF */
        shutdown(sock, 2);
        close(sock);
    } else {
        sys_err("błąd funkcji read()");
    }
}
```

Przykład serwera protokołu UDP

```
#define PORT 7000

main (int argc, char ** argv)
{
    int sock, cl_len, n;
    struct sockaddr_in serv_addr, cl_addr;

    if ((sock=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        sys_err("server: nie można utworzyć gniazdka");

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(PORT);

    if (bind(sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
        sys_err("server: nie można nadać gniazdku lokalnego adresu");

    cl_len=sizeof(cl_addr):
    n = recvfrom(sock, buf, sizeof(buf), 0, &cl_addr, &cl_len);

    if (n<0)
        sys_err("server: błąd w funkcji recvfrom());

    strcpy(buf, "dane wysyłane do klienta");
    n=sendto(sock, buf, strlen(buf)+1, &cl_addr, cl_len);

    if (n != strlen(buf)+1) {
        sys_err("server: błąd wysyłania danych - sendto()");
    }
    close(sock);
}
```


Przykład klienta protokołu UDP

```
#define PORT 7000
#define SERWER "localhost"
#define SERWER_IP "127.0.0.1"

main (int argc, char ** argv)
{
    int sock;
    struct sockaddr_in serv_addr, cl_addr;
    struct hostent *hname;

    if ((sock=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        sys_err("klient: nie można utworzyć gniazdka");

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERWER_IP);
    serv_addr.sin_port = htons(PORT);

    if (argc>1) {
        hname=gethostbyname(argv[1]);
        if (hname != NULL) {
            memcpy(&serv_addr.sin_addr, hname->h_addr_list[0], hname->h_length);
            /* kopiujemy tylko pierwszy znaleziony adres, w bardziej rozbudowanej
               wersji moglibyśmy próbować wszystkich po kolei, do skutku */
        }
    }

    memset(&cl_addr, 0, sizeof(cl_addr));
    cl_addr.sin_family = AF_INET;
    cl_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    cl_addr.sin_port = htons(0);

    if (bind(sock, (struct sockaddr*) &cl_addr, sizeof(cl_addr)) < 0)
        sys_err("klient: nie można nadać gniazdku lokalnego adresu");
    ...
    n=sendto(sock, "test", strlen("test"), &serv_addr, sizeof(serv_addr));
    if (n != strlen("test")) {
        /* błąd wysyłania */
        ...
    }
    ...
    close(sock);
    exit(0);
}
```

Zaawansowane zagadnienia dotyczące gniazd

Funkcje wysyłania i odbierania danych

```
#include <unistd.h>
#include <sys/uio.h>

ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);

#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvmsg(int s, struct msghdr *msg, int flags);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);

struct msghdr {
    caddr_t      msg_name;           /* opcjonalny adres */
    int          msg_namelen;        /* wielkość adresu */
    struct iovec *msg_iov;           /* tablica fragmentów danych */
    int          msg_iovlen;         /* liczba elementów msg_iov */
    caddr_t      msg_accrights;
    int          msg_accrightslen;
}
```

- *readv()* i *writev()* pozwalają zminimalizować konieczność kopiowania fragmentów danych z pamięci do bufora pamięci, który byłby użyty w standardowych funkcjach *read()* i *write()*.
- *readmsg()* i *sendmsg()* pozwalają na minimalizację kopiowania danych w przypadku komunikacji UDP (można ich użyć zamiast *recvfrom()* i *sendto()*).

Zwielokrotne wejście/wyjście

- Pisanie serwerów współbieżnych wymaga obsługi wielu klientów jednocześnie.
- Najprościej, choć wcale nie najefektywniej – dla każdego klienta uruchamiać osobny proces serwera.
- Obsługa wszystkich klientów przez jeden serwer – nie można sobie pozwolić na blokowanie serwera w oczekiwaniu na dane od jednego z klientów, podczas gdy gotowe są już dane od innego.

Możliwe wyjścia z sytuacji:

- korzystanie z gniazd nieblokujących (użycie `fcntl(FNDELAY)` lub `ioctl(FIONBIO)`).
 - aktywne odpytywanie, które gniazdo jest gotowe,
 - sprawdzanie wartości `EWOULDBLOCK` przed każdą operacją mogącą zablokować proces (`read()`, `accept()` itp.)
- korzystanie z asynchronicznego wejścia/wyjścia
 - przechwytywanie sygnałów jest kosztowne,
 - otrzymanie sygnału nie informuje o tym *który* deskryptor jest gotowy do odczytu/zapisu – trzeba przebadać wszystkie.
- korzystanie z funkcji `select()` lub `poll()`
 - pozwala określić listę deskryptorów, które nas interesują,
 - czeka na gotowość dowolnego z nich w sposób bierny (umieszczając proces w kolejce oczekujących na spełnienie warunku),
 - może zostać zakończona po upływie zadanego czasu, nawet jeśli żaden z deskryptorów nie jest gotowy do zapisu/odczytu.

Funkcja select()

```
int select(int nfd, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);
```

```
void FD_ZERO(fd_set *fdset);
```

```
void FD_SET(int fd, fd_set *fdset);
```

```
void FD_CLR(int fd, fd_set *fdset);
```

```
int FD_ISSET(int fd, fd_set *fdset);
```

- Listy deskryptorów są upakowane w odpowiednich zmiennych `readfds`, `writefds` i `errorfds`, po 1 bit na deskryptor;
- Do manipulowania tymi listami służą makra takie, jak `FD_ZERO`, zerujące listę, czy `FD_SET`, ustawiające bit odpowiadający podanemu deskryptorowi pliku, gniazda, strumienia FIFO itp.
- Jeśli lista jest pusta, można zamiast niej podać wskaźnik `NULL`;
- Struktura `timeout` określa maksymalny czas oczekiwania na dostępność określonych deskryptorów.

```
struct timeval {
    long tv_sec; /* sekundy */
    long tv_usec; /* mikrosekundy */
};
```

Sposób interpretowania tego czasu:

- Powrót natychmiast, jeśli żaden deskryptor nie jest gotowy: struktura wskazywana przez `timeout` wypełniona zerami;
- Czekaj zadany czas: jeśli wartości w `*timeout` nie są obie równe zero;
- Czekaj w nieskończoność (aż któryś z deskryptorów będzie gotów): `timeout == NULL`.

Wartość zwracana przez funkcję *select()* określa liczbę deskryptorów, które są gotowe do wykonania żądanej operacji:

- -1 : błąd funkcji;
- 0 : żaden deskryptor nie jest gotów, funkcja zakończyła się z powodu przekroczenia maksymalnego czasu oczekiwania;
- > 0 : liczba deskryptorów.

Jeśli choć jeden deskryptor jest gotów, należy sprawdzić który, wołając w pętli makro `FD_ISSET`:

```
while (1) {
    wait.tv_sec=5;
    wait.tv_usec=0;

    FD_ZERO(&rd_tmpl);
    FD_SET(0, &rd_tmpl);      /* stdin */
    FD_SET(sock, &rd_tmpl);   /* gniazdko */

    max = sock;                /* najwyższy numer deskryptora wart sprawdzania */

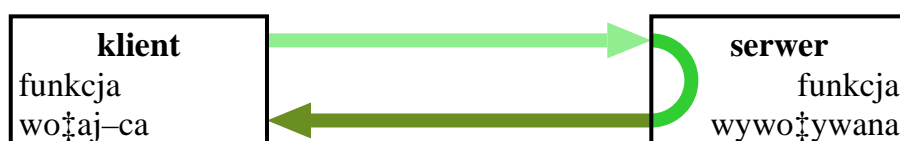
    n=select(max, &rd_tmpl, NULL, NULL, &wait);

    if (nb>0) {
        int i;
        for (i=0; i<=max; i++) {
            if FD_ISSET(i, &rd_tmpl) {
                ...
                ... read(i, ..., ...); /* deskryptor nr i jest gotów do odczytu */
                ...
            }
        } /* for */
    } /* while */
}
```

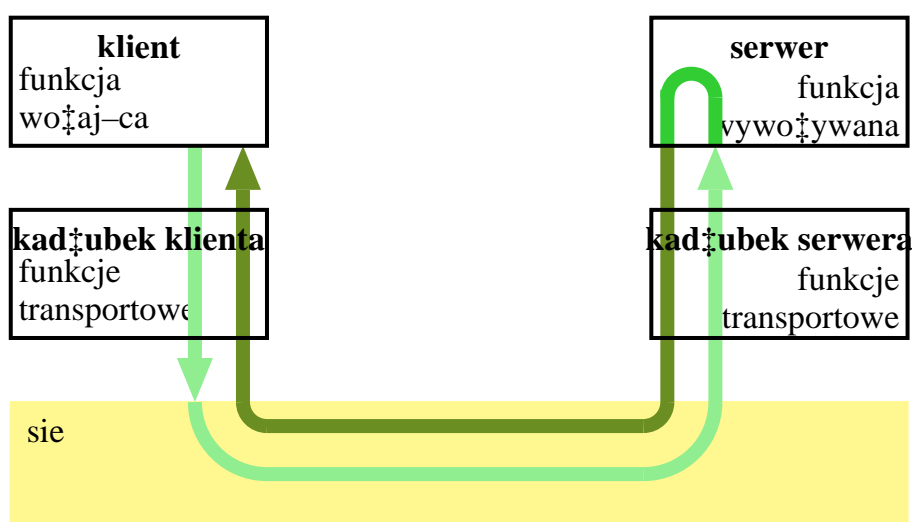
W systemie Linux i niektórych innych, zmienna `timeout` po wyjściu z *select()* zawiera czas pozostały do zakończenia funkcji gdyby żaden z deskryptorów nie był gotów do wykonania na nim operacji (czyli zadany czas minus czas faktycznego oczekiwania), w innych systemach – wartość jest nieokreślona. Dlatego zawsze trzeba tę zmienną inicjować tuż przed wywołaniem *select()*.

Funkcje RPC

RPC – Remote Procedure Call – mechanizm ułatwiający pisanie aplikacji sieciowych.



Lokalne wywołanie funkcji



Wywołanie funkcji z użyciem RPC

- Przy wywołaniu lokalnym, funkcja wołająca i funkcja wywoływana znajdują się w jednym obszarze adresowym (jednym lub kilku segmentach pamięci, ale znajdujących się w przestrzeni adresowej programu).
- Przy wywołaniu zdalnym, we wspólnej przestrzeni adresowej znajdują się odpowiednio: w kliencie: funkcja wołająca i kadłubek klienta (zastępujący funkcję wołaną), w serwerze: wykonywana funkcja oraz jej kadłubek, pobierający argumenty z sieci i przekazujący przez sieć wyniki.

Fragmenty programu generowane automatycznie:

- funkcje-kadłubki klienta (*client stub*)
- funkcje-kadłubki serwera (*server stub*)
- funkcje pomocnicze konwersji typów danych
- pliki nagłówkowe

a także:

- szkielet aplikacji serwera
- plik *makefile* ułatwiający kompilację całości

Fragmenty programu konieczne do napisania:

- funkcje klienta
- funkcje serwera
- specyfikacja interfejsu klient-serwer i używanych typów danych

Specyfikacja interfejsu jest w rzeczywistości sformalizowaną deklaracją wszystkich funkcji serwera, które mogą być wołane przez klienta, a także typów danych, używanych jako argumenty lub wartości tych funkcji.

Przykładowa specyfikacja w pliku *prog.x*:

```
const MAX = 256;

struct record {
    string name<MAX>;
    int val;
    char initial;
};

program PROG {
    version PRVERS {
        record GETNAME(string) = 1;
        record GETPHONE(int) = 2;
        int ADDRECORD(record) = 3;
    } = 1;
} = 0x200000001;
```

Po przetworzeniu tego pliku programem `rpcgen`, uzyskujemy pliki:

- *prog.h* – plik nagłówkowy, zawierający dyrektywy `#define` dla stałych takich jak `MAX`, a także prototypy funkcji i kadłubków.
- *prog_clnt.c* – funkcje-kadłubki klienta.

W przykładzie z poprzedniego slajdu będą to funkcje:

```
record * getname_1(char **argp, CLIENT *clnt);
record * getphone_1(int *argp, CLIENT *clnt);
int * addrecord_1(record *argp, CLIENT *clnt);
```

- *prog_svc.c* – szkielet aplikacji serwera, zawierający funkcję *main()* funkcję „sortującą” wywołania przychodzące z sieci na konkretne funkcje-kadłubki serwera.
- *prog_server.c* – kadłubki funkcji serwera, dekodujące dane wejściowe i wywołujące właściwe funkcje przetwarzające te dane. Na przykład:

```
record * getname_1(char **argp, struct svc_req rqstp)
{
    static record result;

    /*
     * insert server code here
     */

    return (&result);
}
```

- *prog_xdr.c* – funkcje konwersji typów danych. Przykładowo:

```
bool_t xdr_record(register XDR *xdrs, record *objp)
{
    if (!xdr_string(xdrs, &objp->name, MAX))
        return (FALSE);
    if (!xdr_int(xdrs, &objp->val))
        return (FALSE);
    if (!xdr_char(xdrs, &objp->initial))
        return (FALSE);
    return (TRUE);
}
```

- *makefile.prog*, pozwalający skompilować całość po wywołaniu komendy `make`.

Standard XDR

Różne typy danych są prezentowane w różnych architekturach systemów w odmienny sposób:

- Typ „integer”, czyli `int`:
 - PC/MSDOS: 2 bajty, LSB (little-endian)
 - PC/UNIX: 4 bajty, LSB (little-endian)
 - Sparc/UNIX: 4 bajty, MSB (big-endian)
 - Alpha/UNIX: 8 bajtów, MSB (big-endian, 64-bitowy procesor)
- Typ wskaźnikowy,
- Tablice i struktury.

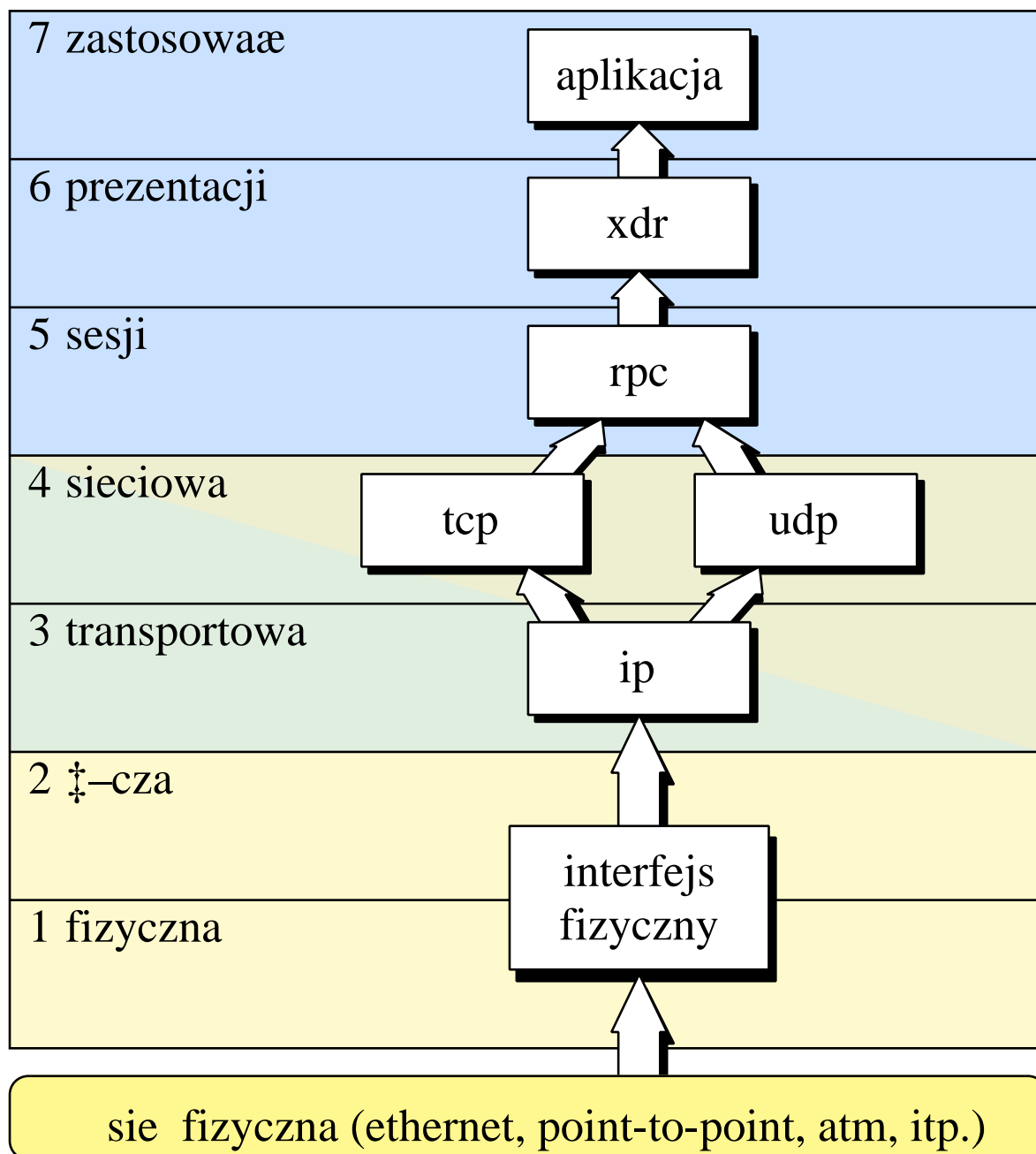
Potrzebne są funkcje konwersji!

- 1. „każdy z każdym”
 - jak zachowa się program po wprowadzeniu na rynek nowego procesora?
 - jaki ma to wpływ na wielkość programu?
 - na efektywność działania?
- 2. jeden wspólny standard i zestaw procedur konwersji dla wszystkich typów lokalnych
 - skalowalność?
 - wielkość programu?
 - efektywność działania?
- wspólny standard: XDR – eXternal Data Representation

Funkcje XDR

- Standard pozwalający przenosić dane różnych typów pomiędzy komputerami o różnych architekturach wewnętrznych.
- Funkcje xdr dokonują *serializacji* lub *deserializacji* danych, czyli przekształcenia struktur, z których korzysta program, na płaski strumień danych, który może być przesłany przez sieć, a następnie odтворzenie z niego odpowiadających mu struktur wyjściowych.
- W bibliotekach xdr znajdują się funkcje konwersji dla wszystkich typów podstawowych, takich jak `char`, `int`, `long`, `short`, `unsigned int` (`u_int`) i pozostałe, zmiennoprzecinkowe typy `float` i `double`, a także typ `void` i `enum`.
- Dodatkowo, w bibliotekach znajdują się funkcje pozwalające budować z typów podstawowych inne typy, takie jak unie, tablice, wskaźniki i odwołania, a także łańcuchy tekstowe i ciągi bajtów.
- Funkcje przetwarzające struktury generowane są przez `rpcgen` ze specyfikacji aplikacji, a ich działanie polega na wywołaniu po kolei odpowiednich funkcji XDR dla wszystkich elementów struktury.

RPC w modelu OSI/ISO



Portmapper

- W typowej komunikacji TCP/UDP serwery są identyfikowane przez numer portu, czytany z pliku */etc/services* lub uzyskiwany za pomocą usług takich, jak NIS.
- W RPC programy identyfikowane są na poziomie symbolicznym, przez numer programu i numer wersji programu. Dzięki podziałowi na wersje mogą współistnieć starsze i nowsze implementacje różnych protokołów sieciowych, np. istotnie się od siebie różniące NFS v2 i NFS v3.
- Tłumaczeniem nazw/numerów RPC na numery portów TCP/UDP zajmuje się program/usługa **portmap**.
- W implementacjach RPC niezależnych od warstwy transportowej (pozwalających korzystać nie tylko z TCP/UDP), abstrakcyjne adresy (nazwy/numery programów) są tłumaczone na parametry potrzebne warstwie transportowej przez usługę/program **rpcbind**.
- Zarówno **portmap** jak i **rpcbind** muszą zostać zlokalizowane z pominięciem mechanizmów RPC – obie te usługi korzystają z portu numer 111.

Komunikacja RPC pomiędzy klientem a serwerem odbywa się w następujący sposób:

- Serwer RPC po uruchomieniu łączy się z portmapperem i rejestruje w nim swój adres (w warstwie transportowej) powiązany z numerem programu i wersji. Służy do tego funkcja *registerrpc()*.
- Klient RPC, chcąc uzyskać połączenie z serwerem, łączy się z portmapperem i dowiadyuje, pod jakim adresem znajduje się żądany program.
- Po uzyskaniu adresu serwera, klient nawiązuje połączenie z serwerem i wysyła żądanie wywołania funkcji, a serwer zwraca wyniki. Do wywołania zdalnej procedury potrzebna jest po stronie klienta tylko jedna funkcja – *callrpc()*.

Indeks

accept(), 9-1–9-4, 9-7, 10-1, 11-2
bind(), 8-7, 8-9, 8-10, 9-3, 9-7, 10-1, 10-2
callrpc(), 12-7
close(), 3-1, 8-7
connect(), 9-1, 9-3, 10-1, 10-3, 10-4
dup(), 3-5
dup2(), 3-5
exec(), 2-4, 2-6, 2-7, 3-5, 6-6, 9-7
execl(), 2-6
execle(), 2-6
execlp(), 2-6
execv(), 2-6
execve(), 2-6
execvp(), 2-6
exit(), 2-7, 2-9, 5-8, 6-6
fcntl(), 2-6, 3-2, 3-3, 10-3, 11-2
fork(), 2-1, 2-4, 2-5, 6-6, 7-13, 9-2, 9-7
fprintf(), 3-1
fread(), 6-8
fseek(), 6-8
ftok(), 4-9
fwrite(), 6-8
gethostbyaddr(), 8-12
gethostbyaddr_r(), 8-12
gethostbyname(), 8-12
gethostbyname_r(), 8-12
gethostent_r(), 8-12
gethostent_r(), 8-12
getpid(), 2-4
getppid(), 2-4
getservbyname(), 8-3, 8-4
getservbyname_r(), 8-4
getservbyport(), 8-3, 8-4
getservbyport_r(), 8-4
getservent(), 8-3, 8-4
getservent_r(), 8-4
htonl(), 7-12, 8-11
htons(), 7-12
ioctl(), 3-2, 11-2
kill(), 2-4
listen(), 9-1, 9-3, 9-7
main(), 2-6, 12-3
mknod(), 3-7
mmap(), 6-8
msgctl(), 4-5
msgget(), 4-4
msgrcv(), 4-7
msync(), 6-8
munmap(), 6-8
ntohl(), 7-12, 8-11
ntohs(), 7-12
open(), 3-7
pclose(), 3-6
pipe(), 3-1, 8-8
poll(), 9-4, 11-2
popen(), 3-6

<i>printf()</i> , 1-5	<i>unlink()</i> , 3-7
<i>read()</i> , 3-1–3-3, 3-7, 8-7, 11-1, 11-2	<i>wait()</i> , 2-4, 2-9, 3-6, 9-2, 9-5
<i>readmsg()</i> , 11-1	<i>write()</i> , 3-1, 3-2, 3-7, 8-7, 11-1
<i>readv()</i> , 11-1	<i>writew()</i> , 11-1
<i>recv()</i> , 10-3, 10-4	<i>Zajmij()</i> , 5-3, 5-4
<i>recvfrom()</i> , 10-1–10-4, 11-1	<i>Zwolnij()</i> , 5-3, 5-4
<i>registerrpc()</i> , 12-7	
<i>rresvport()</i> , 8-10	
<i>sbrk()</i> , 6-6	
<i>select()</i> , 3-3, 9-4, 9-7, 10-3, 11-2–11-4	
<i>semctl()</i> , 5-7	
<i>semget()</i> , 5-6	
<i>semop()</i> , 5-11	
<i>send()</i> , 10-3, 10-4	
<i>sendmsg()</i> , 11-1	
<i>sendto()</i> , 10-1, 10-3, 10-4, 11-1	
<i>setgid()</i> , 9-7	
<i>setpgrp()</i> , 2-9	
<i>setuid()</i> , 9-7	
<i>shmat()</i> , 6-5	
<i>shmctl()</i> , 6-4, 6-6	
<i>shmdt()</i> , 6-5, 6-6	
<i>shmget()</i> , 6-3, 6-6	
<i>shutdown()</i> , 9-4	
<i>sigaction()</i> , 4-7	
<i>sighold()</i> , 2-8	
<i>sigignore()</i> , 2-8	
<i>signal()</i> , 2-8, 5-11	
<i>sigpause()</i> , 2-8	
<i>sigrelse()</i> , 2-8	
<i>sigset()</i> , 2-8	
<i>socket()</i> , 8-7, 8-8	
<i>socketpair()</i> , 8-7, 8-8	

Spis treści

Podstawy systemu UNIX	1-1
System UNIX	1-1
System plików	1-3
Zarządzanie pamięcią	1-4
Biblioteki	1-5
Użytkownicy	1-6
Prawa dostępu	1-7
Wejście/wyjście	1-9
Interpreter poleceń – shell	1-9
Środowisko programistyczne	1-10
 Procesy	 2-1
Kontekst procesu	2-2
Sterowanie procesami z powłoki użytkownika	2-3
Tworzenie nowych procesów	2-4
Sygnały	2-7
Grupy procesów	2-9
Kończenie procesu	2-9
 Strumienie pipe i FIFO	 3-1
Strumienie pipe	3-1
Pisanie do strumienia pipe	3-2
Czytanie ze strumienia pipe	3-3

Łączenie dwóch procesów strumieniem pipe	3-4
Tworzenie strumieni stdout-stdin	3-5
Funkcja <i>popen()</i>	3-6
Strumienie FIFO	3-7
Korzystanie ze strumieni FIFO	4-1
Kolejki komunikatów	4-1
Struktura kolejki w jądrze systemu	4-3
Tworzenie kolejek	4-4
Usuwanie kolejek	4-5
Wysyłanie komunikatów	4-6
Odbieranie komunikatów	4-7
Selektywne przesyłanie komunikatów	4-8
Funkcja <i>ftok()</i>	4-9
Semaforey	5-1
Sekcja krytyczna	5-1
Problem pięciu filozofów	5-5
Tworzenie semaforów	5-6
Usuwanie semaforów	5-7
Zajmowanie i zwalnianie semaforów	5-8
Zajęcie semafora	5-9
Zwolnienie semafora	5-10
Oczekiwanie na zajęcie semafora przez inny proces	5-11

Pamięć wspólna	6-1
Stronicowanie pamięci	6-1
Segmentacja pamięci	6-2
Tworzenie segmentu pamięci wspólnej	6-3
Usuwanie pamięci wspólnej	6-4
Korzystanie z pamięci wspólnej	6-5
Zarządzanie segmentami pamięci przez system	6-6
Pamięć wspólna widziana przez 2 procesy	6-7
Wspólny dostęp do plików – <i>mmap()</i>	6-8
Pamięć wirtualna	6-9
Pełny diagram stanów procesów	6-10
 Komunikacja sieciowa	 7-1
Dostarczanie pakietów:	7-2
Dostęp do medium:	7-3
Skalowalność:	7-3
Model OSI/ISO	7-4
Sieć ethernet – warstwa 2	7-4
Protokoły sieciowe IP – warstwa 3	7-5
TCP/UDP – warstwa 4	7-5
Przykładowy pakiet	7-6
Adresy w sieciach IP	7-7
Maski sieciowe i routing	7-8
Odwzorowanie adresów warstw 2 i 3	7-9

Abstrakcyjny model komunikacji sieciowej	7-10
UDP	7-10
TCP	7-10
Identyfikacja połączeń	7-11
Problem kolejności bajtów	7-12
Model komunikacji klient–serwer	7-13
Uproszczony model OSI	7-13
Protokoły sieciowe	8-1
Powiązania między usługami a numerami portów	8-3
Tłumaczenie nazw usług w systemie UNIX	8-4
Przykład protokołu – SMTP	8-5
Tworzenie gniazdek	8-7
Funkcja <i>socket()</i>	8-8
Funkcja <i>socketpair()</i>	8-8
Nadanie nazwy – funkcja <i>bind()</i>	8-9
Usługi tłumaczenia nazw – DNS/NIS/NIS+	8-11
Komunikacja TCP	9-1
Nawiązywanie połączeń TCP	9-1
Serwery współbieżne	9-2
Funkcja <i>connect()</i>	9-3
Funkcja <i>listen()</i>	9-3
Funkcja <i>accept()</i>	9-4

Funkcja <i>shutdown()</i>	9-4
Serwery samodzielne i uruchamiane przez <i>inetd</i>	9-5
Serwer <i>inetd</i>	9-6
Komunikacja UDP	10-1
Funkcje <i>recvfrom()</i> i <i>sendto()</i>	10-3
Korzystanie z <i>connect()</i>	10-4
Przykład serwera protokołu TCP	10-5
Przykład klienta protokołu TCP	10-6
Przykład serwera protokołu UDP	10-7
Przykład klienta protokołu UDP	10-8
Zaawansowane zagadnienia dotyczące gniazd	11-1
Funkcje wysyłania i odbierania danych	11-1
Zwielokrotne wejście/wyjście	11-2
Funkcja <i>select()</i>	11-3
Funkcje RPC	12-1
Standard XDR	12-4
Funkcje XDR	12-5
RPC w modelu OSI/ISO	12-6
Portmapper	12-7
Indeks	13-1
Spis treści	14-1

<i>Sieciowe Systemy Operacyjne – UNIX</i>	14-5
---	------