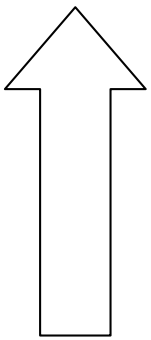
Rodzaje pamięci w systemie komputerowym

- *hierarchia pamięci*
  - rej. procesora (np. AX)
  - pamięć podręczna p. operacyjnej; [ang. cache]
  - pamięć operacyjna
  - pamięć podręczna p. dyskowej; [ang. disk cache]
  - dysk magnetyczny
  - dysk optyczny
  - taśmy magnetyczne

niedostępna dla s.o.

obszar pam. oper.; inaczej: *podręczna pamięć buforowa*

pamięć pomocnicza (ang. secondary storage)



- szybszy dostęp
- bardziej bezpośredni dostęp
- mniejszy rozmiar
- większy koszt 1 bitu
- większa ulotność

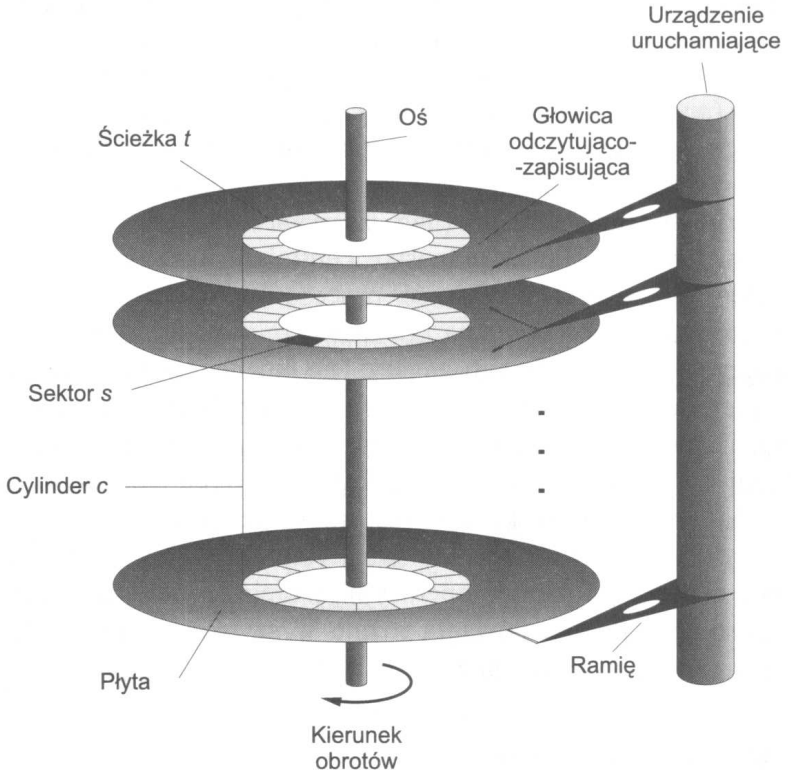
Rodzaje pamięci w systemie komputerowym

- dlaczego w sys. komp. używa się tylu rodzajów pamięci ?
  - potrzebujemy pamięci:
    dużej + szybkiej + taniej
  - to jest sprzeczność !
  - sprzeczność tę rozwiązuje się przez stosowanie **hierarchii pamięci**, w której pamięć *mała ale szybka* służy jako pamięć podręczna pamięci *dużej ale wolnej* ...
  - dlaczego to działa ? odp: "locality of reference"

Dysk magnetyczny

- sektor = najmniejszy blok jaki można odczytać "za jednym zamachem"
- cylinder/ ścieżka = okrąg na którym leżą sektory
- głowice
- adres fizyczny sektora:
  - nr głowicy
  - nr cylindra
  - nr sektora
- adres logiczny sektora:
  - kolejne sektory są ponumerowane
  - procedura BIOSu która pozwala odczytać/ zapisać sektor o podanym adr logicznym
- szybkość dysku magnetycznego
  - tempo przesyłania
    (ile "bajtów na sekundę"; zazw. kilka MB na sekundę)
  - czas ustalania pozycji głowicy
    - czas przesuwania głowicy do odp. cylindra
    - opóźnienie obrotowe

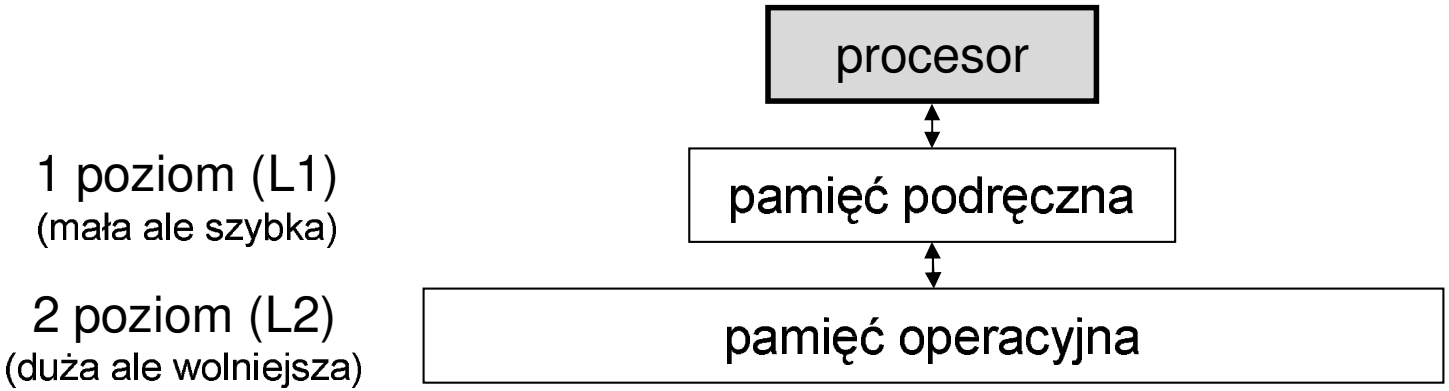
    (zazw. kilka milisekund)



Rys. 2.5 Mechanizm dysku z ruchomymi głowicami

Pamięć podręczna [ang. cache]

- pamięć podręczna (pamięci operacyjnej) pośredniczy między procesorem a pamięcią operacyjną
- przyspiesza czas dostępu do pamięci operacyjnej
- s.o. nie ma na nią wpływu (jest obsługiwana sprzętowo w 100%)



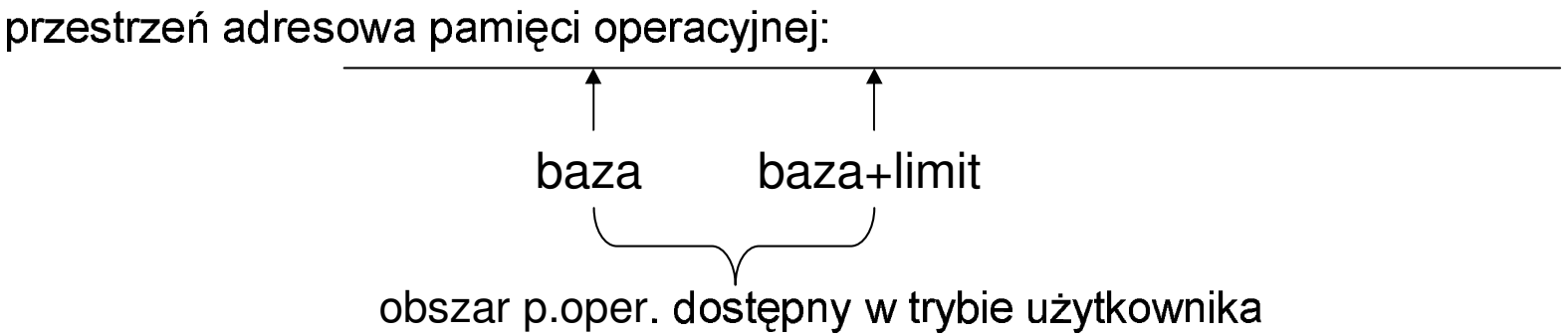
- pamięć podręczna zawiera mały fragment pamięci operacyjnej
  - zamiast odczytywać komórkę z pamięci operacyjnej odczytuje się odpowiednią komórkę z pamięci podręcznej
  - podobnie z zapisem komórek ...
  - pamięć operacyjna i podręczna są podzielone na bloki tego samego rozmiaru; blok p. podr. ma przyporządkowany blok p. oper.; problem pojawia się gdy wszystkie bloki p. podr. są zajęte i trzeba zwolnić jeden z nich ... (tzw *algorytm zastępowania stron*)
- pamięć operacyjna może być pamięcią podręczną pamięci pomocniczej

Ochrona sprzętowa

- sprzęt powinien mieć środki zapewniające że proces 1 nie będzie mógł zakłócać pracy procesu 2 ani pracy systemu operacyjnego ...
  (coś czego nie posiada procesor 8086)
- co złego może zrobić proces ?
  - modyfikować niedozwolone obszary pamięci operacyjnej
  - niewłaściwie używać urządzeń we/wy (np. dysku)
  - przywłaszczyć sobie procesor i nie chcieć go oddać (nieskończona pętla)
- środki ochrony sprzętowej: dualny tryb operacji procesora
  - istnieje "bit trybu"
    - bit trybu = 1 – tryb użytkownika
    - bit trybu = 0 – tryb monitora = tryb jądra = tryb systemu = tryb uprzywilejowany
  - niektóre rozkazy procesora są "uprzywilejowane", tj mogą być wykonywane tylko w trybie jądra (a nie w trybie użytkownika !)
  - gdy procesor wykonuje proces użytkownika to znajduje się w trybie użytkownika
  - gdy wystąpi przerwanie, sprzętowe lub programowe, to procesor przełącza się do trybu jądra (robi to sprzęt !)
  - zanim procesor zacznie kontynuować wykonywanie procesu użytkownika jest przywracany tryb użytkownika

Ochrona sprzętowa c.d.

- ochrona we/wy
  - rozkazy procesora do "rozmawiania" z urządzeniami we/wy (8086: in, out) są uprzywilejowane czyli mogą być wykonywane tylko w trybie jądra
  - gdy proces chce porozmawiać z urz. we/wy musi poprosić o to s.o. tj wywołać funkcję systemową
  - fun. sys. wywołuje się przy pomocy przerwań programowych (8086: int) tak więc tryb przełącza się na tryb jądra !!!
- ochrona pamięci operacyjnej
  - istnieją rejestry: bazowy (ang. base) i graniczny (ang. limit)



  - kod w trybie użytkownika ma dostęp tylko do zakresu pamięci operacyjnej wyznaczonego przez te rejestry
  - modyfikowanie rej. bazowego/ granicznego jest możliwe tylko w trybie jądra

Ochrona sprzętowa c.d.

- ochrona przed "nieskończoną pętlą"
  - problem nieskończonej pętli: proces wykonuje nieskończoną pętle i nie chce dobrowolnie "oddać" procesora innym procesom (np. przez wywołanie funkcji systemowej odczytującej dane z dysku)
  - przy pomocy "zegara" zgłaszającego przerwanie sprzętowe co kwant czasu
  - przerwanie zgłaszane przez zegar powoduje uruchomienie procedury obsługi i przejście do trybu jądra; procedura obsługi w trybie jądra może "przełączyć" procesor na inny proces ...
  - ***dygresja 1***: przełączenie procesora na inny proces następuje zawsze w wyniku wystąpienia przerwania:
    - programowego (wywołanie funkcji systemowej)
    - sprzętowego (np od dysku)
    - sprzętowego/ zegarowego
  - ***dygresja 2***: przerwania zegarowe są niezbędne w s.o. z podziałem czasu !

Usługi s.o.

- usługi programowe
  - uruchamianie programów/procesów
  - operacje we/wy
    (procesy nie mogą rozmawiać bezpośrednio z urządzeniami, potrzebują pośrednictwa s.o. = funkcji systemowych)
  - manipulowanie plikami i katalogami
  - komunikacja między procesami
    ("wspólna pamięć" lub "przesyłanie komunikatów")
  - wykrywanie błędów
    (obsługa wyjątków, np. dostępu do niedozwolonego obszaru pamięci operacyjnej)
- usługi operacyjne
  - przydzielanie zasobów
    (zasobem może być "czas procesora", "pamięć oper.", "pamięć dyskowa", "przewijak taśmy magnetycznej")
  - rozliczanie
    (zapamiętywanie którzy użytkownicy i w jakim stopniu korzystają z zasobów komputera; np. można śledzić ile pamięci dyskowej używają użytko wnicy i wprowadzic ograniczenia - Unix/ quota)
  - ochrona
    (kontrola dostępu do zasobów; np. logowanie gdy rozpoczynamy pracę z Unix-em)

Funkcje systemowe Unix-a

Funkcje systemowe Unix-a

- czyli Unix z perspektywy programisty (język C)
- usługi s.o. udostępniane są programom poprzez zbiór funkcji systemowych
- fun. sys. mogą być uruchamiane specjalnym rozkazem procesora; zazwyczaj jest to rozkaz generujący przerwanie programowe (8086: int)
- w Unix-ie istnieje biblioteka standardowa języka C zawiera funkcje biblioteczne zaimplementowane jako pojedyncze wywołanie fun. sys.
  (i właśnie te funkcje biblioteczne będziemy nazywać fun. sys. !!!)
  przykłady:
  read() – funkcja systemowa
  printf() – "zwykła" funkcja biblioteczna
- podział fun. sys. Unix-a na dotyczące:
  - procesów: fork(), wait(), exit(), exec*()
  - IPC: pipe()
  - plików: open(), read(), write(), close(), dup2(), unlink()
- opis funkcji systemowych Unix-a:
  http://main.amu.edu.pl/~mhanckow
  *** Materiały do ćwiczeń SOP121 | Temat D ***

Obsługa procesów;
fun. sys. Unix-a: opis fork()

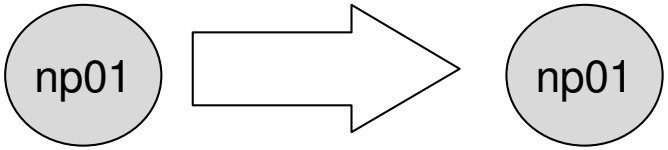
int fork ( );

- proces, który wywołał fork() "rozdwaja się" - mamy teraz proces *macierzysty* i proces *potomny*, który początkowo jest dokładną kopią procesu macierzystego
  - w procesie potomnym fork() zwraca 0
  - w procesie macierzystym fork() zwraca PID procesu potomnego
- proces potomny ma taką samą tablicę deskryptorów jak proces macierzysty (czyli pliki otwarte w procesie macierzystym pozostają otwarte w procesie potomnym !)
- proces potomny tak samo obsługuje sygnały, jak proces macierzysty

Fun. sys. Unix-a; opis fork()

- procesy; fun. sys. fork()

```
// plik źródłowy:
//   np01.cc
// kompilacja:
//   gcc np01.cc -o np01

#include <unistd.h>
...

main()
{   int p;

    p= fork();
    if( p == 0 ) {
       // pp
       sleep(10);
       exit(123);
    } else {
       // pm
       sleep(15);
       exit(0);
    }
}
```



pid=100
15 sek.

pid=101
ppid=100
10 sek.

"pp" (=proces potomny)
i "pm" (=proces macierzysty)
działają równocześnie
(czyli współbieżnie)

Fun. sys. Unix-a; opis wait()

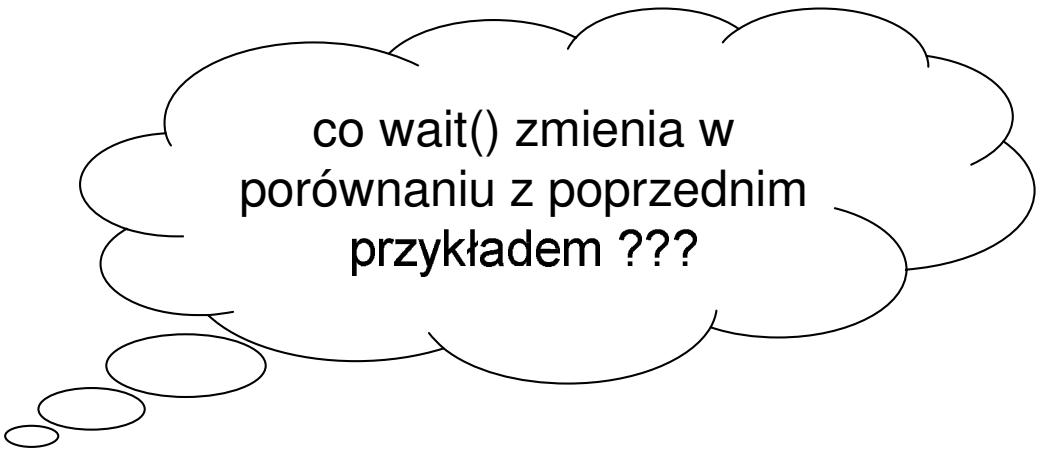
int wait (int *status);

- wait() powoduje, że proces macierzysty czeka na zakończenie (dowolnego) procesu potomnego
- wartością zwracaną przez wait() jest PID procesu potomnego, który się zakończył
- w zmiennej "status" jest zwracana informacja o sposobie zakończenia działania potomka;
- niech: status == HHLL (4 cyfry hex)
  - potomek zakończył się przez wywołanie "exit(y)"; wtedy HH=y, LL=0
  - potomek zakończył się z powodu sygnału; wtedy HH=0, 7-my bit LL zawiera 1 jeśli wygenerowano plik "core", bity 6-0 LL zawierają nr sygnału
- Uwaga: fun. sys. ***exit(kod_zakończenia)*** powoduje zakończenie procesu oraz określa jego kod zakończenia

Fun. sys. Unix-a; opis wait()

- procesy; fun. sys. fork() + wait()

```
// plik np01a.cc

#include <unistd.h>
...

main()
{    int p;

     p= fork();
     if( p == 0 ) {
        // pp
        sleep(10);
        exit(123);
     } else {
        // pm
        int p, status;
        p= wait(&status);
        sleep(15);
        exit(0);
     }
}
```



(pokazać przykład fork01.cc !!!)

Fun. sys. Unix-a; opis exec*()

int exec*(...)

- `int execl  (char *path, char *arg, ...);`
- `int execv  (char *path, char *argv[]);`
- `int execle (char *path, char *arg, ..., char *envp[]);`
- `int execve (char *path, char *argv[], char *envp[]);`
- `int execlp (char *file, char *arg, ...);`
- `int execvp (char *file, char *argv[]);`

- funkcja "exec()" zastępuje kod bieżącego procesu nowym kodem programu
- nowy program dziedziczy po procesie, który wywołał "exec" wiele atrybutów (np otwarte deskryptory plików !)
- pierwszy parametr funkcji "exec*()" identyfikuje plik z nowym programem

Fun. sys. Unix-a; opis exec*()

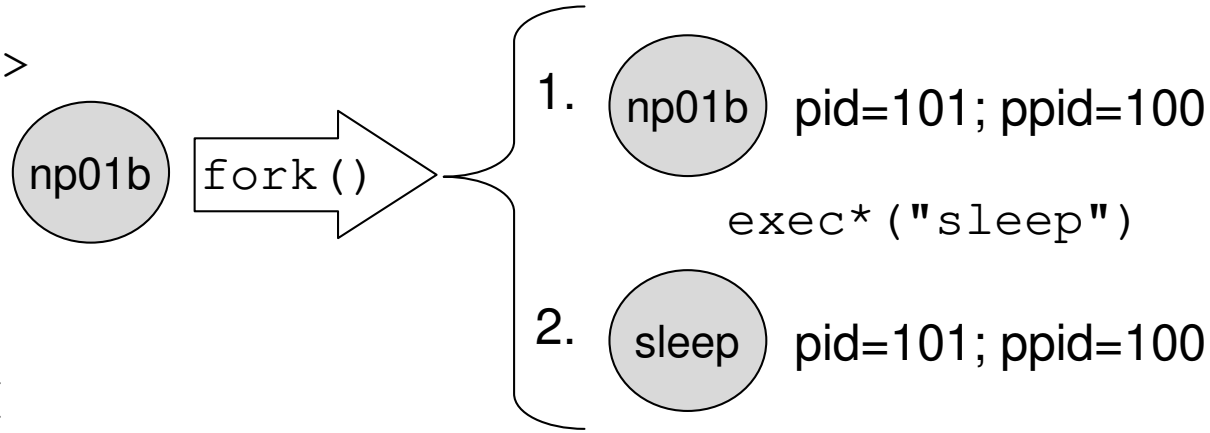
- istnieją różne sposoby przekazywania "parametrów" i "środowiska" do nowego programu; literki w nazwie funkcji "exec*()" maja następujące znaczenie :
  - "v" oznacza, ze parametry są przekazywanie przez tablice wskazań
    ```
    char *v[]={"ls", "-l", NULL};
    execv("/bin/ls", v);
    ```
  - "l" oznacza, ze parametry są przekazywanie przez listę
    ```
    execl("/bin/ls", "ls", "-l", NULL);
    ```
  - "p" nie potrzeba podawać pełnej ścieżki do pliku z programem; będzie wykorzystywana zmienna PATH !!!
    ```
    execlp("ls", "ls", "-l", NULL);
    ```
  - "e" oznacza, ze podajemy nowe środowisko za pomocą tablicy wskazań do napisów "zmienna=wartosc".
    ```
    char *v[]={"zm1=wart1", "zm2=wart2", NULL};
    execle("./prog", "prog", "par1", "par2", NULL, v);
    ```
  - UWAGA: jeśli nie ma literki "e" to środowisko pozostaje takie jakie było przed wykonaniem exec*() ! (tablica "environ")
- "***Parametry***" i "***środowisko (=zmienne środowiska)***" są dostępne w programie napisanym w języku "C" przy pomocy parametrów "argv" i "env" w fun. main():

```
main (int argc, char *argv[], char *env[])
{  int i; char **c;
   // wyswietlanie argumentow:
   for(i=0; i<argc; i++) printf("%s\n", argv[i]);
   // wyświetlanie środowiska:
   c=env; while(*c) printf("%s\n", *c++);
   // wyświetlanie środowiska 2:
   c=environ; while(*c) printf("%s\n", *c++);
}
```

Fun. sys. Unix-a; opis exec*()

- procesy; uruchamianie programów; fork() + wait() + exec*()

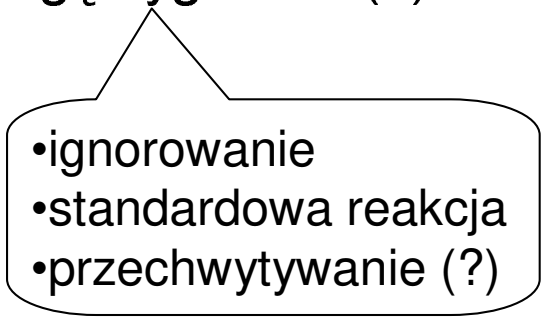
```
// plik np01b.cc

#include <unistd.h>
...

main()
{    int p;

     p= fork();
     if( p == 0 ) {
        // pp
        write(1,"to ja pp ...\n", 13);
        printf("to ja pp\n");
        execl("/bin/sleep", "sleep", "100", NULL);
     } else {
        // pm
        int p, status;
        p= wait(&status);
        printf("zaraz sie zakoncze; pm\n");
        exit(0);
     }
}
```



(pokazać przykład fork02.cc !!!)

Fun. sys. Unix-a; fork() i exec*() c.d.

- co proces potomny dziedziczy po macierzystym:
  - `fork()`
  - `exec*()`
    - otwarte pliki
    - zmienne środowiska
    - bieżący katalog
    - kod programu (`exec*(kod_programu)`)
    - obsługę sygnałów (?)



Operacje na plikach;

fun. sys. Unix-a: opis open(), read(), write(), close()

int open(char *path, int oflag [, mode_t mode]);

- służy do otwierania pliku o nazwie podanej przez "path"; zwraca *deskryptor* udostępniający plik
- bity parametru "oflag" pozwalają określić następujące rzeczy:
  - File Acces Flag (tylko jeden bit ustawiony)
    - O_RDONLY - The file is open for reading only.
    - O_WRONLY - The file is open for writing only.
    - O_RDWR - The file is open for reading and writing.
  - File Status Flag
    - specjalne przetwarzanie przy otwieraniu pliku
      - O_CREAT - jeśli plik nie istniał to zostanie utworzony
      - O_EXCL - razem z O_CREAT kończy się błędem, jeśli plik istnieje
      - O_TRUNC - jeśli plik istnieje to zostanie "skrócony do zera"
    - początkowy stan otwartego pliku
      - O_APPEND - bieżąca pozycja pliku jest ustawiana na końcu pliku przed każdym zapisem
      - O_NONBLOCK, O_NDELAY - fun. sys. dotyczące tego pliku nie będą blokować
- jeśli jest tworzony nowy plik (O_CREAT), to używane jest "mode", zawiera ono prawa dla nowo tworzonego pliku; parametr "mode" można podawać tak jak ósemkowy parametr dla polecenia "chmod" (uwaga na "umask")

Operacje na plikach;

fun. sys. Unix-a: opis open(), read(), write(), close()

int close(int filedes);

- zamyka plik o deskryptorze "filedes"

int read(int filedes, void *buffer, int nbytes);

- próbuje wczytać *nbytes* bajtów z pliku o podanym deskryptorze do podanego bufora
- bieżąca pozycja w pliku przesuwa się o tyle, ile bajtów przeczytano
- read() zwraca ilość bajtów naprawdę przeczytanych (zawracana wartość może być mniejsza od nbytes !)
- gdy "bieżąca pozycja" przekroczy koniec pliku, to read() zwraca 0

int write(int filedes, void *buffer, int nbytes);

- działa podobnie do read(), z ta różnicą, że pisze do pliku zamiast czytać

Operacje na plikach c.d.

- przepisywanie plik1.txt do plik2.txt :

```
#include <stdio.h>
.................

main()
{   int f1,f2; char c;

    // otwieramy pliki:
    f1=open("plik1.txt", 0);
    f2=open("plik2.txt", O_RDWR|O_CREAT|O_TRUNC, 0600);

    while( read(f1, &c, 1)==1 ) // wyjaśnić jak działa ta pętla
    {
       write(f2, &c, 1);
    }

    // zamykamy deskryptory (to nie jest konieczne):
    close(f1); close(f2);
}
```

(uruchomić przykład fork05.cc !!!)

Operacje na plikach c.d.

- jak się realizuje "przeadresowanie", czyli:
  ```
  prog >plik.txt
  ```
- służą do tego celu funkcje systemowe:
  int dup (int filedes);
  int dup2 (int filedes, int new);
- funkcja "dup()" zwraca nowy deskryptor dotyczący pliku dostępnego poprzez podany (otwarty) deskryptor *filedes;* będzie to pierwszy wolny deskryptor w tablicy deskryptorów procesu ...
- ten sam plik będzie dostępny poprzez dwa różne deskryptory; będą one używać tej samej pozycji w tablicy plików, co oznacza że będą miały tę samą bieżącą pozycję !
- funkcja "dup2()" zapewnia, że tym nowym deskryptorem będzie "new"; pozwala ona dokonać p*rzeadresowania* w następujący sposób:

```
int desk=open("plik.txt",O_RDWR|O_CREAT|O_TRUNC,0600);
dup2(desk,1);
close(desk);
write(1,"ABC",3);
   // to jest zapis na "stdout",
   // ale w rzeczywistości wszystko idzie do "plik.txt";
   // "stdout" zostało przeadresowane do pliku "plik.txt"
```

Operacje na plikach c.d.

- realizacja "przeadresowania" przy uruchamianiu programu przy pomocy fun. sys. dup2()

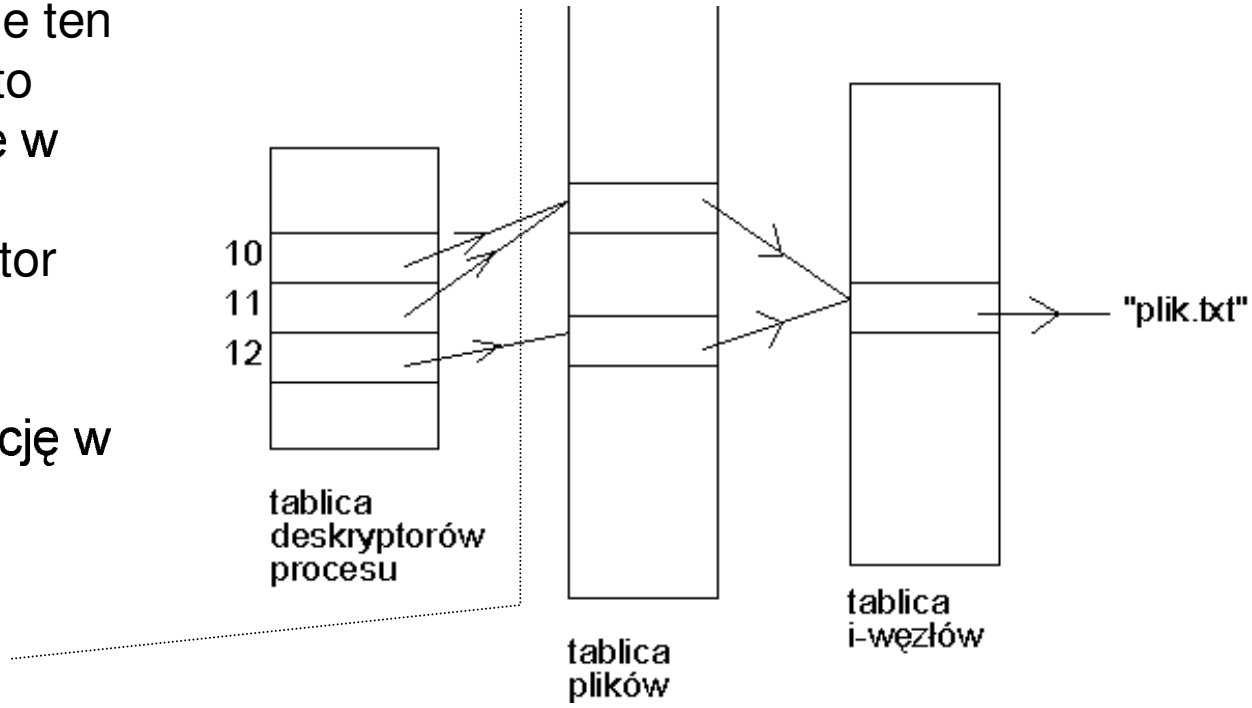
(pokazać przykład fork03.cc !!!)

- ten sam efekt można uzyskać wydając w powłoce polecenie:

```
env > plik.txt
```

Operacje na plikach c.d.

- Czym się różnią deskryptory otrzymane przez open() i dup()/dup2() ?
- Niezbędne pojęcia aby to wyjaśnić:
  - *bieżąca pozycja* w otwartym pliku
  - *tablica deskryptorów procesu* (deskryptory to indeksy elementów tej tablicy)
  - *tablica plików* (tu przechowuje się "bieżącą pozycję" !)
  - *tablica i-węzłów* (każdy element tej tablicy identyfikuje plik)
- Każdy proces ma własną *tablicę deskryptorów*.
- W danej maszynie istnieje jedna *tablica plików* i jedna *tablica i-węzłów.*
- Na poniższym rysunku deskryptory 10,11,12 udostępniają ten sam plik "plik.txt". Jeśli teraz przesuniemy bieżącą pozycję poprzez desk 10 to ma to wpływ na bieżącą pozycje poprzez desk 11 lecz nie poprzez desk 12.
- Gdy otwieramy dwukrotnie ten sam plik funkcją **open()** to otrzymamy różne pozycje w tablicy plików.
- Gdy duplikujemy deskryptor funkcją **dup()/dup2()** lub wykonujemy **fork()** to otrzymamy tę samą pozycję w tablicy plików.

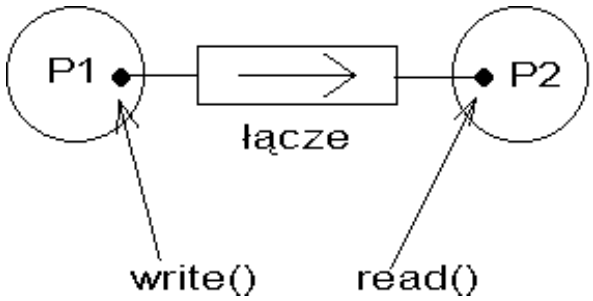


IPC: fun. sys. pipe()

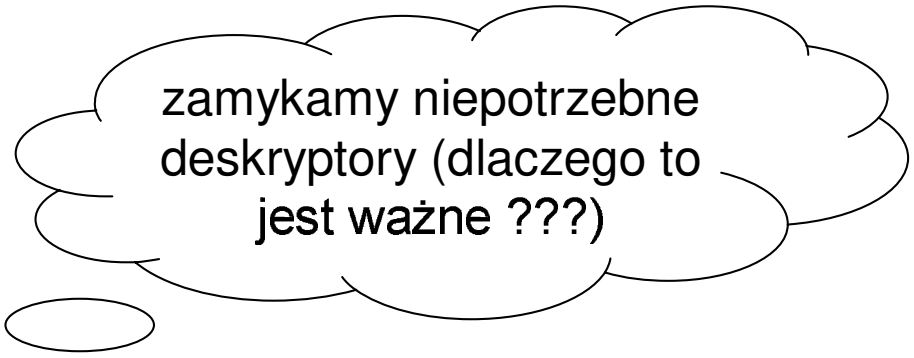
- fun. sys. pipe() służy do tworzenia *łączy* (tych samych które służą do realizacji potoków !)

int pipe (int filedes[2]);

- funkcja ta tworzy łącze i przydziela dwa deskryptory dające dostęp do jego końcówek, które umieszcza w podanej przez parametr tablicy dwóch integer-ów:
  - filedes[0] - końcówka do czytania
  - filedes[1] - końcówka do pisania
- jesli czytamy funkcją "read()" z łącza, ktorego NIKT nie ma otwartego do zapisu, to "read()" zwraca 0 (zupełnie tak, jakby plik się skonczył !).
- jeśli ktoś ma łącze otwarte do zapisu, lecz jest ono PUSTE - to fun. "read()" blokuje, tak długo aż nie pojawią się jakieś znaki. Jeśli w pewnym momencie okaże się, ze nie ma procesów, ktore maja to łącze otwarte do zapisu to fun. "read()" przestanie blokować i zwróci 0.



IPC: fun. sys. pipe()

- jak użyć pipe() do komunikacji między procesami ...

```
#include <stdio.h>
..................

main()
{
   int d[2], stat;
   pipe(d);
   if( fork()!=0 )
   {   // pm
       close(d[0]);
       write(d[1], "ABC\n", 4);
       close(d[1]);
       sleep(10);
       wait(&stat);
   } else {
       // pp
       dup2(d[0], 0);
       close(d[0]); close(d[1]);
       execl("/bin/cat", "cat", NULL);
   }
}
```