Termin Egzaminu (Język C):

>> *CZWARTEK*, 4 LUTEGO <<

GODZ. 11:00 (=>*zdalnie!*)

[*W sesji POPRAWKOWEJ:* 24 lutego, 11:00]

Forma zaliczenia kursu: Egzamin pisemny (test wyboru) * **

* Warunkiem przystąpienia do egzaminu jest ***zaliczenie ćwiczeń***
(w uzasadnionych przypadkach: *zgoda prowadzącego ćwiczenia*)

** **Ocena 5.0 (bdb)** z ćwiczeń *zwalnia z pisemnej części egzaminu*

[OCENA KOŃCOWA: *0.5*ocena z ćwiczeń + 0.5*wynik egzaminu* **]**

Egzamin PRÓBNY:

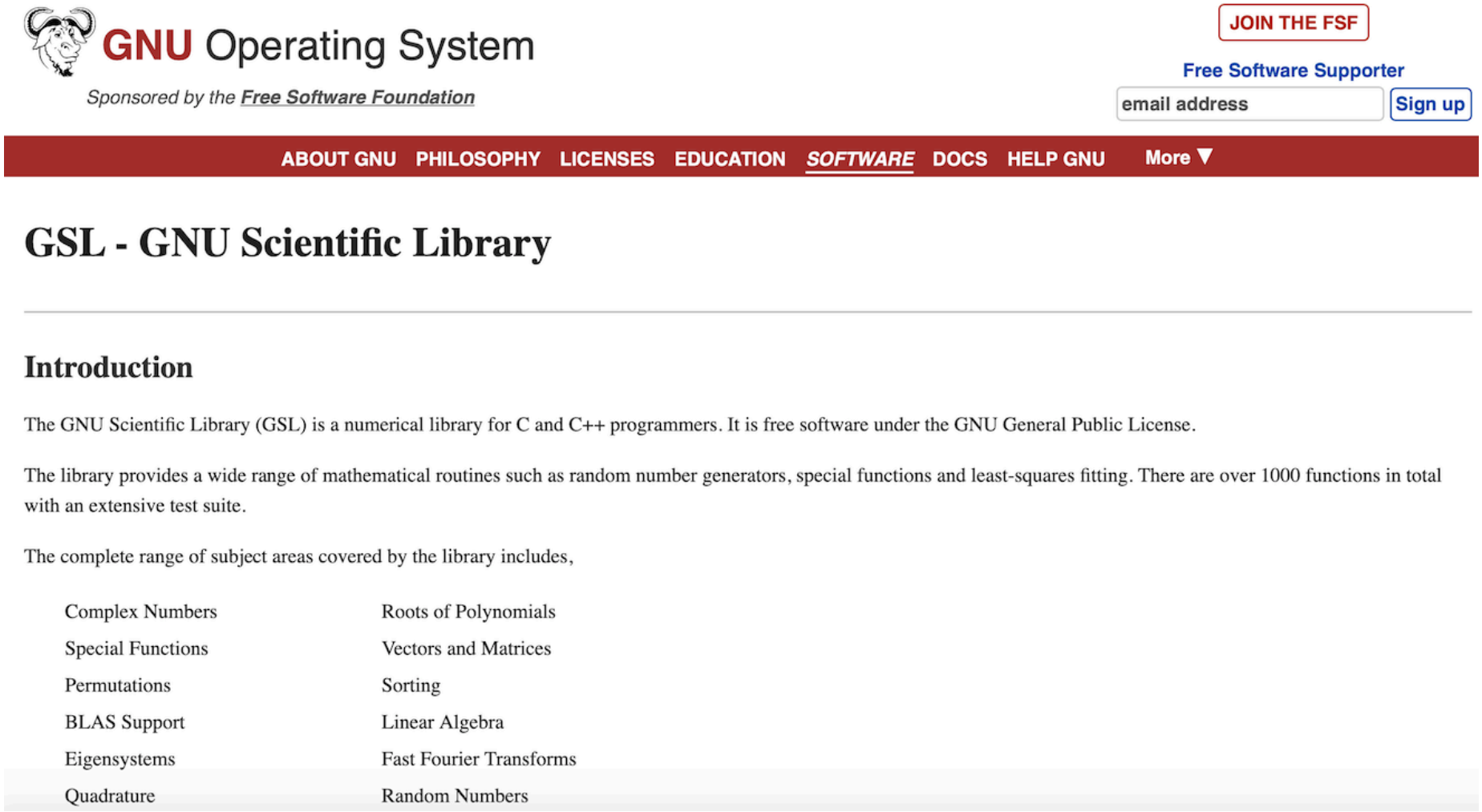
(nieobowiązkowy, bez konsekwencji ...)

>> *WTOREK, 26 stycznia, 12.15* <<

[Pegaz-egzaminy]

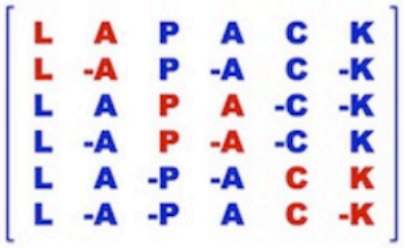
Kiedy potrzebujemy obliczeń numerycznych ...

— **GSL (*G*NU *S*cientific *L*ibrary):** www.gnu.org/software/gsl/



Wydajna numeryczna algebra liniowa …

— LAPACK (*Linear Algebra PACKage*): www.netlib.org/lapack/



Version 3.9.0
LAPACK on GitHub
Browse the LAPACK User Forum
Browse the LAPACK User Forum
Contact the LAPACK team

Get the latest LAPACK News

access

LAPACK is a software package provided by Univ. of Tennessee; Univ. of California, Berkeley; Univ. of Colorado Denver; and NAG Ltd..

Presentation

LAPACK is written in Fortran 90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

Porównanie wydajności GSL i LAPACK-a:

Thursday, June 25, 2009

gsl vs lapack performance

I had some doubts about the LU routines in the gsl library (GNU Scientific Library). See http://yetanothermathprogrammingconsultant.blogspot.com/2009/06/gsl-gnu-scientific-library.html. Here I try a quick experiment by inverting a square nxn matrix. As test matrix I used the Pei matrix (http://portal.acm.org/citation.cfm?id=368975). Here are the results:

library	**gsl**	**lapack**
routine	gsl_linalg_LU_decomp + gsl_linalg_LU_invert	dgesv
compiler	cygwin gcc	Lahey lf95
n=100	0.047 seconds	0.024 seconds
n=1000	6.735 seconds	2.017 seconds
n=2000	1:01 minutes	17.257 seconds

[*http://yetanothermathprogrammingconsultant.blogspot.com/2009/06/gsl-vs-lapack-performance.html*]

Dlaczego pisanie dobrych programów numerycznych nie jest łatwe?

[*DALEJ — omówimy przykład/pułapkę Jamesa H. Wilkinsona: https://en.wikipedia.org/wiki/Wilkinson's_polynomial*]

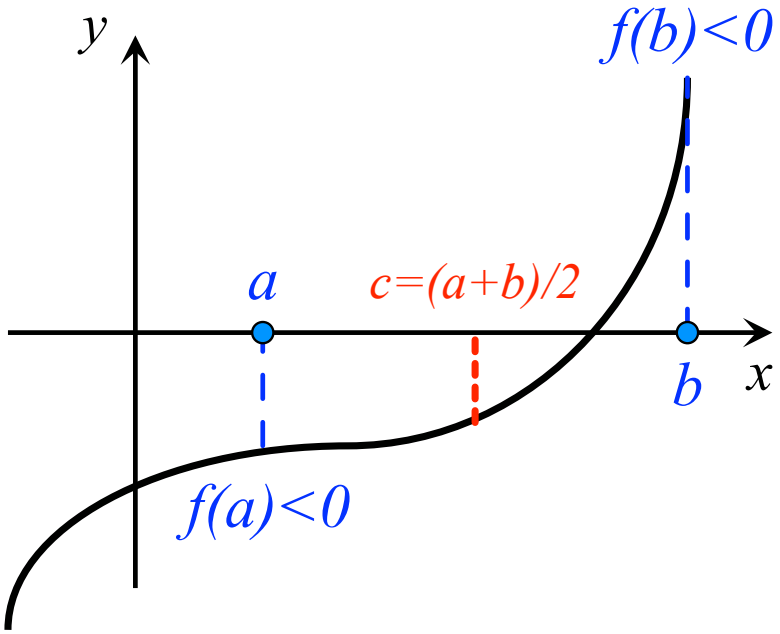
Wprowadzenie: *Jak szukać miejsc zerowych funkcji?*

— Metoda bisekcji (*równego podziału*) [=> *wolna,* ***b. stabilna***]

— Metoda stycznych (*Newtona*) [=> *szybka, ale sprawia kłopoty…*]

— …

Metoda bisekcji



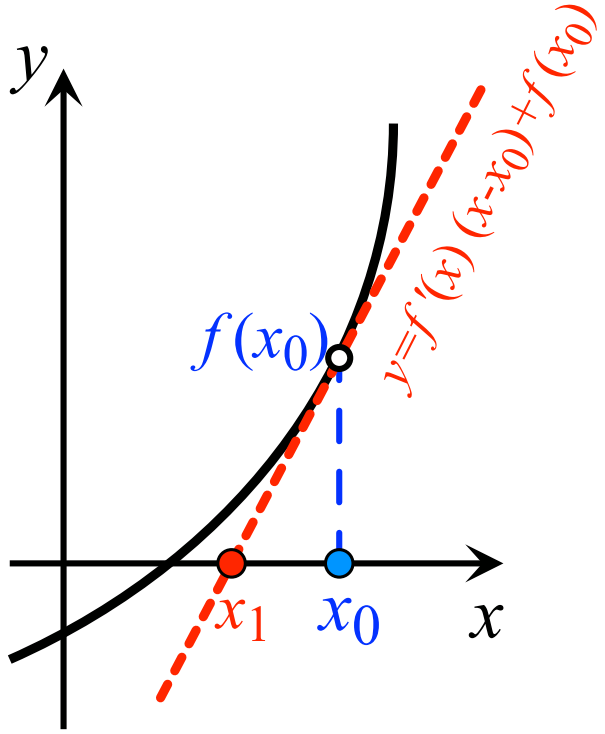
1. Wybieram *a* i *b* t., że *f(a)f(b)<0*
2. Obliczam: *c=(a+b)/2* oraz *f(c)*
3. Jeśli *f(a)f(c)<0*, podstawiam: *b=c*
4. W przeciwnym przypadku: *a=c*
5. Wracam do kroku 2.

W każdym kroku, przedział (a,b) ulega zmniejszeniu o czynnik 2; po n krokach znamy zatem rozwiązanie z dokładnością do $|a-b|\cdot 2^{-n}$.

[=> *~3 nowe cyfry dziesiętne co 10 podziałów*]

Metoda stycznych (m. Newtona)

Jeśli potrafimy obliczać pochodną $f'(x)$ funkcji $f(x)$, możemy postąpić inaczej niż w omówionej wcześniej metodzie bisekcji.



1. Ustalam n=0, wybieram punkt x_0
2. Obliczam: $f(x_n)$ oraz $f'(x_n)$
3. Obliczam: $x_{n+1} = x_n - f(x_n)/f'(x_n)$
4. Zwiększam n o 1; wracam do kroku 2.

Jeśli metoda Newtona jest zbieżna (a zależy to od własności funkcji f…), wynik w arytmetyce podwójnej precyzji

najczęściej zostaje ustalony [`xn==xn+(xn-xn-1)`] *już po kilkunastu krokach. Łatwo jednak wskazać przypadki, kiedy metoda* ***w ogóle nie działa****, zob. https://en.wikipedia.org/wiki/Newton's_method [i przypisy!]*

Wielomian Wilkinsona

Niemal na początku ery komputerów (*w 1963 roku*), J. H. Wilkinson rozważał problem szukania miejsc zerowych wielomianu:

$$p(x) = (x-1)(x-2)(x-3)\cdot\ldots\cdot(x-20) =$$

$$x^{20} - 210\,x^{19} + 20615\,x^{18} - \ldots + 20!$$

[*Znając rozwiązania dokładne, możemy testować algorytmy numeryczne!*]

Wilkinson szukał zer metodą Newtona, na komputerze *Pilot ACE (zaprojektowanym przez A.Turinga)*, z 30 bitową reprezentacją liczb zmiennopozycyjnych. Mantysa liczyła 22 bity, a zatem $\varepsilon = 2^{-23}$ było już liczbą, dla której: $1.0 + \varepsilon == 1.0$.

Okazuje się (*czego Wilkinson początkowo nie był świadomy…*), że dla wielomianu p(x) zmiana współczynnika przy x^{19}, z wartości `-210` na `-210 - 2^-23` sprawia, że część rozwiązań równania $p(x) = 0$ w ogóle znika (*zamienia się w rozwiązania zespolone*), zaś np. $x_{20}=20$ przesuwa się do $x_{20} \approx 20.8$ (**!**)

Ku zaskoczeniu Wilkinsona, algorytm Newtona (*ani żaden inny!*) poszukiwania zer nie mógł zatem na takiej maszynie poprawnie działać …

Więcej: https://en.wikipedia.org/wiki/Wilkinson's_polynomial

Poprzedni wykład [*12. 01. 2021*] :

— ***Struktury rekurencyjne*** (drzewa binarne, jednokierunkowe łańcuchy odsyłaczy)

— Tablice mieszające (*„hashmap-y”*)

— Synonimy typów: **`typedef`**

— Unie (*union*); pola bitowe

Środowisko systemu UNIX

W systemach operacyjnych zgodnych ze standardem **POSIX** (*Portable Operating System Interface*), takich jak Unix, Linux, MacOSX (*i kilku innych...*) implementacje C (oraz C++) zaopatrzone są w nagłówek standardowy `<unistd.h>` zawierający deklaracje szeregu *funkcji rezydujących wewnątrz systemu operacyjnego* (*inaczej:* ***odwołań systemowych***).

Odwołania systemowe często możemy zastąpić funkcjami z biblioteki standardowej (*tak jest np. z omawianymi wcześniej funkcjami plikowymi, zob.* wyklad07); przydają się jednak jeśli chcemy osiągnąć maksymalną wydajność programu, kontrolować buforowanie zapisu/odczytu danych, itp.

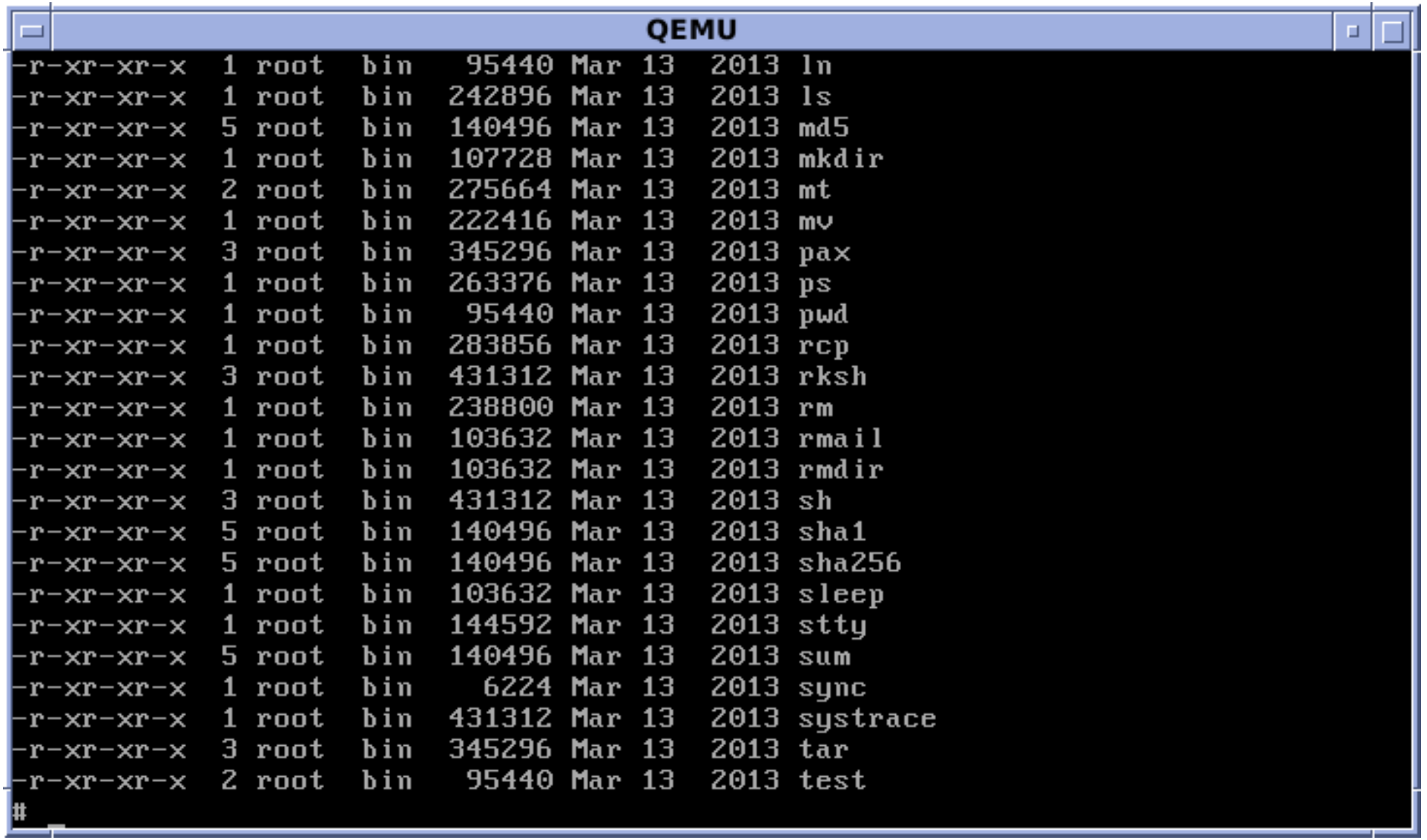
Operacje wejścia/wyjścia

W systemach typu Unix, operacje wejścia/wyjścia wyrażają się poprzez ***czytanie z plików*** lub ***pisanie do plików***. Każde urządzenie zewnętrzne (także *klawiatura i ekran!*) jest powiązane z pewnym plikiem wchodzącym w skład systemu plików.

A zatem, całą komunikację programu z urządzeniami zewnętrznymi obsługuje pewien wspólny, jednorodny aparat.

Otwieranie plików. W ogólnym przypadku, aby czytać z pliku lub do niego pisać, musimy najpierw poinformować system o naszym zamiarze (=> *otworzyć plik*). Jeśli chcemy pisać do pliku, może być konieczne jego utworzenie lub skasowanie dotychczasowej zawartości; system musi zatem sprawdzić, czy mamy do tego prawo [=> *prawa dostępu w systemie Unix*].

$ ls -l


QEMU
-r-xr-xr-x 1 root bin 95440 Mar 13 2013 ln
-r-xr-xr-x 1 root bin 242896 Mar 13 2013 ls
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 md5
-r-xr-xr-x 1 root bin 107728 Mar 13 2013 mkdir
-r-xr-xr-x 2 root bin 275664 Mar 13 2013 mt
-r-xr-xr-x 1 root bin 222416 Mar 13 2013 mv
-r-xr-xr-x 3 root bin 345296 Mar 13 2013 pax
-r-xr-xr-x 1 root bin 263376 Mar 13 2013 ps
-r-xr-xr-x 1 root bin 95440 Mar 13 2013 pwd
-r-xr-xr-x 1 root bin 283856 Mar 13 2013 rcp
-r-xr-xr-x 3 root bin 431312 Mar 13 2013 rksh
-r-xr-xr-x 1 root bin 238800 Mar 13 2013 rm
-r-xr-xr-x 1 root bin 103632 Mar 13 2013 rmail
-r-xr-xr-x 1 root bin 103632 Mar 13 2013 rmdir
-r-xr-xr-x 3 root bin 431312 Mar 13 2013 sh
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 sha1
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 sha256
-r-xr-xr-x 1 root bin 103632 Mar 13 2013 sleep
-r-xr-xr-x 1 root bin 144592 Mar 13 2013 stty
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 sum
-r-xr-xr-x 1 root bin 6224 Mar 13 2013 sync
-r-xr-xr-x 1 root bin 431312 Mar 13 2013 systrace
-r-xr-xr-x 3 root bin 345296 Mar 13 2013 tar
-r-xr-xr-x 2 root bin 95440 Mar 13 2013 test
#

Prawa dostępu

W systemie Unix z każdym plikiem związana jest 9 bitowa liczba całkowita, przechowująca informacje o uprawnieniach do czytania (***r****ead*), pisania (***w****rite*), i wykonywania pliku (*e****x****ecute*), które zdefiniowane są osobno dla *właściciela* pliku, *zespołu* do którego należy, oraz *pozostałych użytkowników* systemu.

Przykładowo, sekwencja znaków: **`rw-rw-rw-`** w lewej kolumnie po wpisaniu komendy `ls -l` oznacza, że wszyscy mają prawo *czytać i pisać* do pliku, nikt natomiast nie może go *wykonać*.

W zapisie ósemkowym, zwyczajowo stosowanym do kodowania praw dostępu w programach, **`rw-rw-rw-`** odpowiada **`0666`**.

[***Dalej****, prawa dostępu pojawią się przy omawianiu funkcji* **`open`** *i* **`creat`**]

Otwieranie plików (c.d.)

Jeśli operacja otwarcia się powiedzie, system przekazuje do programu pewną nieujemną liczbę całkowitą nazywaną ***deskryptorem pliku*** (lub `-1`, jeśli wystąpi błąd).

Wcześniej [***zob.*** *wyklad07*], *działając na poziomie biblioteki standardowej, posługiwaliśmy się* ***wskaźnikami plikowymi*** *(np.* `FILE *f` *). Technicznie, struktura wskazywana przez* `f` *zawiera* ***między innymi*** *deskryptor pliku.*

W praktyce, wiele odwołań dotyczy *klawiatury i ekranu*, stworzono zatem specjalne mechanizmy upraszczające te formy komunikacji.

Unixowy ***interpretator poleceń*** (=> powłoka; ang. *shell*) uruchamiając program otwiera trzy pliki, o deskryptorach 0, 1, i 2, tożsame ze *standardowym wejściem*, *wyjściem*, *i wyjściem błędów*.

Czytając *z pliku o deskryptorze* 0, oraz pisząc do *plików o deskryptorach* 1 *lub* 2, nasz program może zatem wprowadzać dane i wypisywać wyniki ***bez otwierania żadnych plików***.

Dalej, *standardowe wejście/wyjście* możemy przekierować do „prawdziwego” pliku dzięki mechanizmowi potoków:

```
$ ./a.out < input-file > output-file
```

Powłoka (=> *shell*) zmieni wówczas domyśle powiązania deskryptorów 0 oraz 1 i połączy je ze wskazanymi plikami.

[*Zwykle desktyptor 2 pozostaje trwale związany z ekranem, aby tam pojawiały się komunikaty o błędach.*]

W mechanizmie potoków, powiązania plików są zmieniane przez powłokę a nie przez program; program używa zatem deskryptorów (0, 1, 2) nie wiedząc, skąd dane są pobierane i dokąd wysyłane.

Wejście/wyjście niskiego poziomu (`read` i `write`)

Funkcje **`read`** i **`write`** z nagłówka `<unistd.h>` udostępniają odwołania systemowe (o identycznych nazwach) umożliwiające elementarne operacje wejścia i wyjścia.

```
int read(int fd, char *buff, int n);
int write(int fd, char *buff, int n);
```

Pierwszym argumentem każdej z tych funkcji jest *deskryptor pliku*, ***drugim*** — *wskaźnik znakowy* do miejsca w pamięci, w którym przychodzące dane mają przechowane (lub z którego mają być wysłane do pliku), ***trzecim*** — liczba bajtów do przesłania.

Każda z funkcji (**`read`**/**`write`**) zwraca liczbę faktycznie przesłanych bajtów. W szczególności, przy czytaniu liczba bajtów może być mniejsza niż żądana; 0 oznacza koniec pliku, a -1 błąd.

Najczęściej pojawiające się wartości `n` to 1 (wymusza „niebuforowane” przesyłanie danych po 1 bajcie) lub potęgi dwójki, w rodzaju `1024`, `4096` itp, odpowiadające rozmiarowi jednostki alokacji (*bloku pamięci*) w danym systemie plików.

[*Przesyłanie danych w większych paczkach na raz jest na ogół bardziej efektywne, gdyż wymaga mniejszej liczby odwołań do systemu.*]

Prosty program ***kopiujący dane ze standardowego wejścia*** (`fd==0`) ***na standardowe wyjście*** (`fd==1`) może wyglądać np. tak [*zob. też K & R, rozdz. 8*]: …

```c
#include <unistd.h>
#define BUFFSIZE 1024


main()
{
  char buff[BUFFSIZE];
  int n;

  while ((n = read(0, buff, BUFFSIZE))>0)
    write(1, buff, n);

  return 0;
}
```

Za pomocą funkcji **`read`** i **`write`** możemy łatwo zbudować własne wersje funkcji wyższego poziomu (takich jak **`getchar`** i **`putchar`**).

Pokażemy teraz [*za K&R*], jak napisać funkcję **`getchar`** tak, aby czytała dane wejściowe dużymi porcjami (zdefiniowanymi wartością stałej `BUFFSIZE`), ale wyprowadzała zawsze po jednym znaku (lub wartość `EOF`, w przypadku napotkania znacznika końca pliku).

[*Stała **`EOF`** — **E**nd **O**f **F**ile — jest zdefiniowana w nagłówku <**`stdio.h`**>; zwykle jej wartość to -1, nie należy jednak zakładać, że tak jest zawsze.*]

Wymagane zachowanie można łatwo osiągnąć używając zmiennych automatycznych o klasie pamięci **`static`** ...

```c
#include <stdio.h>
#include <unistd.h>
#define BUFFSIZE 1024


#undef getchar

int getchar(void)
{
  static char buff[BUFFSIZE];
  static char *bufp = buff;
  static int n = 0;

  if (n == 0) {  /* pusty bufor */
    n = read(0, buff, sizeof buff);
    bufp = buff;
  }
  return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Dyrektywa **`#undef getchar`** „zakrywa” deklarację funkcji getchar() zawartą w pliku nagłówkowym `<stdio.h>`

Kilka dalszych (nieoczywistych!) szczegółów:

Tablica **`buff`** służąca do przechowywania (*buforowania*) wczytanych znaków musi być typu `char`, ponieważ funkcja **`read`** akceptuje jedynie ***wskaźniki znakowe***.

[*Z drugiej strony: wartość zwracana przez* `getchar()` *jest typu* ***`int`****, aby „zmieścić” EOF …*]

Przed zwróceniem wartości wczytanego znaku (*jeśli nie zwracamy EOF-a*) wykonujemy zatem rzutowanie na typ: `unsigned char` aby uniknąć możliwych problemów z bitem znaku.

Funkcje: `open`, `creat`, `close`, `unlink`

Pliki inne niż *standardowe wejście*, *wyjście*, czy też *wyjście błędów* muszą zostać ***jawnie otwarte*** zanim wykonamy na nich jakiekolwiek operacje.

W *bibliotece standardowej* [**zob.** `wyklad07`] służy do tego funkcja `fopen(...)` zwracająca wskaźnik plikowy.

Działając na poziomie ***odwołań systemowych***, mamy do dyspozycji dwie funkcje: **`open`** i **`creat`** (*„You left out the »E«”…*)

```
int open(char *name, int flags, int perms);
int creat(char *name, int perms);
```

Funkcja **`open`** jest *(nieco…)* podobna **`fopen`**; zamiast wskaźnika plikowego zwraca deskryptor pliku.

Jeśli gdzieś w programie zadeklarujemy zmienną: `int fd;`
możemy wywołać instrukcje postaci:

```
fd = open(name, flags, perms);
```

gdzie **`name`** wskaźnikiem do ciągu znaków zawierających nazwę pliku; **`flags`** jest liczbą typu `int` określającą sposób otwarcia wskazanego pliku. Najważniejsze wartości (nagłówek: `<fcntl.h>`)

`O_RDONLY`	*otwarcie tylko do czytania*
`O_WRONLY`	*otwarcie tylko do pisania*
`O_RDWR`	*otwarcie do czytania i pisania*

Jeżeli np. chcemy otworzyć istniejący plik *do czytania*, piszemy:

```
fd = open(name, O_RDONLY, 0);
```

argument perms równy 0 oznacza, że nie zmieniamy praw dostępu.

Próba otwarcia pliku, który nie istnieje, na ogół będzie błędem (chociaż w nagłówku `<fcntl.h>` mamy zdefiniowaną wartość **`O_CREAT`** argumentu `flags` do tworzenia nowego pliku).

Zwykle używamy do tego drugiej funkcji (**`creat`**), której drugi argument (perms) określa prawa dostępu dla tworzonego pliku.

Funkcja **`creat`** zwraca deskryptor do utworzonego pliku.

Jeśli funkcja **`creat`** nie zdoła utworzyć pliku (*większość systemów ogranicza liczbę jednocześnie otwartych plików dla jednego procesu…*) zwraca `-1`.

Jeśli tworzony plik już istnieje, zostanie **wyzerowany** (*tj. skrócony do zerowej długości, poprzednia zawartość ulegnie skasowaniu*).

Wspomiane wyżej ograniczenia ***liczby jednocześnie otwartych*** plików sprawiają, że program musi być przystosowany do wielokrotnego używania tych samych deskryptorów plików.

Kiedy plik staje się niepotrzebny (tj. *nie musi dłużej pozostawać otwarty*) możemy go zamknąć, tzn. ***przerwać połącznie pliku z jego deskryptorem***. Służy do tego funkcja: `close(int fd);`

[*Funkcja* `close` *zasadniczo jest odpowiednikiem* `fclose` *z bibloteki standardowej, nie ma jednak żadnych buforów do opróżnienia.*]

Z kolei funkcja `unlink(char *name)` usuwa plik o nazwie `name` z systemu plików. [Jest odpowiednikiem `remove` z biblioteki standardowej.]

Podobnie, funkcja **`fseek`** *z biblioteki standardowej (=> wyklad08) ma swój odpowiednik powiązany z odwołaniem systemowym.*

Jest nim funkcja jest **`lseek`** umożliwiająca ***dostęp swobodny*** do wybranego miejsca w otwartym pliku (a zarazem — *poruszanie się po pliku bez czytania/pisania* danych):

```
long lseek(int fd, long offset, int origin);
```

Funkcja **`lseek`** zmienia pozycję w pliku o deskryptorze `fd` na pozycję zadaną przez przesunięcie (`offset`) liczone w bajtach, od wybranego miejsca (`origin`).

Historycznie, wartości `origin` równe `0`, `1`, `2` oznaczają, że przesunięcie będzie liczone od początku, bieżącej pozycji, lub końca pliku.

W praktyce, zadając `origin` zwykle odwołujemy się do stałych zdefiniowanych w nagłówku **`<unistd.h>`** *(których nazwy w większości przypadków są samoopisujące)*:

`SEEK_SET` *(od początku)*, **`SEEK_CUR`** *(…)*, **`SEEK_END`** *(od końca)*, **`SEEK_HOLE`** (od początku *największej dziury* o rozmiarze większym lub równym wartości `offset`) …

Ostatni przykład (**`SEEK_HOLE`**) daje pojęcie o *zaletach odwołań systemowych* [=> ***przenośność programu .VS. łatwość pisania …***]

Obszar pliku wypełniony zerami może (*ale nie musi…*) być oznaczony przez system jako „dziura” (*hole*); odwołanie `lseek` może wyprowadzać poza pozycje `EOF` (wówczas, zapis w takim miejscu utworzy „dziurę”).

Informacje o pliku (np. o nazwie `name`) można uzyskać za pomocą funkcji **`stat`** (nagłówek: <**`sys/stat.h`**>)

```
int stat(const char *name, struct stat *buf);
```

Dane trafiają do *struktury*, która zawiera pola:

```
struct stat {
    mode_t         st_mode;     /* atrybuty pliku */
    ino_t             st_ino;   /* numer węzła */
    dev_t             st_dev;   /* urządzenie powiązane z węzłem */
    dev_t             st_rdev;  /* info. dla plików specjalnych */
    nlink_t           st_nlink; /* liczba linków do pliku */
    uid_t             st_uid;   /* identyfikator właściciela */
    gid_t             st_gid;   /* identyfikator zespołu */
    off_t             st_size;  /* rozmiar (w bajtach) */
    struct timespec st_atim;  /* data/czas ostatniego dostępu */
    struct timespec st_mtim;  /* data/czas ostat. modyfikacji */
    struct timespec    st_ctim; /* […] ostat. zmiany w węźle */
    …
};
```

Przydział pamięci

Odwołania systemowe umożliwiają także *przydzielania/zwalnianie* bloków pamięci dla programu. Służy do tego funkcja **`sbrk(n)`**, która zwraca wskaźnik (`void*`) do nowej porcji n bajtów pamięci, lub wartość `-1` (***a nie `NULL`!***) jeśli wolny blok o pożądanym rozmiarze nie został znaleziony.

*Przyjmuje się założenie, że wskaźniki zwracane przez **`sbrk`** mogą być **porównywane** (tak jak wskaźniki powiązane z tą samą tablicą!).*

Możemy w szczególności napisać własną wersję funkcji bibliotecznej **`malloc`** lub **`calloc`** [***zob.*** *Kernighan & Ritchie, …*]

Odwołania systemowe: *Podsumowanie*

Standard **POSIX** zapewnia *identyczność* (z punktu widzenia programisty…) licznych funkcji zdefiniowanych w nagłówku `<unistd.h>` a udostępniających ***odwołania systemowe***.

Odwołania systemowe umożliwiają w szczególności:

- niskopoziomowe operacje wejścia/wyjścia
- działania w systemie plików
- zarządzanie pamięcią

W wielu przypadkach, istnieją ***funkcje biblioteki standardowej*** zastępujące odwołania. Zwykle są prostsze w użyciu (*kosztem wydajności …*) i powiązane ze *standardem języka* a nie systemu.