jeszcze dygresja o macierzach...

```
#include <stdlib.h>

#include <stdio.h>

/* przyklad pokazuje, jak dla macierzy
wielowymiarowych dziala operator

dereferencji * ; otoz jesli macierz jest np.
trzywymiarowa i nazywa sie mat,

  to wskazniki  mat, *mat, **mat zawieraja TEN SAM
ADRES! taka jest umowa   w jezyku C        */
```

jeszcze dygresja o macierzach...

```
int main ()
{   int i,k,l;
    int mat[3][5][7];
  for(i=0;i<3;++i)
  for(k=0;k<5;++k)
  for(l=0;l<7;++l)
   mat[i][k][l]=10000*i + 1000*k + 100*l + 7;
   printf("\n  wskazniki mat=%p *mat=%p **mat=%p ***mat=%p
***mat=%d\n",  mat,*mat,**mat,***mat, ***mat);
   printf("\n\n");
   printf("    *(*(*(mat+1)+2)+3) = %d\n", *(*(*(mat+1)+2)+3) );
   exit(0);    } /*  koniec  main  */
```

jeszcze dygresja o macierzach...

/* to wynik programu */

wskazniki mat=737f0a54 *mat=737f0a54 **mat=737f0a54
***mat=00000007 ***mat=7

((*(mat+1)+2)+3) = 12307

Rekurencja prawostronna

```
float silnia (int n)
/* to nie jest rekurencja prawostronna  */
{ if(n==0 || n==1) return (1.);
   else
   return(n * silnia(n-1) );
}
```

Rekurencja prawostronna

wywołanie rekurencyjne jest rekurencją prawostronną, gdy jest ostatnią instrukcją wykonywaną w funkcji, a zwracana wartość nie jest częścią wyrażenia; kompilator potrafi to wykorzystać

Rekurencja prawostronna

```
float silnia(int n, float a)
{  if(n<0) return(0.);  if(n==0) return (1.);
   if(n==1) return(a);
   else
   return(  silnia(n-1, n*a)  );
}/* koniec funkcji silnia   */
```

silnia(n,a) jest równa a, gdy n=1

jest równa silnia(n-1,na) gdy n>1

Rekurencja prawostronna

Jak zapisać przez rekurencję prawostronną sumowanie

$H(n)=1 +1/2 + 1/3 + 1/4 +...+1/n$?

$$H(n,a) = \begin{cases} a+1, & \text{gdy } n=1 \\ H(n-1, a+1/n) & \text{gdy } n>1 \end{cases}$$

Little Endian oraz Big Endian

W zależności od tego, jaki komputer jest używany, porządek bajtó może być różny od tego porządku zależy interpretacja zmiennych (stałych) wielobajtowych.

Np. dla 4-bajtowej zmiennej long int

bajt3 bajt2 bajt1 bajt0

uporządkowanie w pamięci może być następujące:

bajt0 Adres+0

bajt1 Adres+1

bajt2 Adres+2

bajt3 Adres+3

Little Endian oraz Big Endian

Komputery z takim porządkiem bajtów (najmniej znaczący bajt ma najniższy adres) są określane jako „Little Endian machines”.

Należą do nich w szczególności wszystkie komputery z procesorami Intela (oraz AMD). Z kolei komputery IBM RISC były/są maszynami „Big Endian”; także maszyny tzw. „mainframe” (również komputery Apple Inc.) :

bajt3 Adres+0
bajt2 Adres+1
bajt1 Adres+2
bajt0 Adres+3

(bajt najbardziej znaczący ma najniższy adres)

Little Endian oraz Big Endian

Używa się jeszcze określenia „Network byte order”. To to samo co „Big Endian”.

Niekiedy można się spotkać z określeniem „Host byte order”. Zwykle oznacza to „Little Endian”.

Little Endian oraz Big Endian

- Adobe Photoshop - Big Endian
- BMP - Little Endian
- GIF - Little Endian
- IMG (GEM Raster Image) – Big Endian
- JPEG - Big Endian
- MacPaint - Big Endian
- PCX - Little Endian
- Quicktime Movies (Mac) - Little Endian
- Microsoft RTF - Little Endian

Little Endian oraz Big Endian

- TGA (Targa) - Little Endian
- WPG (WordPerfect Graphics Metafile) - Big Endian (używany na PC!)
- TIFF - Little Endian i Big Endian, znacznik kodowania w zbiorze)
- SGI (Silicon Graphics) - Big Endian

używanie komend UNIXa z wnętrza programu napisanego w C

#include <stdio.h>

int **system**(char *string)

string może być komendą systemu UNIX lub ”shell script” lub programem użytkownika ; **system** zwraca po powrocie wartość **exit.**

```
int  main()
{  printf(”Oto zbiory w kartotece: \n");  system(”ls -l"); }
```

(najczęściej **system** jest zbudowany z wywołań trzech innych funkcji

exec(), wait(), fork())

używanie komend UNIXa z wnętrza programu napisanego w C

execl(char *path, char *arg0,...,char *argn, 0);

path wskazuje na nazwę zbioru zawierającego komendę

arg0 wskazuje na string który jest taki sam jak path (lub przynajmniej jego ostatni człon)

Ostatni parametr musi być zero.

```
main()
{ printf(”Zbiory: \n");   execl (”/bin/ls",   ”ls",   ”-l",   0);
}  /*  koniec funkcji  main  */
```

używanie komend UNIXa z wnętrza programu napisanego w C

int **execv** *(const char *filename, char *const argv[])*

Funkcja ta wykonuje plik o nazwie *filename* jako nowy proces (wykona fork); argument argv jest macierzą wskaźników; będzie ona argumentem argv w funkcji main tego nowego procesu. Ostatni element macierzy argv musi być wskaźnikiem zerowym (patrz execl). Pierwszy element macierzy argv musi być nazwą wykonywanego pliku ”bez kartoteki”.

używanie komend UNIXa z wnętrza programu napisanego w C

int **execve** *(const char *filename, char *const argv[], char *const env[])*

pozwala dodatkowo przesłać informację o otoczeniu procesu

używanie komend UNIXa z wnętrza programu napisanego w C

int **execle** *(const char *filename, const char *arg0, char *const env[], ...)*

int **execvp** *(const char *filename, char *const argv[])*

int **execlp** *(const char *filename, const char *arg0, ...)*

fork

#include <unistd.h>

int **fork()**

Dokonuje zamiany procesu na dwa identyczne procesy, nazywane rodzic i potomek (parent and child). W przypadku sukcesu, **fork** zwraca 0 do potomka i numer procesu (process ID) do rodzica. W przypadku błędu fork zwraca –1 i nie tworzy potomka (procesu potomnego).

fork

```
#include <unistd.h>
int main()
{   printf(”Forking process\n");
fork();
printf(”process id jest rowny %d\n",   getpid() );
/*   ewentualnie inne instrukcje programu  */
 printf(”\nOstatnia linia\n");
} /*  koniec  funkcji  main   */
```

fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
    int main ()
{   int n,l;
    n=fork();
    if(n==0) sleep(5);
    printf("\n getpid()=%d  n=%d\n", getpid(), n);
    exit(0);
}/*   koniec funkcji main   */
```

fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
   int main ()
{ int status, n, m;
   n=fork();
  m=wait (&status);
  printf("\n m=%d  n=%d status=%d\n",m,n,status);
  printf("\n getpid()=%d  n=%d\n", getpid(), n);
  if(n==0) exit(10);  /* kod zakończenia dla potomka   */
  else exit(0);                }              *   koniec funkcji main   */
```

wynik programu

m=-1 n=0 status=134518208

getpid()=11430 n=0

m=11430 n=11430 status=2560

getpid()=11429 n=11430

wait

int **wait** (int *status) – proces wołający jest zawieszony (”suspended”) dopóki proces potomny nie zakończy się.

wait zwraca ID procesu-potomka lub –1 w przypadku błędu. ”exit status” potomka jest zwracany do miejsca pamięci wskazywanego przez wskaźnik status.

uwaga: wait jest uproszczona wersją funkcji

waitpid (-1, &status, 0)

waitpid

#include <sys/wait.h>

pid_t **waitpid** (*pid_t pid, int *status-ptr, int options*)

funkcja ta dostarcza informacji o procesie potomnym o numerze procesu pid. Zazwyczaj proces wołający jest zawieszony tak długo, aż proces potomny udostępni informację o statusie swojego zakończenie (np. kończąc się normalnie czy ”padając” na skutek jakiegoś błędu).

Niektóre wartości pid mają specjalne znaczenie, np. –1 czy WAIT_ANY oznacza którykolwiek z procesów potomnych. Np. pid równy 0 żąda informacji o jakimkolwiek procesie pochodnym z tej samej grupy procesów jak proces wołający.

Argument *options* : można użyć WNOHANG dla zaznaczenia, że proces nadrzędny nie powinien czekać (czyli waitpid() go nie zawiesi), czy WUNTRACED by zgłosić wymaganie by zwracana była informacja tak od procesów zastopowanych jak i zakończonych.

Inter Process Communication (IPC) - pipe

Podstawowym użyciem ”of pipes” czyli potoków jest przesyłanie lub otrzymywanie danych z programu uruchomionego jako proces potomny (”child process”). Można to zrobić albo wykorzystując ”Low Level piping”, czyli używając funkcji **pipe** (aby stworzyć pipe), **fork** (aby stworzyć proces potomny), **dup2** (aby wymusić na procesie potomnym by używał dany pipe jako swój standard input lub standard output), **exec** (aby wykonać nowy program).

Lub można skorzystać z ”High Level Piping” czyli funkcji

popen oraz **pclose**

Inter Process Communication (IPC) - pipe

FILE * **popen** *(const char **command, const char **mode)* **;**

int **pclose** *(FILE **stream);*

popen uruchamia proces potomny. Nie czeka na zakończenie tego procesu, tworzy pipe do tego procesu i zwraca strumień odpowiadający temu pipe. Jeśli jako **mode** poda się ”r”, to można czytać ze standard output związanym z wykonującym się procesem potomnym. Analogicznie jeśli podać **mode** jako ”w”, to można pisać po standard input związanym z wykonującym się procesem potomnym.

pclose zamyka to co **popen** otwarło

Inter Process Communication (IPC) - pipe

```
#include <stdio.h>
#include <stdlib.h>
        int main ( )
{ int n;
  FILE * fp;
  fp = popen (  ”ls –l” , "r");
  while( (n=fgetc(fp))!=EOF )
  {
   printf("%c",n);
  }    }       /*  koniec funkcji main */
```

Inter Process Communication (IPC) - Low Level Pipe

int **pipe**(int fd[2]) – tworzy pipe i zwraca dwa deskryptory do plików, fd[0], fd[1]. fd[0] jest otwarte dla czytania, fd[1] dla pisania.

pipe() zwraca 0 w przypadku sukcesu, -1 w przypadku błędu

Standardowy sposób użycia polega na tym, że zaraz po tym jak pipe jest utworzone, dwa (albo więcej) współdziałających procesów będzie stworzonych przez fork ; dane będą przesyłane używając read() oraz write() (można bez problemu używać także np. fscanf, fgetc, fprintf, fputc, można przecież użyć fdopen dla uzyskania wskaźnika do struktury FILE)

pipe otwarte przez **pipe()** można zamknąć przez **close(int fd).**

Przykład: rodzic pisze do dziecka

Inter Process Communication (IPC) - Low Level Pipe

```
int pdes[2];
pipe(pdes);
if ( fork() == 0 )
 { /* child */    close(pdes[1]);    /* niepotrzebne */
read( pdes[0],.....);    /* czytaj od rodzica*/ ..... }
else
{ close(pdes[0]); /*  niepotrzebne  */
write( pdes[1].....); /* pisz do potomka */ .....
}
```

Low Level Pipe - przykład

```
#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/types.h>

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

Low Level Pipe - przykład

```
void read_from_pipe (int file)
/* czytaj znaki z  "potoka" (czyli z pipe)  i wypisuj je na stdout  */
{ int n=1;
    while( n>=0 )
    { int n;
      char b;
      n=read(file,&b,1);
      if(n==0) {printf(" potomek robi break");break;}
      write(STDOUT_FILENO,&b,n);
    }
    printf("\n potomek zaraz konczy \n");   }  /*   koniec  funkcji
read_from_pipe*/
```

Low Level Pipe - przykład

```
void write_to_pipe (int file)
{
  char buf[30]=”Przesyłam pewien tekst, potem usnę, następnie zamknę";
  write(file,buf,30);
  sleep(6);  /* usypianie na 6 sekundy  */
  close (file);
}
```

Low Level Pipe - przykład

```
int main (void)
{
  pid_t pid;
  int mypipe[2];
       /* utwórz pipe */
  if (pipe (mypipe))
    {
      fprintf (stderr, "Pipe zawiodlo!\n");
      return EXIT_FAILURE;
    }
```

Low Level Pipe - przykład

```
        /* utworz proces pochodny */
pid = fork ();
if (pid == (pid_t) 0)
  {
    /* to jest proces pochodny  */
    close (mypipe[1]);
    read_from_pipe (mypipe[0]);
    return EXIT_SUCCESS;
  }
```

Low Level Pipe - przykład

```
else if (pid < (pid_t) 0)   /*  nadal część dla procesu potomnego   */
   {
    /* fork zawiodl */
    fprintf (stderr, "Fork zawiodl\n");
    return EXIT_FAILURE;
   }
```

Low Level Pipe - przykład

```
else
   {
    /* to jest proces-rodzic czyli proces początkowy */
    close (mypipe[0]);
    write_to_pipe (mypipe[1]);
    printf("\n Rodzic zaraz konczy\n");
    return EXIT_SUCCESS;
   }
}
```

funkcja dup2()

int **dup** (*int old*)

Ta funkcja kopiuje deskryptor *old* na pierwszy wolny numer deskryptora (tzn. pierwszy nie otwarty)

int **dup2** (*int old, int new*)

Ta funkcja kopiuje deskryptor *old* na deskryptor *new*.

Zastosowanie: można zduplikować deskryptor pliku; zduplikowane deskryptory odwołują się do tego samego otwartego zbioru. Wskazują na tą samą pozycję wewnątrz zbioru. Zduplikowanie deskryptora umożliwia ”redirection of input or output”, czyli można zmienić zbiór czy potok (=”pipe”) do którego odnosi się dany deskryptor.

funkcja dup2() - przykład

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
  #include <sys/types.h>
  #include <sys/stat.h>
        int main ()
{ char buf[20]="Kaskada\n";   /*  osiem znaków  */
  FILE * fp1;
  int index;
  int k;
```

funkcja dup2() - przykład

```
index=open("nic.dat",O_RDWR|O_TRUNC|O_CREAT,0664);
        /* do czytania i pisania;
           jeśli istnieje skróć plik do zera;
           jeśli plik nie istnieje to go utwórz  */

 k=write(index,buf,8);
        /* fp1=fdopen(index,"r+");
           fprintf(fp1,"\nDodatkowa Linia\n");  */
 fp1=fdopen(index,"w");
```

funkcja dup2() - przykład

```
fprintf(fp1,"\nTo powinno zapisac sie w zbiorze nic.dat\n");

      fflush(fp1); /* co jesli fflush() usuniete?     */

 printf("\n fileno(stdout)=%d\n    za chwile  dup2()\n",
          fileno(stdout) );

 dup2(fileno(stdout), index);

 k=write(index,buf,8);  /* to będzie zapisane na stdout  */

   fprintf(fp1,"\n  to takze bedzie zapisane na stdout\n");



   exit(0);

} /*   koniec  main   */
```

Inter Process Communication (IPC) - FIFO

FIFO jest inaczej nazywane ”named pipe”. FIFO pojawia się jako specjalny zbiór w systemie zbiorów. Aby go stworzyć należy zawołać funkcje **mkfifo().**

Gdy plik FIFO jest utworzony, każdy proces może go otworzyć do czytania i pisania jak zwykły plik. Ale - musi być otwarty z obu końców. Otwarcie FIFO do czytania powoduje zawieszenie procesu, aż jakiś proces otworzy to samo FIFO do pisania, i odwrotnie.

#include <sys/stat.h>.

int **mkfifo** *(const char *filename, mode_t mode)*

/ utworzy plik FIFO o nazwie filename; argument mode ma to samo znaczenie co przy funkcji open, ustawia prawa dostępu do tego specjalnego pliku */*

Inter Process Communication (IPC) - FIFO

FIFO (inaczej ”named pipe”) można utworzyć komendą systemu LINUX

mkfifo ”nazwa zbioru”

potem oczywiście można otworzyć taki potok z dwóch stron np. dwoma programami z których jeden czyta a drugi pisze

przykład programu: **users.uj.edu.pl/~ufrudy/forkop2.c** (w przykładzie tym po wykonaniu fork(), potomek sortuje i wynik sortowania przekazuje do nadrzędnego procesu)

ls –l nazwany_potok

prwxr--r-- 1 rudy users 0 Jan 5 16:07 nazwany_potok

IPC - Sygnały

Sygnały są programowo generowanymi przerwaniami przesyłanymi do procesu gdy zdarza się jakieś zdarzenie. W szczególności **sygnał może być przesłany do procesu z innego procesu**. Większość sygnałów powoduje zakończenie procesu do którego zostały przesłane, chyba że zostanie podjęta jakaś akcja w odpowiedzi na sygnał.Każdy sygnał ma jakąś określoną standardową (”default”) akcję, którą może być:

przerwanie procesu, zastopowanie procesu itp..

IPC - Sygnały

Sygnał przekazuje informację o zdarzeniu się czegoś wyjątkowego. Np. że program dzieli przez zero lub że zażądał adresu pamięci spoza przydzielonego obszaru. Inne możliwości:

-żądanie użytkownika (procesu) dotyczące zatrzymania programu (procesu) lub przesłanie mu przerwania, tak by poinformowany proces mógł to przerwanie obsłużyć

-zakończenie procesu potomnego.

Ogólnie, sygnały są ograniczoną ale bardzo użyteczną formą porozumiewania się procesów pomiędzy sobą (porozumiewanie typu ”look at me”)

IPC - Sygnały

SIGHUP 1 /* hangup */

SIGINT 2 /* interrupt CTRL-c*/

SIGQUIT 3 /* quit CTRL-\ */

SIGILL 4 /* illegal instruction */

SIGABRT 6 /* used by abort */

SIGKILL 9 /* hard kill */

SIGALRM 14 /* alarm clock */

SIGCONT 18 /* continue a stopped process */

SIGCHLD 20 /* to parent on child stop or exit */

SIGSTOP 19/*zatrzymaj proces*/

SIGTERM 15

Signals mogą mieć numery od 0 do 31

IPC - Sygnały

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
   int main()
{
   printf("\n strsignal(9) = %s\n", strsignal(9) );
   printf("\n strsignal(12) = %s\n", strsignal(12) );

   exit(0);
} /* koniec funkcji main */
/* wynik programu
 strsignal(9) = Killed

 strsignal(12) = User defined signal 2
*/
```

IPC - Sygnały

char * **strsignal** (*int signum*)

Funkcja powyższa zwraca wskaźnik do łańcucha znakowego przechowującego informację o tym, czym jest sygnał *signum*.

IPC - Sygnały

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
   int main()
{
   printf("\n strsignal(9) = %s\n", strsignal(9) );
   printf("\n strsignal(12) = %s\n", strsignal(12) );

   exit(0);
} /* koniec funkcji main */
/* wynik programu
 strsignal(9) = Killed

 strsignal(12) = User defined signal 2
*/
```

IPC - Sygnały

Sygnały zdefiniowane w systemie należą do jednej z poniższych klas:

- sytuacje hardware'owe
- sytuacje software'owe
- kontrolowanie procesu (w tym WE/WY)
- kontrolowanie zasobów systemu

IPC – Sygnały; przykładowy program

```
/* Przykład jak dwa procesy mogą rozmawiać */
/* używając  kill() oraz signal() */
/* wykonamy fork(), rodzic wyśle kilka sygnałow do potomka  */
     #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
void sighup(); /* funkcje które bedzie wołał potomek gdy  sigtrap */
void sigint();
void sigquit();
void sig11();
```

IPC – Sygnały; przykładowy program

```
 int  main()
{ int pid;


 /* proces potomny */


  if ((pid = fork()) < 0) {
       perror("fork");
       exit(1);
    }
```

IPC – Sygnały; przykładowy program

```
if (pid == 0)
  { /* potomek */
    signal(11,sig11);
    signal(SIGHUP,sighup);
    signal(SIGINT,sigint);
    signal(SIGQUIT, sigquit);
    for(;;); /* wieczna pętla */
  }
```

IPC – Sygnały; przykładowy program

```
else      /*    rodzic    */
 {  /* pid  zawiera ID  potomka */
   printf("\nPARENT: sending SIGHUP\n\n");      kill ( pid, SIGHUP);
   sleep(3); /* pause for 3 secs */
   printf("\nPARENT: sending SIGHUP\n\n");      kill ( pid, SIGHUP);
   sleep(3); /* pause for 3 secs */
   printf("\nPARENT: sending     11\n\n");           kill ( pid, 11);
   sleep(3); /* pause for 3 secs */
   printf("\nPARENT: sending SIGINT\n\n");       kill ( pid, SIGINT);
   sleep(3); /* pause for 3 secs */
   printf("\nPARENT: sending SIGQUIT\n\n");      kill (pid, SIGQUIT);
   sleep(3);
 }
```

IPC – Sygnały; przykładowy program

```
void sig11()
{   printf("CHILD: I have received a SIGNAL=11\n");
}
void sighup()
{    printf("CHILD: I have received a SIGHUP\n");
}
void sigint()
{    printf("CHILD: I have received a SIGINT\n");
}
void sigquit()
{ printf("Mój tata mnie zabił!!!\n");    exit(0);       }
```

IPC – Sygnały

Dostępne są dwie funkcje do wysyłania sygnałów

int kill(int pid, int signal)

wysłanie sygnału *signal* do procesu *pid;* jeśli *pid* jest większe niż zero, sygnał jest wysłany do procesu którego *pid* jest równe *pid* ; jeśli *pid* jest 0, sygnał zostanie wysłany do wszystkich procesów użytkownika; *kill* zwraca 0 w przypadku sukcesu, -1 w przypadku niepowodzenia.

IPC – Sygnały

Dostępne są dwie funkcje do wysyłania sygnałów

int raise (int sig) ;

wysyła sygnał *sig* do wykonującego się procesu; *raise()* przy użyciu *kill()* wysyła sygnał komendą:

kill(getpid(), sig);

(istnieje komenda UNIX'a kill)

IPC – Sygnały

Sygnały przyjmuje się przy użyciu funkcji *signal()*

int (signal(int sig, void(*func) ( ))) ( )

funkcja *signal()* w przypadku otrzymania sygnału *sig* zawoła funkcję *func()*; *signal ()* zwraca wskaźnik do funkcji *func* w przypadku sukcesu oraz –1 w przypadku niepowodzenia

func() może przyjmować trzy wartości:

- SIG_DFL
- SIG_IGN
- napisana przez użytkownika funkcję

IPC – Sygnały

Np. aby **ignorować** komendę CTRL/C można wywołać

signal(SIGINT, SIG_IGN)

(uwaga – nie można **ignorować** sygnału SIGKILL)

aby SIGINT ponownie powodowało przerwanie programu/procesu należy wywołać

signal(SIGINT, SIG_DFL)

IPC – Sygnały

```
/*  przykład programu z funkcjami obsługującymi sygnały   */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void sigproc();    /* to będą funkcje obsługujące sygnały czyli
                          ”signal handlers”   */
void quitproc();
```

IPC – Sygnały

```
main()
{               signal(SIGINT, sigproc);
                signal(SIGQUIT, quitproc);
                printf("ctrl-c nie działa użyj ctrl-\\  \n");
                for(;;); /* pętla nieskończona */
}
```

IPC – Sygnały

```
void sigproc()
{           signal(SIGINT, sigproc);
            /* Niektóre wersje UNIX'a znoszą obsługę sygnału po
każdym zawołaniu. Dla pewności ponownie zgłaszana jest obsługa
sygnału */

            printf("nacisnales  ctrl-c\n");
}
```

IPC – Sygnały

```
void quitproc()
{               printf(”nacisnieto   ctrl-\\   wychodze!! \n");
                exit(0); /* normal exit  */
}
```

IPC – Sygnały
blokowanie/ignorowanie

Czym się różni blokowanie sygnału od jego ignorowania?

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
main()
{ sleep(5);                    printf("\n poczatek blokowania...\n");
  sighold(SIGINT);        sleep(5);
  printf("\n koniec blokowania...\n");
  sigignore(SIGINT);     signal(SIGINT,SIG_DFL);  /*    ?!!!!   */
  sigrelse(SIGINT);        for(;;);
}   /* w tym przykładzie sighold()  blokuje sygnał SIGINT   */
```

IPC – Sygnały
blokowanie/ignorowanie

Inne funkcje pracujące z sygnałami

#include <signal.h>

int sighold(int sig) - dodaje sig do sygnałowej maski procesu

int sigrelse(int sig) - usuwa sig z sygnałowej maski programu

int sigignore(int sig) - ustawia dyspozycję dla sygnału sig na ”ignore”

(jeśli nie ma sighold(), to jest funkcja sigblock())

IPC – Sygnały
(zatrzymanie ,uruchomienie procesu potomnego)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
 int main()
{ int pid;
    /* proces potomny */
   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }
```

IPC – Sygnały

(zatrzymanie ,uruchomienie procesu potomnego)

```
if (pid == 0)
    { /* potomek */
     for(;;)  /* wieczna pętla */
     {
       printf("\n child: %d\n",getpid());
       sleep(1);
     }
    }
```

IPC – Sygnały
(zatrzymanie ,uruchomienie procesu potomnego)

```
else /* parent */
    {  /* pid  zawiera ID  potomka */
     sleep(5);          printf("\nPARENT: sending SIGSTOP\n\n");
     kill(pid,SIGSTOP); /* zatrzymanie */
          sleep(5);     printf("\nPARENT: sending SIGCONT\n\n");
          kill(pid,SIGCONT); /* uruchomienie */
     sleep(5);          printf("\nPARENT: sending SIGQUIT\n\n");
     kill(pid,SIGQUIT); /* sygnal quit */
    }
}/* koniec  main  */
```

IPC – Sygnały
(floating point exceptions)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fenv.h>
#include <signal.h>
      void sigfpe();
int main()
{  float f1,f2;
   int n=-100;
   signal(SIGFPE,sigfpe);/* sigfpe() będzie obsługiwac sygnał*/
```

IPC – Sygnały
(floating point exceptions)

```
 //n=feenableexcept(FE_ALL_EXCEPT);
  n=feenableexcept(FE_DIVBYZERO);
/*   feenableexcept  aktywuje  przerwanie  */
//n=fedisableexcept(FE_ALL_EXCEPT);


  //n=feenableexcept(FE_UNDERFLOW);
  printf("n=%d\n",n);
  printf("    %d\n",FE_ALL_EXCEPT);
     f1=1.0;    f2=0.0;     f1=f1/f2;
```

IPC – Sygnały
(floating point exceptions)

```
 printf("\n f1=%f\n",f1);
  exit(0);
}/*  koniec  funkcji  main  */


void sigfpe()   /* obsługuje sygnał  */
{ printf("\n dzielenie przez zero!\n");
  exit(0);
}
```

IPC – Sygnały
(floating point exceptions)

na zbiorze sigfpe5.c podany jest przykład, jak zachować maskę dla wyjatków zmienno przecinkowych, a następnie ją odtworzyć

IPC – status zakończenia procesu (potomnego)

Jeśli wartość zwracana przez proces potomny jest zero, to także wartość uzyskana poprzez **waitpid()** czy **wait()** jest równa zeru. Można otrzymać inne informacje zakodowane w uzyskanej wartości używając poniżej przedstawionych makr. Makra te są zdefiniowane w pliku **sys/wait.h**, który należy dołączyć poprzez #include <sys/wait.h>

int **WIFEXITED** (*int status*)

To makro zwraca wartość różną od zera jeśli proces potomny zakończy się normalnie wywołaniem **exit()** lub _exit() .

int **WEXITSTATUS** (*int status*)

Jeśli WIFEXITED(status) jest TRUE, makro to zwraca 8 bitów wartości zwracanej przez exit() w procesie potomnym

int **WIFSIGNALED** (*int status*)

Makro to zwraca wartość niezerową jeśli proces potomny zakończył się, gdyż otrzymał sygnał który nie został obsłużony (nie było odpowiedniego ”signal handling”)

IPC – status zakończenia procesu (potomnego)

int **WTERMSIG** (*int status*)

Jeśli WIFSIGNALED(status) jest TRUE, makro to zwraca numer sygnału który zakończył proces potomny.

int **WCOREDUMP** (*int status*)

To makro zwraca wartość niezerową, jeśli proces potomny zakończył się i wyprodukował ”core dump”.

int **WIFSTOPPED** (*int status*)

Powyższe makro zwraca wartość niezerową jeśli proces potomny jest zastopowany.

int **WSTOPSIG** (*int status*)

Jeśli WIFSTOPPED(status) jest TRUE, to makro zwraca numer sygnału który spowodował zastopowanie procesu potomnego.

IPC – status zakończenia procesu (potomnego)

```
/* przykład  zawołania   WEXITSTATUS()   */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <fcntl.h>
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <sys/wait.h>
```

IPC – status zakończenia procesu (potomnego)

```
/* przykład  zawołania   WEXITSTATUS()   */
 int main ()
{
  int n;

  n=fork();
  if(n==0)
  { /*  to potomek  */
   exit(5);

  }
```

IPC – status zakończenia procesu (potomnego)

```
/* przykład  zawołania   WEXITSTATUS()   */
else
  {     /*  to proces nadrzędny == rodzic  */
   int status;
   wait(&status);
   printf("\n potomek zwrocil status=%d\n",status);
   printf("\n czyli WEXITSTATUS=%d\n",WEXITSTATUS(status)); /* 5!!  */
    printf("\n program nadrzedny zaraz konczy prace...\n");
    exit(0);
  }
}/*   koniec funkcji main */
```

IPC – funkcja pause()

Jeśli chcemy, by program zatrzymał się, dopóki nie nadejdzie do niego jakis sygnał (który zdecyduje o jego dalszym losie), zazwyczaj używa się funkcji pause()

```
#include <unistd.h>

int pause ()
```

Funkcja ta zawiesza wykonanie programu; oczekiwanie jest albo nieskończone, albo nadejdzie jakiś sygnał który zakończy program (SIGINT, SIGQUIT ,SIGKILL) albo też sygnał nadchodzący wykona ”handler function of given signal”, czyli przygotowaną przez użytkownika funkcję obsługującą dany sygnał.

Jeśli ”pauzowanie” zostanie przerwane, to pause() zwraca –1.

Następujący kod **errno** jest zdefiniowany dla funkcji pause():

EINTR funkcja została przerwana na skutek otrzymania sygnału

IPC – funkcja pause()

```
/* Przyklad jak dwa procesy moga rozmawiac */
/* uzywajac  kill() oraz signal() */
/* wykonamy fork()  */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void sigcont();

main()
{ int pid;
  int n;
```

IPC – funkcja pause()

```
/* wykonanie fork()    */


  if ((pid = fork()) < 0) {
       perror("fork");
       exit(1);
    }
```

IPC – funkcja pause()

```
if (pid == 0)
    { /* potomek */
     for(n=1;;++n)  /* wieczna pętla   /
     {
       signal(SIGCONT, sigcont);
       printf("\n child: %d\n",getpid());
       sleep(1);
       if(n==2) pause();   /*  proces zatrzyma się, będzie czekał */
     }
    }
```

IPC – funkcja pause()

```
else        /* parent   czyli  proces nadrzedny  czyli  rodzic */
    {  /*   pid  zawiera ID  potomka */

      sleep(7);  /* symulacja jakiegoś przetwarzania....  */
      printf("\nPARENT: sending SIGCONT\n\n");
      kill(pid,SIGCONT); /* uruchomienie */
      sleep(7);

    }
}/* koniec  main  */
```

IPC – funkcja pause()

```
void sigcont()
{
  printf("\n potomek: byl sygnal SIGCONT\n");
}/*  koniec  sigcont  */
```

/* proces może zatrzymać czasowo jakiś inny proces, wysyłając do niego najpierw sygnał SIGSTOP, a po pewnym czasie może go obudzić wysyłając sygnał SIGCONT   */

IPC – funkcja sigpause()

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
    void function11();
main()
{ int maska,n=0;
    signal(11,function11 );
   printf("\n numer procesu = %d\n", getpid() );
   maska=sigmask(SIGINT) | sigmask(SIGQUIT);
     sigpause(maska); /*ustawia maskę sygnałową procesu na
           maska i czeka na sygnał;   podobnie jak sigsuspend()*/
```

IPC – funkcja sigpause()

```
for(;;)
  {  printf("\n zaraz usne na sekunde  n=%d\n",++n);
     sleep(1);
  }
}/* koniec funkcji  main  */


    void function11()
{
  printf("\n handler sygnalu 11,  function11\n");
}
```

IPC – funkcja sigpause()

```
#include <stdio.h>   /* ten program może obudzić poprzedni program  */
#include <stdlib.h>  /* ten poprzedni zawołał  sigpause ()  */
#include <signal.h>
     main()
{  int n;
   printf("\npodaj numer procesu\n");
   scanf("%d", &n);
   kill(n,11); /* wyslij sygnal 11 */
   sleep(10);
   kill(n,SIGKILL); /* wyslij sygnal SIGKILL */
   exit(0);          } /* koniec funkcji main  */
```

Zaawansowana obsługa sygnałów

#include <signal.h>

int sigaction(...)

Podstawowy efekt działania tej funkcji jest podobny jak funkcji signal() – definiuje jak przychodzący sygnał powinien zostać obsłużony; sigaction oferuje jednak znacznie wiecej możliwości poprzez możliwość wyspecyfikowania dodatkowych flag (znaczników) kontrolnych. Użycie sigaction jest znacznie bardziej skomplikowane.

Tworzenie zbioru sygnałów

”signal set”

Wszystkie blokujące sygnały funkcje używają struktury danych nazwanej **signal set** w celu wyspecyfikowania o które sygnały chodzi. Używanie tych funkcji jest dwustopniowe: najpierw tworzy się **”signal set”,** następnie przesyła się go jako argument do odpowiedniej funkcji **(bibliotecznej).**

#include <signal.h>

sigset_t

Ten powyższy typ danych reprezentuje **”signal set”.** W danej wersji języka C może być zaimplementowany jako typ int lub jako struktura. Dla celów przenaszalności (”portability of code”) należy inicjalizować, zmieniać, odzyskiwać informację ze zmiennych typu sigset_t ***wyłącznie*** przy użyciu przedstawionych poniżej funkcji.

”signal set”

Są dwa sposoby zainicjalizowania zbioru sygnałów. Można rozpocząć od **sigemptyset(),** to określa pusty zbiór sygnałów, a następnie dodawać poszczególne sygnały przez **sigaddset()** . Inny sposób to zainicjowanie zbioru sygnałów pełnym zbiorem istniejących możliwych sygnałów przez **sigfillset()** a następnie usuwanie poszczególnych sygnałow przez **sigdelset()** .

Wysoce zalecane jest wybranie jednej z tych dwóch możliwości!

”signal set”

int **sigemptyset** (*sigset_t *set*)

Incjalizuje zbiór sygnałów (pusty). Zawsze zwraca zero.

int **sigfillset** (*sigset_t *set*)

Inicjalizuje zbiór sygnałów (pełny). Zawsze zwraca zero.

int **sigaddset** (*sigset_t *set, int signum*)

Dodaje sygnał *signum* do zbioru sygnałów. Zwraca zero w przypadku sukcesu oraz –1 w przypadku niepowodzenia.

int **sigdelset** (*sigset_t *set, int signum*)

Usuwa sygnał *signum* ze zbioru sygnałów. Zwraca zero w przypadku sukcesu oraz –1 w przypadku niepowodzenia.

int **sigismember** (*const sigset_t *set, int signum*)

Sprawdza czy sygnał *signum* jest zawarty w zbiorze sygnałów. Zwraca 1 jeżęli sygnał jest w zbiorze sygnałow, 0 jeśli go nie ma, -1 jeśli był błąd.

”signal set”

int **sigprocmask** (*int how, sigset_t * set, sigset_t * oldset*)

funkcja ta służy do zmiany maski sygnałów w danym procesie lub uzyskania informacji jaka jast ta maska. Argument *how* wybiera, jak funkcja działa.

Możliwe wartości *how* :

SIG_BLOCK

Blokuj sygnały które są w set -- dodaj je do istniejącej maski. Czyli nowa maska jest sumą dotychczasowej maski i *set.

SIG_UNBLOCK

Odblokuj sygnały które są w *set* – usuń je z istniejącej maski.

SIG_SETMASK

Sygnały z *set* tworzą nową maskę; zignoruj poprzednią wartość maski.

(jeśli *set* jest NULL, maska procesu nie zostanie zmieniona, jedynie zostanie załadowany *oldset* informacją o bieżącej masce procesu.

”signal set”

int **sigsuspend** (*const sigset_t *set*)

Ta funkcja zastępuje maskę sygnałową procesu przez *set* *a* następnie zawiesza proces dopóki nie dotrze sygnał który nie należy do *set* .

Funkcja **sigsuspend** kończąc się zawsze przywraca poprzednią maskę sygnałową procesu.

”signal set”: przykład programu

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
   void sigcont();
main()
{ int pid;
  int n;
        if ((pid = fork()) < 0)
        {
           perror("fork");
           exit(1);
        }
```

”signal set”: przykład programu

```
if (pid == 0)       /* potomek */
    {  sigset_t tempset;
       sigfillset(&tempset);
       sigdelset(&tempset,SIGCONT); /* sygnalu SIGCONT nie ma w masce  */
       signal(SIGCONT, sigcont);
             for(n=1;;++n)  /* wieczna pętla */
             {
                printf("\n child: %d\n",getpid());
                sleep(1);
                if(n==3) sigsuspend(&tempset);
             }
    }
```

”signal set”: przykład programu

```
else /* parent */
    {  /* pid  zawiera ID  potomka */
     sleep(6);
     printf("\nPARENT: wysyla sygnal SIGCONT\n\n\n");
     kill(pid,SIGCONT); /* uruchomienie */
     sleep(7);
     }
}/* koniec  main  */
            void sigcont()
           {
              printf("\n potomek: byl sygnal SIGCONT\n");
           }/*  koniec  funkcji   sigcont       */
```

sygnały: blokowanie a ignorowanie

na czym polega różnica pomiędzy **blokowaniem sygnału** a **ignorowaniem sygnału**?

(wyjaśniający przykład jest na slajdzie nr 63)