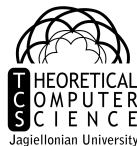


Systemy Operacyjne.

Jakub Kozik

semestr zimowy 2016/2017

Informatyka Analityczna
tcs@jagiellonian



Zasady

"This is not Nam. [...] There are rules."

Zaliczenie i egzamin

- 2 duże zadania + obrony projektów (2 x 30 punktów),
- 2 małe zadania (2 x 10 punktów),
- aktywność na ćwiczeniach (po 1 punkcie na każde ćwiczenia)
- egzamin pisemny (20 punktów).

Za każde z dużych zadań oraz za egzamin trzeba uzyskać przynajmniej 50% możliwych punktów.

Przeliczenie punktów na ocenę:

50-60 3.0; 60-70 3.5; 70-80 4.0; 80-90 4.5; 90-100 5.0

Zaliczenie w II-gim terminie

Po terminie oddania maksymalna liczba punktów za każde z zadań spada liniowo do 50% w ciągu dwóch tygodni.

Program Wykładu

- 1 POSIX - strona użytkownika.
- 2 MINIX - strona systemu.

Zagadnienia:

- Procesy.
- Wejście/Wyjście.
- Pamięć.
- System plików.

THE MINIX BOOK



Andrew S Tanenbaum, Albert S Woodhull,
Operating Systems Design and Implementation,
3rd Edition, Pearson Prentice Hall 2009

- Andrew S. Tanenbaum, **Systemy operacyjne**
 - Abraham Silberschatz, James L. Peterson, Peter B. Galvin, **Podstawy systemów operacyjnych**
- 1 <http://www.minix3.org/>
 - 2 POSIX.1-2008 - IEEE Std 1003.1™-2008 - The Open Group Technical Standard Base Specifications, Issue 7.

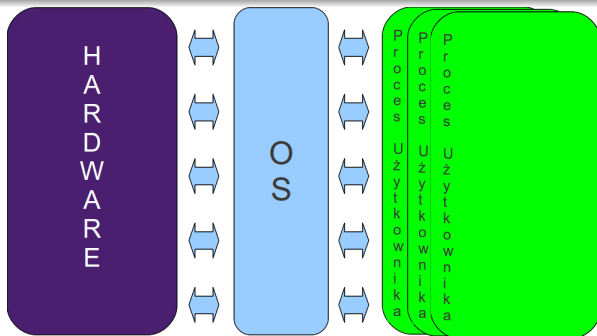
1 Zasady

2 POSIX

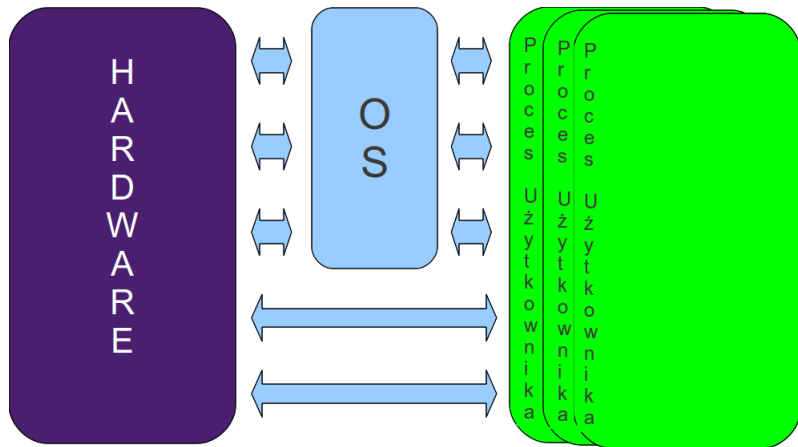
- Wstęp
- POSIX - standard
- POSIX - procesy
- POSIX - pliki
- POSIX - sygnały
- POSIX - remanent

Główne funkcje systemu.

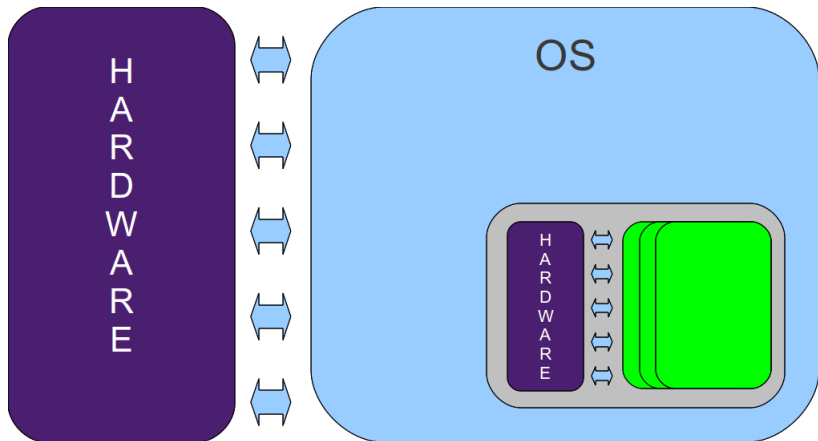
- Extended Machine
- Resource Management



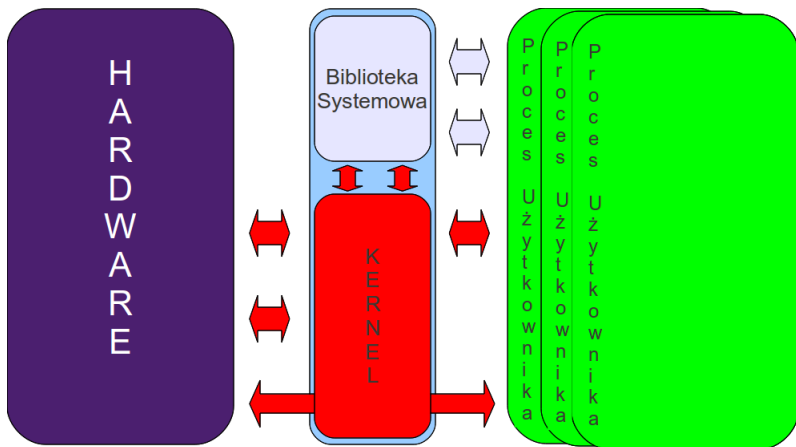
Bez zarządzania zasobami.



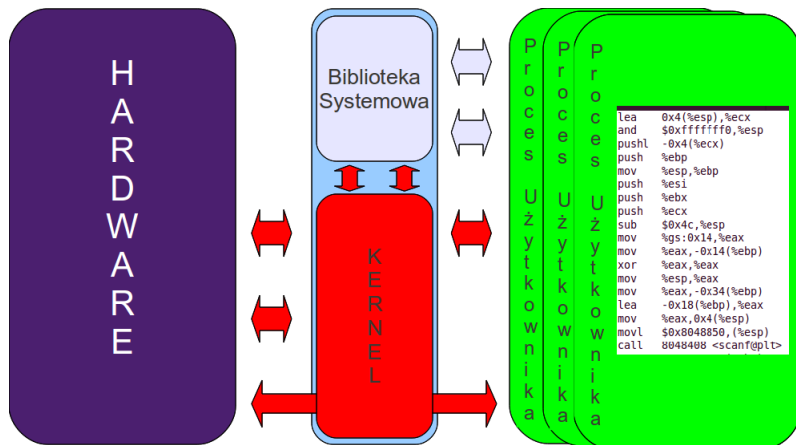
Wirtualna maszyna.



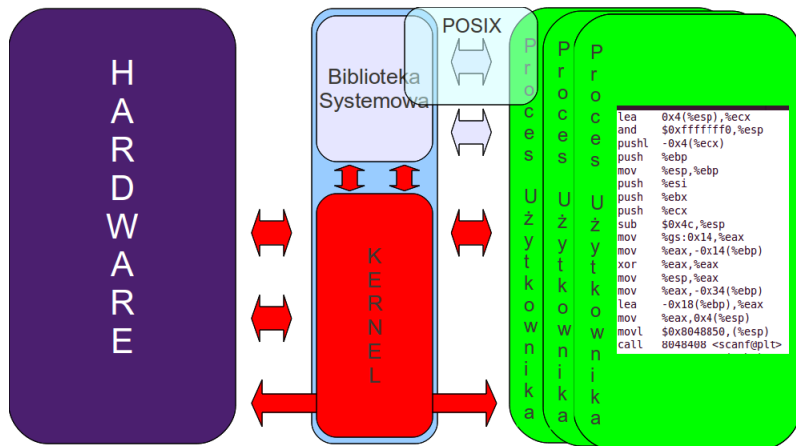
"Złoty środek."



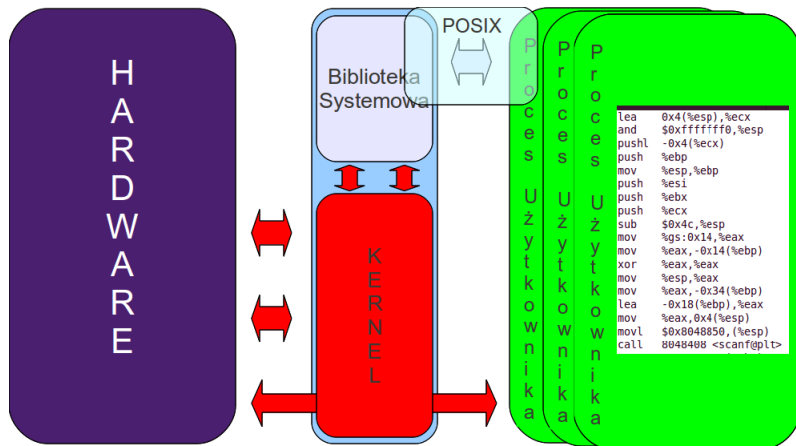
System calls - wywołania systemowe.



POSIX



POSIX programming.



1 Zasady

2 POSIX

- Wstęp
- **POSIX - standard**
- POSIX - procesy
- POSIX - pliki
- POSIX - sygnały
- POSIX - remanent

POSIX

Portable Operating System Interface

"**POSIX.1-2008** is simultaneously IEEE Std 1003.1TM-2008 and The Open Group Technical Standard Base Specifications, Issue 7."

POSIX principles:

- Application-Oriented
- Interface, Not Implementation
- Source, Not Object, Portability
- The C Language (ISO C)
- No Superuser, No System Administration
- Minimal Interface, Minimally Defined
- Broadly Implementable
- Minimal Changes to Historical Implementations
- Minimal Changes to Existing Application Code

IEEE Std. 1003.1-1990 Standard for Information Technology –
Portable Operating System Interface (POSIX) –
ART 1. System Application Programming Interface (API)
[C Language].

Donald Lewine, POSIX Programmers Guide, O'Reilly Media 1991

1 Zasady

2 POSIX

- Wstęp
- POSIX - standard
- **POSIX - procesy**
- POSIX - pliki
- POSIX - sygnały
- POSIX - remanent

Proces

Program w trakcie wykonywania.

Procesy - system calls

```
1 #include <stdio.h>
2
3 int
4 main( int argc , char *argv [])
5 {
6     printf( "Hey, _you _sass _that _hoopy _Ford _Prefect? \n" );
7 }
```

Procesy - system calls

```
1 #include <stdio.h>
2
3 int
4 main(int argc , char *argv[])
5 {
6     printf("Hey, _you _sass _that _hoopy _Ford _Prefect?\n");
7 }
```

exit

```
#include <unistd.h>
void _exit(int status);

#include <stdlib.h>
void exit(int status);
```

Procesy - system calls

Linux - x86

08048080 <_start>:

| | | |
|-------------------------|-----|------------------|
| 8048080: b8 04 00 00 00 | mov | \$0x4,%eax |
| 8048085: bb 01 00 00 00 | mov | \$0x1,%ebx |
| 804808a: b9 a0 90 04 08 | mov | \$0x80490a0,%ecx |
| 804808f: ba 06 00 00 00 | mov | \$0x6,%edx |
| 8048094: cd 80 | int | \$0x80 |
| 8048096: b8 01 00 00 00 | mov | \$0x1,%eax |
| 804809b: cd 80 | int | \$0x80 |

exit

```
#include <unistd.h>
void _exit(int status);
```

```
#include <stdlib.h>
void exit(int status);
```

fork

```
#include <unistd.h>
pid_t fork(void);
```

```
int
main(int argc, char *argv[])
{
    int k;

    printf("%d,%d\n", \
        getpid(), getppid());
    k= fork();
    printf("%d,%d,%d\n", \
        k, getpid(), getppid());
}
```

- unique process ID.
- different parent process ID
- own copy of the parent's descriptors.
- no pending signals, inactive alarm timer

Fork bomb.

```
int main(){
    while (1) fork();
}
```

execve

```
#include <unistd.h>
```

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

```
extern char **environ;
```

```
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
```

```
int execlp(const char *path, const char *arg0, ... /*,  
          (char *)0, char *const envp[] */);
```

```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
```

```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
```

```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
```

```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
```

```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
```

Deskryptory procesu wywołującego exec pozostają otwarte (domyślnie).

```
#include <unistd.h>
```

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

tic.c

tictac.c

```
1#include <stdio.h>
2
3int
4main(int argc, char *argv[])
5{
6    int i;
7
8    for (i=0; i<10; i++) {
9        printf("%s\n", argv[1]);
10       sleep(1);
11    }
12 }
```

```
1#include <unistd.h>
2
3int
4main(int argc, char *argv[])
5{
6    char* str;
7
8    if (fork()) str = "tic";
9    else str = "tac";
10
11    execl("tic", "tic", str, NULL);
12 }
```

waitpid

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

```
pid_t wait(int *stat_loc);
```

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

$$\text{wait(stat_loc)} \equiv \text{waitpid}(-1, \text{stat_loc}, 0)$$

waitpid

```
1#include <unistd.h>
2#include <stdio.h>
3#include <stdlib.h>
4#include <sys/types.h>
5#define BSIZE 100
6
7int main(){
8    char str[BSIZE];
9    pid_t chld_pid;
10
11    while (fgets(str,BSIZE,stdin)){
12        chld_pid = fork();
13        if (!chld_pid){
14            execlp("echo", "echo", str, NULL);
15            exit(1);
16        } else
17            waitpid(chld_pid, NULL, 0);
18    }
19 }
```

waitpid – exit(1) ???

```
1#include <unistd.h>
2#include <stdio.h>
3#include <stdlib.h>
4#include <sys/types.h>
5#define BSIZE 100
6
7int main(){
8    char str[BSIZE];
9    pid_t chld_pid;
10
11    while (fgets(str,BSIZE,stdin)){
12        chld_pid = fork();
13        if (!chld_pid){
14            execlp("echo", "echo", str, NULL);
15            exit(1);
16        } else
17            waitpid(chld_pid, NULL, 0);
18    }
19 }
```

1 Zasady

2 POSIX

- Wstęp
- POSIX - standard
- POSIX - procesy
- **POSIX - pliki**
- POSIX - sygnały
- POSIX - remanent

File Descriptor

“A **per-process unique, non-negative integer** used to identify an open file for the purpose of file access.

The value of a file descriptor is from zero to OPEN_MAX.”

limits.h

```
#define _POSIX_OPEN_MAX    16 /* a process may have 16 files open */  
...  
#define OPEN_MAX          20 /* # open files a process may have */
```

Open File Description

“A record of how a process or group of processes is accessing a file. Each file descriptor refers to exactly one open file description, but an open file description can be referred to by more than one file descriptor. The file offset, file status, and file access modes are attributes of an open file description.”

Deskryptory plików

Domyślnie otwarte deskryptory.

0 - stdin

1 - stdout

2 - stderr

open (zwraca deskryptor dla otwartego pliku)

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

```
#include <fcntl.h>
```

```
int open(const char *path, int flags [, mode_t mode]);
```

O_RDONLY open for reading only

O_WRONLY open for writing only

O_RDWR open for reading and writing

O_NONBLOCK do not block on open

O_APPEND append on each write

O_CREAT create file if it does not exist

O_TRUNC truncate size to 0

O_EXCL error if create and file exists

Semafor na plikach.

Atomic lock.

`(O_CREAT | O_EXCL)` - `open()` shall fail if the file exists.

```
#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
```

creat & close

close

```
#include <unistd.h>
```

```
int close(int d);
```

creat

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

```
#include <fcntl.h>
```

```
int creat(const char *name, mode_t mode)
```

```
creat(path, mode)  $\equiv$  open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

read

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

```
ssize_t read(int d, void *buf, size_t nbytes);
```

(zwraca liczbę przeczytanych byte'ów)

(0 \rightarrow EOF)

If a `read()` is interrupted by a signal before it reads any data, it shall return `-1` with `errno` set to `[EINTR]`.

If a `read()` is interrupted by a signal after it has successfully read some data, it shall return the number of bytes read.

write

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

```
ssize_t write(int d, const void *buf, size_t nbytes);
```

(zwraca liczbę zapisanych byte'ów)

If `write()` is interrupted by a signal before it writes any data, it shall return `-1` with `errno` set to `[EINTR]`.

If `write()` is interrupted by a signal after it successfully writes some data, it shall return the number of bytes written.

lseek

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

#define SEEK_SET 0    /* offset is absolute */
#define SEEK_CUR 1    /* relative to current position */
#define SEEK_END 2    /* relative to end of file */

off_t lseek(int d, off_t offset, int whence)
```

```
1#include <sys/stat.h>
2#include <fcntl.h>
3#include <unistd.h>
4
5int main(int argc, char* argv[]){
6    int fd=open("foo", O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
7
8    lseek(fd,10000000000L,SEEK_CUR); /*~10GB*/
9    write(fd,"a",1);
10   close(fd);
11 }
```

pipe

```
#include <unistd.h>
```

```
int pipe(int fildes[2])
```

```
1#include <stdio.h>
2
3int main(int argc, char* argv[]){
4    int fd[2];
5
6    if (pipe(fd) != 0) return 1;
7    if (fork()){
8        write(fd[1], "say_something", 13);
9    } else {
10        char buf[21];
11        int n;
12        if (n = read(fd[0], buf, 20) >= 4){
13            buf[n] = 0;
14            printf("%s\n", buf+4);
15        }
16    }
17    return 0;
18}
```

Named pipe - FIFO

mkfifo & mknod

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

```
int mknod(const char *path, mode_t mode, dev_t dev)
int mkfifo(const char *path, mode_t mode)
```

pipe

Mknod may be invoked only by the super-user, unless it is being used to create a fifo.

The call `mkfifo(path, mode)` is equivalent to

```
mknod(path, (mode & 0777) | S_IFIFO, 0)
```

Pipe r/w rules.

Bad news

The behavior of multiple concurrent **reads** on the same pipe, FIFO, or terminal device is **unspecified**.

From read() - rationale

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. It is a known attribute of terminals that this is not honored, and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (**or not**).

Pipe r/w rules.

Good news

Write requests of `PIPE_BUF` bytes or less shall not be interleaved with data from other processes doing writes on the same pipe.

Writes of greater than `PIPE_BUF` bytes may have data interleaved, on arbitrary boundaries, with writes by other processes

```

1#include <sys/stat.h>
2#include <string.h>
3#include <stdio.h>
4int main(){
5    int fd[2],n;
6    char buf[4];
7    pipe(fd);
8    if (!fork()) {
9        while ((n = read(fd[0], buf, 3))>0){
10            buf[n]=0;
11            printf("%s\n", buf);
12        }
13    } else {
14        sleep(1);
15        if (fork()) strcpy(buf,"tic");
16        else strcpy(buf,"tac");
17
18        for (n=0; n<10; n++){
19            write(fd[1],buf,3);
20            sleep(1);
21        }
22    }
23}

```

fcntl - file descriptor control functions

```
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, [data])
```


fcntl(fd, F_DUPFD, int fd2)

```
1#include <fcntl.h>
2#include <unistd.h>
3
4int main(){
5    int fd[2];
6    pipe(fd);
7    if (!fork()) {
8        close(0);
9        close(fd[1]);
10       fcntl(fd[0], F_DUPFD, 0);
11       execlp("cat", "cat", NULL);
12    } else {
13        write(fd[1], "say_hello\n", 10);
14        close(fd[1]);
15        wait(NULL);
16    }
17 }
```

fcntl(fd, F_GETFD, int fd2) - fd flags

```
1#include <fcntl.h>
2#include <unistd.h>
3
4int main(){
5    int fd[2];
6    pipe(fd);
7
8    int flags=fcntl(fd[1],F_GETFD);
9    flags|=FD_CLOEXEC;
10   fcntl(fd[1],F_SETFD, flags);
11
12   if (!fork()) {
13       close(0);
14       /* close(fd[1]); */
15       fcntl(fd[0], F_DUPFD,0);
16       execlp("cat","cat",NULL);
17   } else {
18       write(fd[1],"say_hello\n",10);
19       close(fd[1]);
20   }
21 }
```

`fcntl(fd, F_GETFL, int fd2)` - file status flags

`fcntl(fd, F_GETFL)`

Return the file status flags and file access modes associated with the file associated with file descriptor `fd`.

`fcntl(fd, F_SETFL, int flags)`

Set the file status flags of the file referenced by `fd` to `flags`. Only `O_NONBLOCK` and `O_APPEND` may be changed. Access mode flags are ignored.

```

1#include <fcntl.h>
2#include <errno.h>
3#include <unistd.h>
4int main(){
5    int fd[2],n;
6    pipe(fd);
7    int flags=fcntl(fd[0],F_GETFL);
8    fcntl(fd[0],F_SETFL, flags | O_NONBLOCK);
9    if (!fork()) {
10        char buf[20];
11        close(fd[1]);
12        while ((n=read(fd[0],buf,20))!=0){
13            if (n>0) write(0,buf,n);
14            else if (errno!=EAGAIN) return 1;
15            else write(0," still nothing\n",14);
16            sleep(1);
17        }
18    } else
19    for (n=0; n<5; n++){
20        sleep(3);
21        write(fd[1]," I am a walrus.\n",16);
22    }
23 }

```

O_NONBLOCK for open

```
1#include <fcntl.h>
2#include <errno.h>
3#include <unistd.h>
4#include <sys/stat.h>
5#include <stdio.h>
6
7int main(){
8    int fd,n; char buf[20];
9    mkfifo("mfifo", S_IWUSR | S_IRUSR);
10
11    if (!fork()) {
12        fd=open("mfifo",O_RDONLY|O_NONBLOCK);
13        write(1,"opened\n",7);
14        sleep(10);
15        while ((n=read(fd,buf,20))>0)
16            write(1,buf,n);
17    } else{
18        sleep(5);
19        fd=open("mfifo",O_WRONLY);
20        write(fd,"hello\n",6);
21        write(1,"done\n",5);
22    }
23 }
```

Advisory record locking.

```
fcntl(fd, F_GETLK, struct flock *lkp)
```

Find out if some other process has a lock on a segment of the file associated by file descriptor `fd` that overlaps with the segment described by the `flock` structure pointed to by `lkp`. [...]

```
fcntl(fd, F_SETLK, struct flock *lkp)
```

Register a lock on a segment of the file associated with file descriptor `fd`. [...] This call returns an error if any part of the segment is already locked.

```
fcntl(fd, F_SETLKW, struct flock *lkp)
```

Register a lock on a segment of the file associated with file descriptor `fd`. [...] This call blocks waiting for the lock to be released if any part of the segment is already locked.

```
struct flock {  
    short    l_type;        /* F_RDLCK, F_WRLCK, or F_UNLCK */  
    short    l_whence;      /* SEEK_SET, SEEK_CUR, or SEEK_END */  
    off_t    l_start;       /* byte offset to start of segment */  
    off_t    l_len;         /* length of segment */  
    pid_t    l_pid;         /* process id of the locks' owner */  
};
```

```
1#include <fcntl.h>
2#include <unistd.h>
3#include <sys/stat.h>
4int main(int argc, char * argv[]){
5    int fd;
6    struct flock fl;
7
8    fd= open("lock", O_CREAT | O_RDWR, S_IWUSR | S_IRUSR );
9    /* ... */
10    fl.l_type = F_WRLCK;
11    fl.l_whence = SEEK_SET;
12    fl.l_start = 0;
13    fl.l_len = 3;
14
15    fcntl(fd, F_SETLKW, &fl);
16    lseek(fd, 0, SEEK_SET);
17    write(fd, argv[1], 3);
18    sleep(30);
19    fl.l_type = F_UNLCK;
20    fcntl(fd, F_SETLK, &fl);
21    /* ... */
22    close(fd);
23 }
```


1 Zasady

2 POSIX

- Wstęp
- POSIX - standard
- POSIX - procesy
- POSIX - pliki
- **POSIX - sygnały**
- POSIX - remanent

Sygnały

Sygnał

Informacja o **asynchronicznym** zdarzeniu/błędzie.

Ctrl-c

Ctrl-c powoduje wysłanie sygnału SIGINT do wszystkich procesów z *foreground process group*.

Dzielenie przez 0

Dzielenie liczby (int) przez (int) 0 powoduje wysłanie sygnału SIGFPE do procesu.

Źródła sygnałów.

Terminal Ctrl-C SIGINT,
Ctrl-\SIGQUIT

Hardware dzielenie przez 0 SIGFPE,
niewłaściwe odwołanie do pamięci SIGSEGV,...

Proces syscall kill, domyślny sygnał SIGTERM

System - Software conditions SIGALARM,
SIGPIPE (broken pipe)

Sygnały które nie docierają do adresata

SIGKILL

SIGSTOP

Wysyłanie sygnałów.

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

Permission

...the real or effective user ID of the sending process shall match the real or saved set-user-ID of the receiving process.

Adresaci - pod warunkiem że można do nich wysłać

`pid > 0` proces, którego ID jest równe `pid`

`pid = 0` procesy z tej samej grupy

`pid = -1` wszystkie procesy

`pid < -1` wszystkie procesy z grupy o ID równym `|pid|`

```
int raise(int sig);
```

Obsługa sygnałów - ISO C

```
#include <signal.h>
```

```
void (*signal(int signo, void (*func)(int)))(int);
```

```
typedef void Sigfunc(int);
```

```
Sigfunc *signal(int, Sigfunc *);
```

Obsługa sygnałów

SIG_DFL domyślna obsługa sygnału

SIG_IGN sygnał jest ignorowany

wskaźnik do funkcji która ma obsłużyć sygnał

Znikające i nieobsłużone sygnały.

```
1#include <stdio.h>
2#include <signal.h>
3
4void handler(int sig_nb){
5    write(1,"If everything seems under control ,\
6you're just not going fast enough.\n",70);
7    sleep(1);
8    signal(SIGINT, handler);
9}
10
11int main(){
12    signal(SIGINT, handler);
13
14    while (1)
15        pause();
16}
```

Obsługa sygnałów - POSIX

```
#include <signal.h>
```

```
int sigaction(int sig, const struct sigaction *restrict act,  
              struct sigaction *restrict oact);
```

| | | |
|----------|---|--|
| void | (*sa_handler)(int) | Pointer to a signal-catching function or one of the SIG_IGN or SIG_DFL. |
| sigset_t | sa_mask | Set of signals to be blocked during execution of the signal handling function. |
| int | sa_flags | Special flags. |
| void | (*sa_sigaction)(int, siginfo_t *, void *) | Pointer to a signal-catching function. |

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Signal mask for the duration of the signal-catching function

This mask is formed by taking the union of the current signal mask and the value of the `sa_mask` for the signal being delivered, and unless `SA_NODEFER` or `SA_RESETHAND` is set, then including the signal being delivered.


```

1 #include <unistd.h>
2 #include <sys/types.h>
3 #include <signal.h>
4
5 volatile int ready=0;
6 void handler(int sig_nb){
7     ready=1;
8 }
9
10 int main(){
11     pid_t other;
12     char* str="tic\n";
13     struct sigaction act;
14
15     act.sa_handler= handler;
16     act.sa_flags=0;
17     sigemptyset(&act.sa_mask);
18     sigaction(SIGUSR1,&act ,NULL);
19
20     if (!(other=fork())){
21         str="tac\n";
22         other= getpid();
23     } else ready=1;
24
25     while (1) {
26         if (ready){
27             ready = 0;
28             sleep(1);
29             write(1,str,4);
30             kill(other,SIGUSR1);
31         }
32         pause();
33     }
34 }

```

Flaga SA_SIGINFO

If SA_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:

```
void func(int signo, siginfo_t *info, void *context);
```

info the reason why the signal was generated;

context the receiving thread's context that was interrupted when the signal was delivered.

Syscallle przerwane sygnałami.

read - przypomnienie

If a `read()` is interrupted by a signal before it reads any data, it shall return `-1` with `errno` set to `[EINTR]`.

If a `read()` is interrupted by a signal after it has successfully read some data, it shall return the number of bytes read.

write - przypomnienie

If `write()` is interrupted by a signal before it writes any data, it shall return `-1` with `errno` set to `[EINTR]`.

If `write()` is interrupted by a signal after it successfully writes some data, it shall return the number of bytes written.

```

1 #include <unistd.h>
2 #include <sys/types.h>
3 #include <signal.h>
4 #include <errno.h>
5
6 #define BSIZE 100
7 void handler(int sig_nb){}
8
9 int main(){
10     int n,k,w;
11     char buf[BSIZE];
12     pid_t parent, child;
13     struct sigaction act;
14
15     act.sa_handler= handler;
16     act.sa_flags=0;
17     sigemptyset(&act.sa_mask);
18     sigaction(SIGUSR1,&act ,NULL);
19
20     if (!(child=fork())){
21         parent= getppid();
22         while (1) kill(parent ,SIGUSR1);
23         exit(1);
24     }
25
26     while (n = read(0,buf, BSIZE)){
27         if ((n<0) && (errno!=EINTR)) break;
28         k=0;
29         while(k<n){
30             w= write( 1, buf+k,n-k);
31             if ((w<0) && (errno!=EINTR)) goto end;
32             if (w>0) k+=w;
33         }
34     }
35 end:    kill(child , SIGTERM);
36 }

```

SA_RESTART

If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with [EINTR] unless otherwise specified.

Przykłady

read, write, open, waitpid, fcntl (F_SETLKW)

```

1 #include <unistd.h>
2 #include <sys/types.h>
3 #include <signal.h>
4 #include <errno.h>
5
6 #define BSIZE 100
7 void handler(int sig_nb){}
8
9 int main(){
10     int n,k,w;
11     char buf[BSIZE];
12     pid_t parent, child;
13     struct sigaction act;
14
15     act.sa_handler= handler;
16     act.sa_flags=SA_RESTART;
17     sigemptyset(&act.sa_mask);
18     sigaction(SIGUSR1,&act ,NULL);
19
20     if (!(child=fork())){
21         parent= getppid();
22         while (1) kill(parent ,SIGUSR1);
23         exit(1);
24     }
25
26     while ((n = read(0,buf , BSIZE))>0){
27         k=0;
28         while(k<n){
29             w= write( 1, buf+k,n-k);
30             if (w<0) goto end;
31             k+=w;
32         }
33     }
34 end:    kill(child , SIGTERM);
35 }

```

UWAGA na errno!

```
1 #include <unistd.h>
2 #include <sys/types.h>
3 #include <signal.h>
4 #include <errno.h>
5
6 #define BSIZE 100
7 void handler(int sig_nb){ errno =0;}
8
9 int main(){
10     int n,k,w;
11     char buf[BSIZE];
12     pid_t parent, child;
13     struct sigaction act;
14
15     act.sa_handler= handler;
16     act.sa_flags=0;
17     sigemptyset(&act.sa_mask);
18     sigaction(SIGUSR1,&act ,NULL);
19
20     if (!(child=fork())){
21         parent= getppid();
22         while (1) kill(parent ,SIGUSR1);
23         exit(1);
24     }
25
26     while (n = read(0,buf , BSIZE)){
27         if ((n<0) && (errno!=EINTR)) break; else n=0;
28         write( 1, buf,n);    // nie dbamy o przerwane write'y
29     }
30 end:  kill(child , SIGTERM);
31 }
```

Flaga SA_NOCLDSTOP

SIGCHLD

Child process terminated, stopped, or continued.

SA_NOCLDSTOP

Do not generate SIGCHLD when children stop or stopped children continue.

Uwaga

If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is unspecified.

Normalne sygnały NIE są kolejkowane!

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <sys/types.h>
5 #include <signal.h>
6
7 volatile int s=0;
8 void handler(int sig_nb){ s++; }
9
10 int main(){
11     int n,k,w;
12     pid_t parent;
13     struct sigaction act;
14
15     act.sa_handler= handler;
16     act.sa_flags=0;
17     sigemptyset(&act.sa_mask);
18     sigaction(SIGUSR1,&act ,NULL);
19
20     if (!fork()){
21         parent= getppid();
22         for (n=0; n<10; n++) kill(parent ,SIGUSR1);
23         exit(0);
24     }
25
26     pause();
27     while (s-- >0) {
28         printf("received\n");
29         sleep(3);
30     }
31 }
```

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <signal.h>
6
7 #define CHILDREN 10
8 volatile int z=CHILDREN;
9 void handler(int sig_nb){
10     pid_t child;
11     do{
12         child = waitpid(-1,NULL,WNOHANG);
13         if (child>0 ) z--;
14     } while (child >0);
15     sleep(1);
16 }
17
18 int main(){
19     int n;
20     struct sigaction act;
21     act.sa_handler= handler;
22     act.sa_flags=0;
23     sigemptyset(&act.sa_mask);
24     sigaction(SIGCHLD,&act ,NULL);
25
26     for (n=0; n<CHILDREN ; n++)
27         if (!fork()) {
28             sleep(n);
29             return 0;
30         }
31
32     while (z >0) {
33         printf("%d\children/zombies_left.\n",z);
34         sleep(1);
35     }
36     printf("No_more_zombies.\n" );
37 }

```

SIG_IGN dla SIGCHLD

```
1 #include <unistd.h>
2 #include <signal.h>
3
4 #define CHILDREN 10
5
6 int main(){
7     int n;
8     struct sigaction act;
9     act.sa_handler= SIG_IGN;
10    act.sa_flags=0;
11    sigemptyset(&act.sa_mask);
12    sigaction(SIGCHLD,&act ,NULL);
13
14    for (n=0; n<CHILDREN ; n++)
15        if (!fork()) {
16            return 0;
17        }
18    sleep(10);
19 }
```

LINUX

Ignoring SIGCHLD can be used to prevent the creation of zombies.

async-signal-safe functions

safe

_exit, close, kill, read, write, ...

unsafe

malloc, exit, printf ...

alarm() function → SIGALRM signal

```
#include <unistd.h>
```

```
unsigned alarm(unsigned seconds);
```

Co może pójść źle w poniższym programie?

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <signal.h>
4
5 #define TIME 5
6 void handler(int sig_nb){ }
7
8 int main(){
9     int n;
10    struct sigaction act;
11
12    act.sa_handler= handler;
13    act.sa_flags=0;
14    sigemptyset(&act.sa_mask);
15    sigaction(SIGALRM,&act ,NULL);
16
17    alarm(TIME);
18    pause();
19    printf("No_time_..._no_time_to_lose.\n");
20 }
```

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

how values

SIG_BLOCK The resulting set shall be the union of the current set and the signal set pointed to by set.

SIG_SETMASK The resulting set shall be the signal set pointed to by set.

SIG_UNBLOCK The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by set.

alarm()

Trochę lepsze rozwiązanie.

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <signal.h>
4
5 #define TIME 5
6 void handler(int sig_nb){ }
7
8 int main(){
9     int n;
10    struct sigaction act;
11    sigset_t mask;
12
13
14    act.sa_handler= handler;
15    act.sa_flags=0;
16    sigemptyset(&act.sa_mask);
17    sigaction(SIGALRM,&act ,NULL);
18    sigaction(SIGINT,&act ,NULL);
19
20    sigfillset(&mask);
21    sigdelset(&mask,SIGALRM);
22
23    sigprocmask(SIG_SETMASK, &mask, NULL);
24    alarm(TIME);
25    pause();
26    printf("No_time..._no_time_to_lose.\n");
27 }
```

sigsuspend

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *sigmask);
```

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <signal.h>
4 #define TIME 5
5 void handler(int sig_nb){ }
6
7 int main(){
8     struct sigaction act;
9     sigset_t mask;
10
11     act.sa_handler= handler;
12     act.sa_flags=0;
13     sigemptyset(&act.sa_mask);
14     sigaction(SIGALRM,&act ,NULL);
15     sigaction(SIGINT,&act ,NULL);
16
17     sigemptyset(&mask);
18     sigaddset(&mask,SIGALRM);
19     sigprocmask(SIG_BLOCK, &mask, NULL);
20
21     sigfillset(&mask);
22     sigdelset(&mask, SIGALRM);
23     alarm(TIME);
24     sigsuspend(&mask);
25     printf("No time to lose.\n");
26 }
```



```
#include <signal.h>

int sigpending(sigset_t *set);
```

The `sigpending()` function shall store, in the location referenced by the `set` argument, the set of signals that are blocked from delivery to the calling thread and that are pending on the process or the calling thread.

read with timeout - prawie poprawne

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <signal.h>
4
5 volatile int time_is_up;
6 void handler(int sig_nb){ time_is_up=1;}
7
8 int tread(char * buf, int n, int timeout){
9     int r;
10    struct sigaction act, oact;
11    sigset_t mask, omask;
12
13    act.sa_handler= handler;
14    act.sa_flags=0;
15    sigemptyset(&act.sa_mask);
16    sigaction(SIGALRM,&act ,&oact);
17
18    time_is_up=0;
19    alarm(timeout);
20    do r= read(0, buf, n);
21    while ((!time_is_up) && (r<0));
22    alarm(0);
23
24    sigaction(SIGALRM,&oact ,NULL);
25    if (r<0) return 0;
26    return r;
27 }
28 int main(){
29     char buf[100];
30     int n= tread(buf,100,5);
31     write(1,buf,n);
32 }
```

select

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

int select(int nfds, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict errorfds,
           struct timeval *restrict timeout);

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

Upon successful completion, the `pselect()` or `select()` function shall modify the objects pointed to by the `readfds`, `writefds`, and `errorfds` arguments to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively, and shall return the total number of ready descriptors in all the output sets.

```
struct timeval {
    long    tv_sec;           /* seconds */
    long    tv_usec;         /* microseconds */
};
```

read with timeout - poprawne

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <errno.h>
4 #include <sys/select.h>
5 #include <sys/time.h>
6
7 int tread(char * buf, int n, int timeout){
8     int r;
9     fd_set rfd;
10    struct timeval timeout_s;
11
12    FD_ZERO(&rfd);
13    FD_SET(0,&rfd);
14
15    timeout_s.tv_sec=timeout;
16    timeout_s.tv_usec=0;
17
18    do r= select(1, &rfd, NULL, NULL, &timeout_s);
19    while ((r<0) && (errno== EINTR));
20
21    if (r<=0) return 0;
22
23    return (read(0, buf, n));
24 }
25
26 int main(){
27     char buf[100];
28     int n= tread(buf,100,5);
29     write(1,buf,n);
30 }
```

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

int pselect(int nfd, fd_set *restrict readfds,
            fd_set *restrict writefds, fd_set *restrict errorfds,
            const struct timespec *restrict timeout,
            const sigset_t *restrict sigmask);
```

1 Zasady

2 POSIX

- Wstęp
- POSIX - standard
- POSIX - procesy
- POSIX - pliki
- POSIX - sygnały
- **POSIX - remanent**

Procesy

`getpriority, setpriority` - get and set scheduling priority
`setsid, getpgrp` - create process group, get process group id
`setuid, setgid` - set user or group ID's
`brk, sbrk` - change data segment size

File System

```
access  - determine accessibility of file
chmod   - change mode of file
chown   - change owner and group of a file
link    - make a hard link to a file
mkdir   - make a directory file
mount, umount - mount or umount a file system
rename  - change the name of a file
rmdir   - remove a directory file
stat, lstat, fstat - get file status
sync, fsync - update dirty buffers and super-block
unlink  - remove directory entry
umask   - set file creation mode mask
utime   - set file times
```


Info

```
gettimeofday - get date and time  
getuid, geteuid - get user identity  
time, stime - get/set date and time  
times - get process times  
uname - get system info
```

Inne

- `chroot` - change root directory
- `ptrace` - process trace
- `reboot` - close down the system or reboot
- `mmap` - request memory mapping