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

Program Wykładu

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

Zagadnienia:

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

Literatura

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**

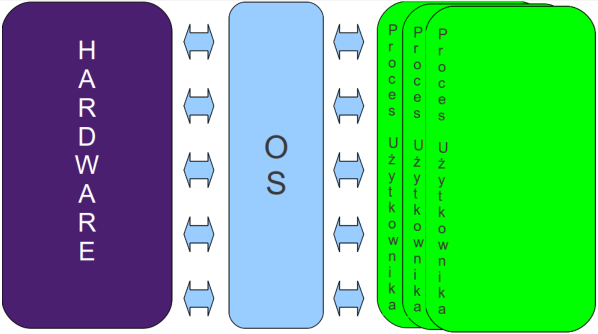
1. http://www.minix3.org/
2. POSIX.1-2008 - IEEE Std 1003.1$^{\mathrm{TM}}$-2008 - The Open Group Technical Standard Base Specifications, Issue 7.

Outline

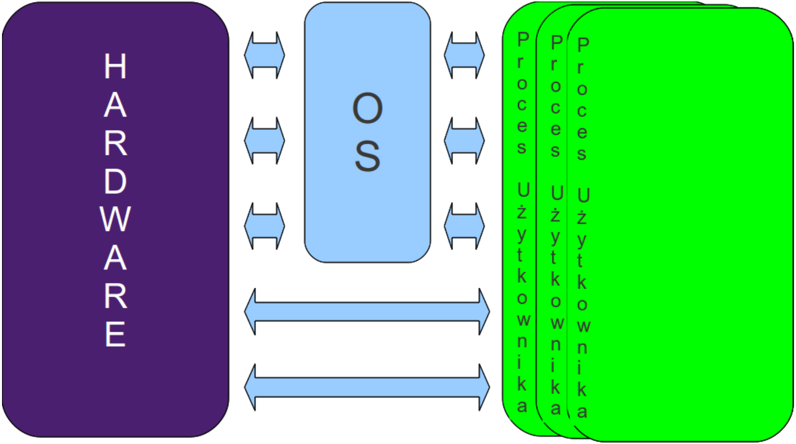
System Operacyjny

Główne funkcje systemu.

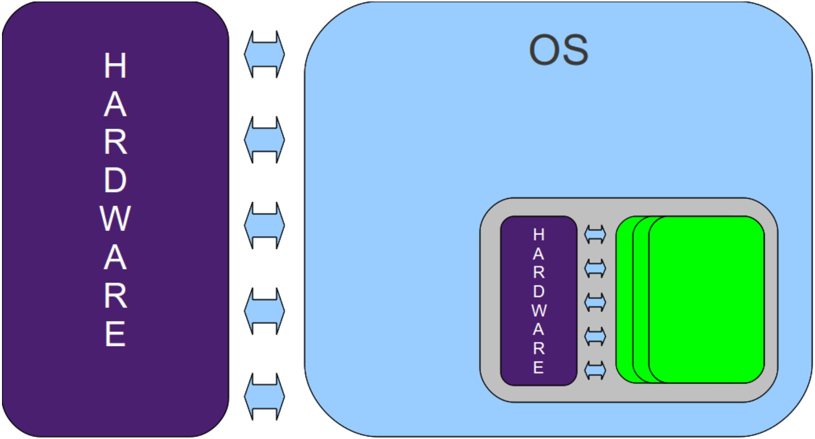
- Extended Machine
- Resource Management



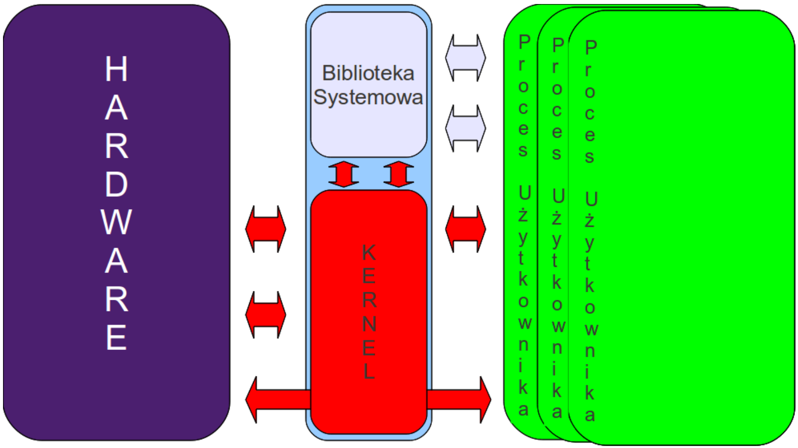
Bez zarządzania zasobami.



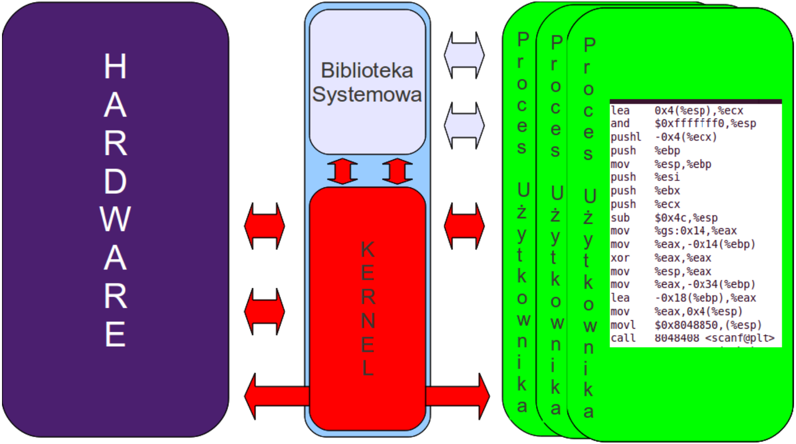
Wirtualna maszyna.



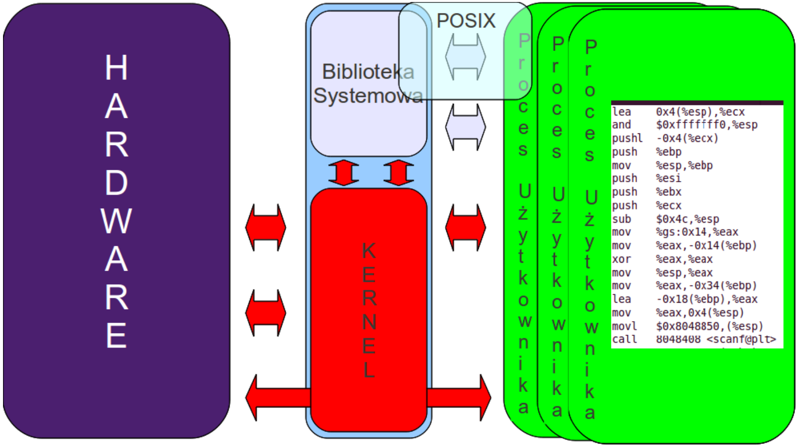
"Złoty środek."



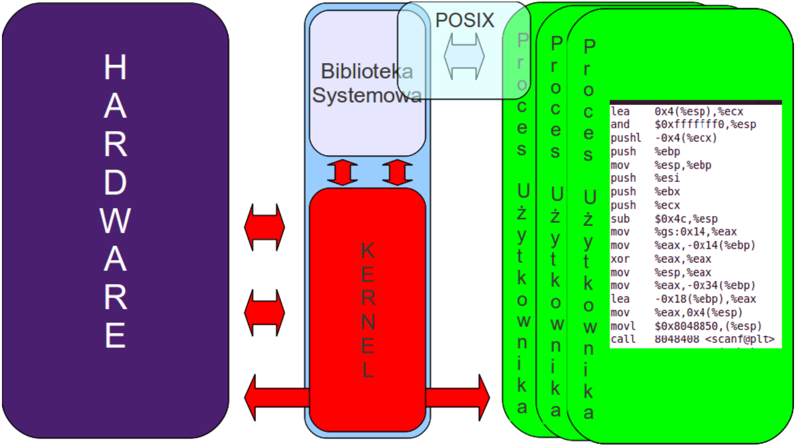
System calls - wywołania systemowe.



POSIX


HARDWARE
Biblioteka Systemowa
POSIX
KERNEL
Proces Użytkownika
Proces Użytkownika
Proces Użytkownika
lea 0x4(%esp),%ecx
and $0xfffffff0,%esp
pushl -0x4(%ecx)
push %ebp
mov %esp,%ebp
push %esi
push %ebx
push %ecx
sub $0x4c,%esp
mov %gs:0x14,%eax
mov %eax,-0x14(%ebp)
xor %eax,%eax
mov %esp,%eax
mov %eax,-0x34(%ebp)
lea -0x18(%ebp),%eax
mov %eax,0x4(%esp)
movl $0x8048850,(%esp)
call 8048408 <scanf@plt>

POSIX programming.



Outline

POSIX

Portable Operating System Interface

"**POSIX.1-2008** is simultaneously IEEE Std 1003.1^{TM}-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

POSIX.1-1990

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

Outline

POSIX - Procesy

Proces

Program w trakcie wykonywania.

Procesy - system calls

```c
#include <stdio.h>

int
main(int argc, char *argv[])
{
  printf("Hey, you sass that hoopy Ford Prefect?\n");
}
```

Procesy - system calls

```c
#include <stdio.h>

int
main(int argc, char *argv[])
{
    printf("Hey, you sass that hoopy Ford Prefect?\n");
}
```

exit

```c
#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();
}
```

exec

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 execle(const char *path, const char *arg0, ... /*,
       (char *)0, char *const envp[]*/);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

Deskryptory procesu wywołującego exec pozostają otwarte (domyślnie).

exec

```
#include <unistd.h>

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

tic.c

```c
#include <stdio.h>

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

  for (i=0;i<10;i++) {
    printf("%s\n",argv[1]);
    sleep(1);
  }
}
```

tictac.c

```c
#include <unistd.h>

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

  if (fork()) str = "tic";
  else str = "tac";

  execl("tic","tic",str,NULL);
}
```

waitpid

waitpid

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

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

$$\texttt{wait(stat\_loc)} \equiv \texttt{waitpid(-1, stat\_loc, 0)}$$

waitpid

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#define BSIZE 100

int main(){
  char str[BSIZE];
  pid_t chld_pid;

  while (fgets(str,BSIZE,stdin)){
    chld_pid = fork();
    if (!chld_pid){
      execlp("echo","echo",str,NULL);
      exit(1);
    } else
      waitpid(chld_pid,NULL,0);
  }
}
```

waitpid – exit(1) ???

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#define BSIZE 100

int main(){
  char str[BSIZE];
  pid_t chld_pid;

  while (fgets(str,BSIZE,stdin)){
    chld_pid = fork();
    if (!chld_pid){
      execlp("echo","echo",str,NULL);
      exit(1);
    } else
      waitpid(chld_pid,NULL,0);
  }
}
```

Outline

Deskryptory plików

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)`

Czytanie.

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.
```

Pisanie.

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)
```

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char* argv[]){
  int fd=open("foo", O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);

  lseek(fd,10000000000L, SEEK_CUR); /*~10GB*/
  write(fd, "a", 1);
  close(fd);
}
```

pipe

```
#include <unistd.h>

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

```c
#include <stdio.h>

int main(int argc,char* argv[]){
  int fd[2];

  if (pipe(fd) != 0) return 1;
  if (fork()){
    write(fd[1],"say something",13);
  }else{
    char buf[21];
    int n;
    if (n = read(fd[0], buf,20) >= 4){
      buf[n] = 0;
      printf("%s\n",buf+4);
    }
  }
  return 0;
}
```

Named pipe - FIFO

mkfifo & mknode

```
#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

```c
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
int main(){
  int fd[2],n;
  char buf[4];
  pipe(fd);
  if (!fork()) {
    while ((n = read(fd[0], buf, 3))>0){
      buf[n] = 0;
      printf("%s\n", buf);
    }
  } else {
    sleep(1);
    if (fork()) strcpy(buf, "tic");
    else strcpy(buf, "tac");

    for (n=0; n<10; n++){
      write(fd[1], buf, 3);
      sleep(1);
    }
  }
}
```

fcntl - file descriptor control functions

```
#include <fcntl.h>

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

fcntl(fd, F_DUPFD, int fd2)

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

int main(){
  int fd[2];
  pipe(fd);
  if (!fork()) {
    close(0);
    close(fd[1]);
    fcntl(fd[0], F_DUPFD,0);
    execlp("cat", "cat", NULL);
  } else {
    write(fd[1], "say hello\n", 10);
    close(fd[1]);
    wait(NULL);
  }
}
```

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

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

int main(){
  int fd[2];
  pipe(fd);

  int flags = fcntl(fd[1],F_GETFD);
  flags |= FD_CLOEXEC;
  fcntl(fd[1],F_SETFD,flags);

  if (!fork()) {
    close(0);
/*    close(fd[1]); */
    fcntl(fd[0],F_DUPFD,0);
    execlp("cat","cat",NULL);
  } else {
    write(fd[1],"say hello\n",10);
    close(fd[1]);
  }
}
```

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 descrip-
          tor 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.
```

```c
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
int main(){
  int fd[2],n;
  pipe(fd);
  int flags=fcntl(fd[0], F_GETFL);
  fcntl(fd[0], F_SETFL, flags | O_NONBLOCK);
  if (!fork()) {
    char buf[20];
    close(fd[1]);
    while ((n=read(fd[0],buf,20))!=0){
      if (n>0) write(0,buf,n);
      else if (errno!=EAGAIN) return 1;
      else write(0,"still nothing\n",14);
      sleep(1);
    }
  } else
  for (n=0; n<5; n++){
    sleep(3);
    write(fd[1],"I am a walrus.\n",16);
  }
}
```

O_NONBLOCK for open

```c
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdio.h>

int main(){
  int fd,n; char buf[20];
  mkfifo("mfifo", S_IWUSR | S_IRUSR);

  if (!fork()) {
    fd=open("mfifo", O_RDONLY|O_NONBLOCK);
    write(1, "opened\n", 7);
    sleep(10);
    while ((n=read(fd, buf, 20))>0)
      write(1, buf, n);
  } else{
    sleep(5);
    fd=open("mfifo", O_WRONLY);
    write(fd, "hello\n", 6);
    write(1, "done\n", 5);
  }
}
```

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 over-
     laps 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 */
};
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int main(int argc, char * argv[]){
  int fd;
  struct flock fl;

  fd= open("lock", O_CREAT | O_RDWR, S_IWUSR | S_IRUSR );
  /*...*/
  fl.l_type = F_WRLCK;
  fl.l_whence = SEEK_SET;
  fl.l_start = 0;
  fl.l_len = 3;

  fcntl(fd,F_SETLKW, &fl);
  lseek(fd,0,SEEK_SET);
  write(fd,argv[1],3);
  sleep(30);
  fl.l_type = F_UNLCK;
  fcntl(fd,F_SETLK, &fl);
  /*...*/
  close(fd);
}
```

Outline

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 wysyłać

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.

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

void handler(int sig_nb){
  write(1,"If everything seems under control, \
you're just not going fast enough.\n",70);
  sleep(1);
  signal(SIGINT,handler);
}

int main(){
  signal(SIGINT,handler);

  while (1)
    pause();
}
```

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.
```

sigset_t

```
#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.

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

volatile int ready = 0;
void handler(int sig_nb){
  ready = 1;
}

int main(){
  pid_t other;
  char* str="tic\n";
  struct sigaction act;

  act.sa_handler= handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGUSR1,&act,NULL);

  if (!(other=fork())){
    str = "tac\n";
    other= getppid();
  } else ready = 1;

  while (1) {
    if (ready){
      ready = 0;
      sleep(1);
      write(1,str,4);
      kill(other,SIGUSR1);
    }
    pause();
  }
}
```

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.

Syscalle 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.`

```c
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <errno.h>

#define BSIZE 100
void handler(int sig_nb){}

int main(){
  int n,k,w;
  char buf[BSIZE];
  pid_t parent, child;
  struct sigaction act;

  act.sa_handler = handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGUSR1, &act, NULL);

  if (!(child = fork())){
    parent = getppid();
    while (1) kill(parent, SIGUSR1);
    exit(1);
  }

  while (n = read(0, buf, BSIZE)){
    if ((n<0) && (errno!=EINTR)) break;
    k = 0;
    while(k<n){
      w = write(1, buf+k, n-k);
      if ((w<0) && (errno!=EINTR)) goto end;
      if (w>0) k+=w;
    }
  }
end:  kill(child, SIGTERM);
}
```

Flaga SA_RESTART

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)`

```c
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <errno.h>

#define BSIZE 100
void handler(int sig_nb){}

int main(){
  int n,k,w;
  char buf[BSIZE];
  pid_t parent, child;
  struct sigaction act;

  act.sa_handler = handler;
  act.sa_flags = SA_RESTART;
  sigemptyset(&act.sa_mask);
  sigaction(SIGUSR1, &act, NULL);

  if (!(child = fork())){
    parent = getppid();
    while (1) kill(parent, SIGUSR1);
    exit(1);
  }

  while ((n = read(0, buf, BSIZE))>0){
    k = 0;
    while(k<n){
      w = write(1, buf+k, n-k);
      if (w<0) goto end;
      k += w;
    }
  }
end:  kill(child, SIGTERM);
}
```

UWAGA na errno!

```c
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <errno.h>

#define BSIZE 100
void handler(int sig_nb){ errno = 0;}

int main(){
  int n,k,w;
  char buf[BSIZE];
  pid_t parent, child;
  struct sigaction act;

  act.sa_handler= handler;
  act.sa_flags=0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGUSR1, &act, NULL);

  if (!(child = fork())){
    parent = getppid();
    while (1) kill(parent,SIGUSR1);
    exit(1);
  }

  while (n = read(0,buf, BSIZE)){
    if ((n<0) && (errno != EINTR)) break; else n = 0;
    write(1, buf, n);    // nie dbamy o przerwane write'y
  }
end:  kill(child, SIGTERM);
}
```

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!

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>

volatile int s=0;
void handler(int sig_nb){ s++; }

int main(){
  int n,k,w;
  pid_t parent;
  struct sigaction act;

  act.sa_handler = handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGUSR1, &act, NULL);

  if (!fork()){
    parent= getppid();
    for (n=0; n<10; n++) kill(parent,SIGUSR1);
    exit(0);
  }

  pause();
  while (s-- >0) {
    printf("received\n");
    sleep(3);
  }
}
```

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

#define CHILDREN 10
volatile int z=CHILDREN;
void handler(int sig_nb){
  pid_t child;
  do{
    child = waitpid(-1,NULL,WNOHANG);
    if (child>0) z--;
  } while (child>0);
  sleep(1);
}

int main(){
  int n;
  struct sigaction act;
  act.sa_handler = handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGCHLD,&act,NULL);

  for (n=0;n<CHILDREN;n++)
    if (!fork()) {
      sleep(n);
      return 0;
    }

  while (z>0) {
    printf("%d children/zombies left.\n",z);
    sleep(1);
  }
  printf("No more zombies.\n");
}
```

SIG_IGN dla SIGCHLD

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

#define CHILDREN 10

int main(){
  int n;
  struct sigaction act;
  act.sa_handler = SIG_IGN;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGCHLD,&act,NULL);

  for (n=0;n<CHILDREN;n++)
    if (!fork()) {
      return 0;
    }
  sleep(10);
}
```

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?

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

#define TIME 5
void handler(int sig_nb){ }

int main(){
  int n;
  struct sigaction act;

  act.sa_handler = handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGALRM, &act,NULL);

  alarm(TIME);
  pause();
  printf("No time ... no time to lose.\n");
}
```

sigprocmask

```
#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.

```c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

#define TIME 5
void handler(int sig_nb){ }

int main(){
  int n;
  struct sigaction act;
  sigset_t mask;


  act.sa_handler = handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGALRM, &act, NULL);
  sigaction(SIGINT, &act, NULL);

  sigfillset(&mask);
  sigdelset(&mask, SIGALRM);

  sigprocmask(SIG_SETMASK, &mask, NULL);
  alarm(TIME);
  pause();
  printf("No time ... no time to lose.\n");
}
```

sigsuspend

```
#include <signal.h>

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

```c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#define TIME 5
void handler(int sig_nb){ }

int main(){
  struct sigaction act;
  sigset_t mask;

  act.sa_handler = handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGALRM, &act, NULL);
  sigaction(SIGINT, &act, NULL);

  sigemptyset(&mask);
  sigaddset(&mask, SIGALRM);
  sigprocmask(SIG_BLOCK, &mask, NULL);

  sigfillset(&mask);
  sigdelset(&mask, SIGALRM);
  alarm(TIME);
  sigsuspend(&mask);
  printf("No time ... no time to lose.\n");
}
```

sigpending

```
#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.

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

volatile int time_is_up;
void handler(int sig_nb){ time_is_up = 1;}

int tread(char * buf, int n, int timeout){
  int r;
  struct sigaction act, oact;
  sigset_t mask, omask;

  act.sa_handler = handler;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGALRM,&act,&oact);

  time_is_up = 0;
  alarm(timeout);
  do r= read(0,buf,n);
  while ((!time_is_up) && (r<0));
  alarm(0);

  sigaction(SIGALRM,&oact,NULL);
  if (r<0) return 0;
  return r;
}
int main(){
  char buf[100];
  int n = tread(buf,100,5);
  write(1,buf,n);
}
```

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

```c
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/select.h>
#include <sys/time.h>

int tread(char * buf, int n, int timeout){
  int r;
  fd_set rfds;
  struct timeval timeout_s;

  FD_ZERO(&rfds);
  FD_SET(0,&rfds);

  timeout_s.tv_sec = timeout;
  timeout_s.tv_usec = 0;

  do r= select(1,&rfds,NULL,NULL,&timeout_s);
  while ((r<0) && (errno==EINTR));

  if (r<=0) return 0;

  return (read(0,buf,n));
}
int main(){
  char buf[100];
  int n = tread(buf,100,5);
  write(1,buf,n);
}
```

pselect

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

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

Outline

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
```

Remanent

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
```

Remanent

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
```

Remanent

Inne

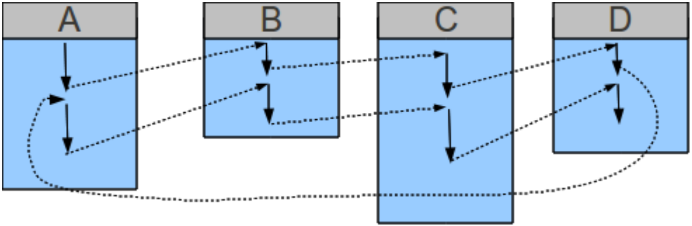
```
chroot  - change root directory
ptrace  - process trace
reboot  - close down the system or reboot
mmap    - request memory mapping
```

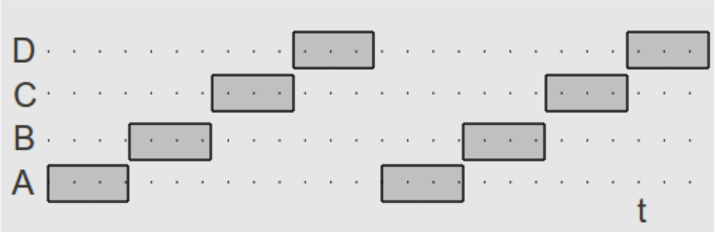
Outline

Multiprogramming (multitasking)

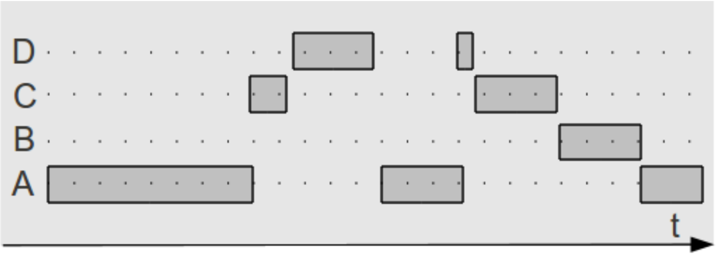
"Wirtualne procesory"

Każdy proces pracuje jak gdyby miał kopię procesora (z ograniczoną funkcjonalnością) dla siebie.



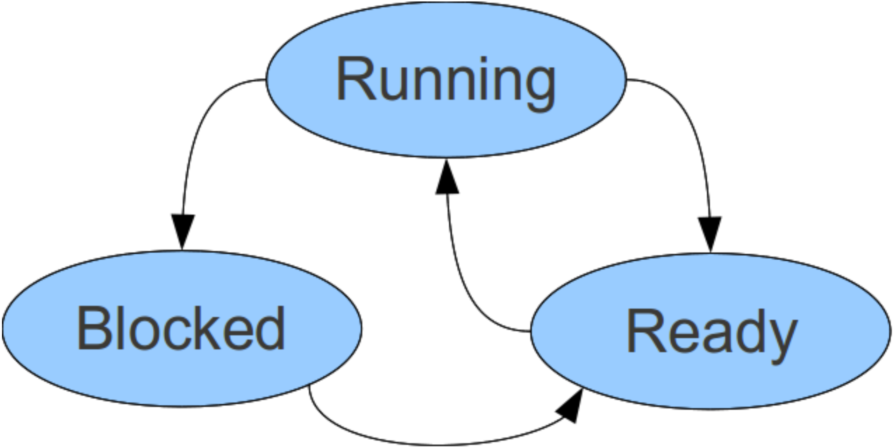


Multiprogramming (multitasking)



nierównomierny postęp w czasie $\longrightarrow$ nie wolno robić założeń dotyczących czasu rzeczywistego.

Stany procesu



Opis procesu

Kernel

- Registers
- Program counter
- Program status word
- Stack Pointer
- Process state
- Curent scheduling priority
- Maximum scheduling priority
- Scheduling ticks left
- Quantum size
- CPU time used
- Message queue pointers
- Pending signal bits
- Various flag bits
- Process name

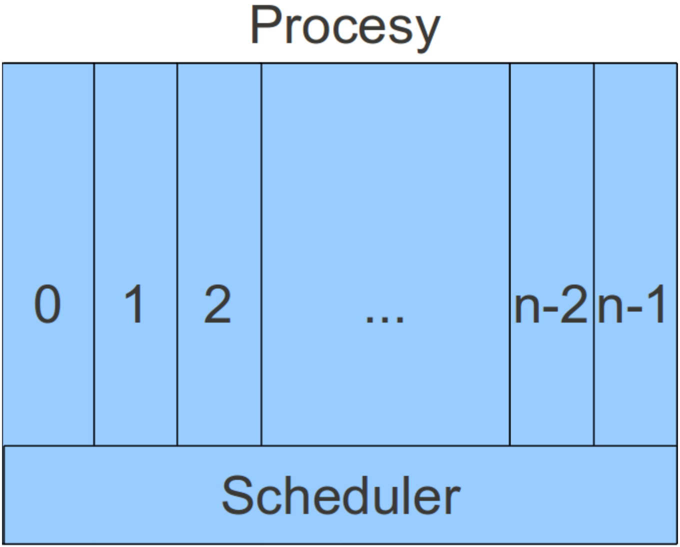
Process management

- Pointer to text segment
- Pointer to data segment
- Pointer to bss segment
- Exit status
- Signal status
- **Proces ID**
- Parent Process
- Process group
- Children's CPU time
- Real UID
- Effective UID
- Real GID
- Effective GID
- File info for sharing text
- Bitmaps for signals
- Various flag bits
- Process name

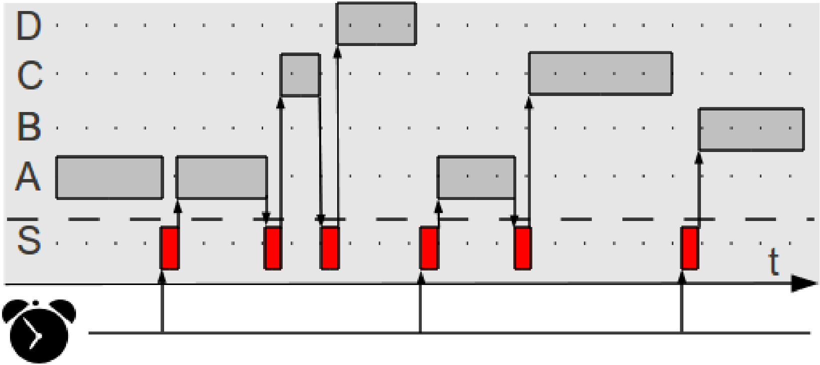
File management

- UMASK mask
- Root directory
- Working directory
- File descriptors
- Real UID
- Effective UID
- Rreal GID
- Effective GID
- Controlling tty
- Save area for read/write
- System call parameters
- Various flag bits

Scheduler



Przerwania zegarowe.



MINIX - 60 przerwań na sekundę.

Pamięć - zmiany kontekstu

Kernel

- **Registers**
- **Program counter**
- **Program status word**
- **Stack Pointer**
- **Process state**
- Curent scheduling priority
- Maximum scheduling priority
- Scheduling ticks left
- Quantum size
- CPU time used
- Message queue pointers
- Pending signal bits
- Various flag bits
- Process name

Process management

- **Pointer to text segment**
- **Pointer to data segment**
- **Pointer to bss segment**
- Exit status
- Signal status
- Proces ID
- Parent Process
- Process group
- Children's CPU time
- Real UID
- Effective UID
- Real GID
- Effective GID
- File info for sharing text
- Bitmaps for signals
- Various flag bits
- Process name

File management

- UMASK mask
- Root directory
- Working directory
- File descriptors
- Real UID
- Effective UID
- Rreal GID
- Effective GID
- Controlling tty
- Save area for read/write
- System call parameters
- Various flag bits

Threads - wątki

Wiele "lekkich procesów" w tej samej przestrzeni adresowej.

Zmiana na inny wątek w tym samym procesie

- Registers
- Program counter
- Stack pointer
- State

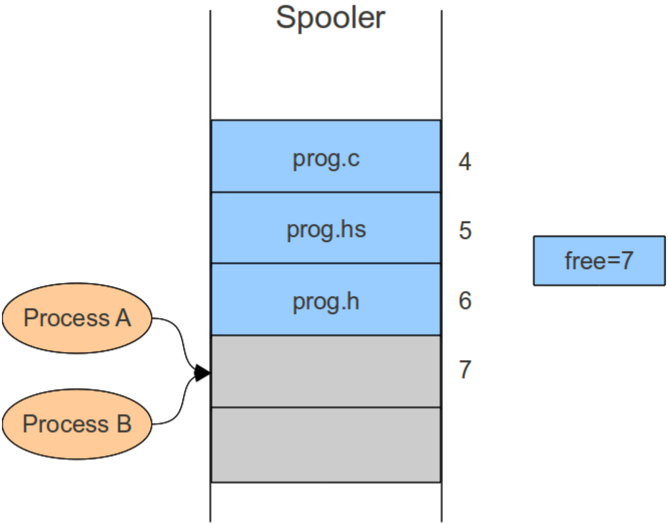
wspólna pamięć $\longrightarrow$ konieczna współpraca (synchronizacja).

systemowe / użytkownika

Outline

Współdzielona pamięć.

Race condition



Sekcje krytyczne

Sekcja krytyczna

Fragment programu z odwołaniami do współdzielonej pamięci.

Cel

Zawsze co najwyżej jeden proces/wątek w sekcji krytycznej.

Dodatkowo wymagamy:

- żaden proces działający poza sekcją krytyczną nie może blokować innego procesu,
- żeden proces nie powinien czekać w nieskończoność na wejście do sekcji krytycznej,
- mechanim powinien działać niezależnie od szybkości i liczby procesorów.

Wyłączanie przerwań.

CLI - processor instruction

`Clear Interrupt Flag - Clears the interrupt flag (IF) in the rFLAGS register to zero, thereby masking external interrupts received on the INTR input.`

Wady

- konieczne uprawnienia do blokowania przerwań
- nieskuteczne w systemach wieloprocesorowych

Busy waiting - spin lock

```
while (TRUE) {
    while (lock!=0);
    lock = 1;
    critical_region();
    lock = 0;
    noncritical_region();
}
```

```
while (TRUE) {
    while (lock!=0);
    lock = 1;
    critical_region();
    lock = 0;
    noncritical_region();
}
```

Busy waiting - spin lock + turns

```
while (TRUE) {
    while (turn!=0);
    critical_region();
    turn = 1;
    noncritical_region();
}
```

```
while (TRUE) {
    while (turn!=1);
    critical_region();
    turn = 0;
    noncritical_region();
}
```

Dude, [..] this determines who enters the next round robin. Am I wrong? Am I wrong?

Rozwiązanie Peterson'a

```
int turn;    //shared
int interested[2]; //shared

void enter_region(int process){
    int other;

    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while ((turn==process) && (interested[other]==TRUE));
}

void leave_region(int process){
    interested[process] = FALSE;
}
```

Test and Set Lock instruction

TSL

`TSL reg, lock` - wczytuje zawartość pamięci `lock` do rejestru reg oraz zapisuje niezerową wartość pod adresem `lock`.

```
enter_region:
   TSL eax, lock
   CMP eax, 0
   JNE enter_region
   RET
```

```
leave_region:
    MOV lock, 0
    RET
```

x64

`CMPXCHG reg/mem32, reg32`
Compare EAX register with a 32-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to EAX.

Kiedy używać spin-locków

Busy Waiting $\rightarrow$ Priority Inversion Problem

Sleep/Wakeup

```
#define N 100
int count=0;
void producer(void){
    int item;
    while (TRUE){
       item = produce_item();
       if (count==N) sleep();
       insert_item(item);
       count++;
       if (count==1) wakeup(consumer);
    }
}
void consumer(void){
    int item;
    while (TRUE){
       if (count==0) sleep();
       item = remove_item();
       count--;
       if (count==N-1) wakeup(producer);
       consume_item(item);
    }
```

Semafory

```
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

```
void producer(void){
    int item;
    while (TRUE){
       item = produce_item();
       down(&empty);
       down(&mutex);
       insert_item(item);
       up(&mutex);
       up(&full);
    }
}
```

```
void consumer(void){
    int item;
    while (TRUE){
       down(&full);
       down(&mutex);
       item = remove_item();
       up(&mutex);
       up(&empty);
       consume_item(item);
    }
}
```

Semafory - UWAGA!

```
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
```

```
void producer(void){
    int item;
    while (TRUE){
       item = produce_item();
       down(&mutex);  //zmiana
       down(&empty);  //kolejności
       insert_item(item);
       up(&mutex);
       up(&full);
    }
}
```

```
void consumer(void){
    int item;
    while (TRUE){
       down(&full);
       down(&mutex);
       item = remove_item();
       up(&mutex);
       up(&empty);
       consume_item(item);
    }
}
```

Monitory

```
monitor ProducerConsumer
   condition full, empty;
   int count;

  void insert(int item)
      if count==N then wait(full);
      insert_item(item);
      count++;
      if count==1 then signal(empty);

  int remove()
      if count==0 then wait(empty);
      remove = remove_item();
      count--;
      if (count= N-1) then signal(full);
```

Bez współdzielonej pamięci.

Message Passing (buffered)

```
#define N 100
```

```
void producer(void){
    int item;
    message m;

    while (TRUE){
       item = produce_item();
       receive(consumer, &m);
       build_message(&m, item);
       send(consumer, &m);
    }
}
```

```
void consumer(void){
    int item,i;
    message m;

    for (i=0; i<N; i++)
       send(producer,&m);
    while (TRUE){
       receive(producer,&m);
       item= extract_item(&m);
       send(producer,&m);
       consume_item(item);
    }
}
```

Message Passing - randezvous

```
#define N 100
```

```
void producer(void){
    int item;
    message m;

    while (TRUE){
       item = produce_item();
       receive(consumer, &m);
       build_message(&m, item);
       send(consumer, &m);
    }
}
```

```
void consumer(void){
    int item,i;
    message m;

    send(producer,&m);
    while (TRUE){
       receive(producer,&m);
       item= extract_item(&m);
       send(producer,&m);
       consume_item(item);
    }
}
```

Outline

Monolithic kernel - Linux

Procesy użytkownika pracują w dwóch trybach

- tryb użytkownika - user mode
- tryb jądra - kernel mode

Przejście do trybu jądra (system calls) poprzez specjalne bramki - w x86 przerwania programowe (`int 80h`) i specjalne instrukcje (`sysenter`, `syscall`).

Skutek: wiele procesów (użytkownika) może pracować równocześnie w trybie jądra.

Obsługa przerwań

- Scheduling.
- Drivery urządzeń w trybie jądra!

Do tego wątki systemu odpowiedzialne za takie rzeczy jak

- zapisywanie buforów (`pdflush`),
- kolejkę (drobnych) zadań jądra (`keventd`),
- zwalnianie/swapowanie stron pamięci (`kswapd`),

Outline

Micro kernel - Minix

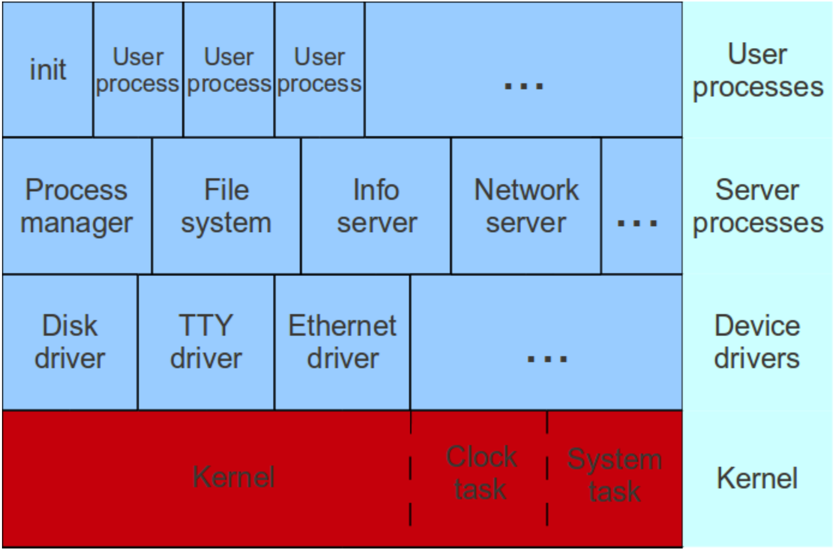
- Minimalizacja kodu pracującego w trybie uprzywilejowanym (kernel mode).
- Funkcjonalności systemu są zaimplementowane jako serwery (process manager, file system, network server).
- Komunikacja z serwerami - message passing.

Odpowiedzialność jądra

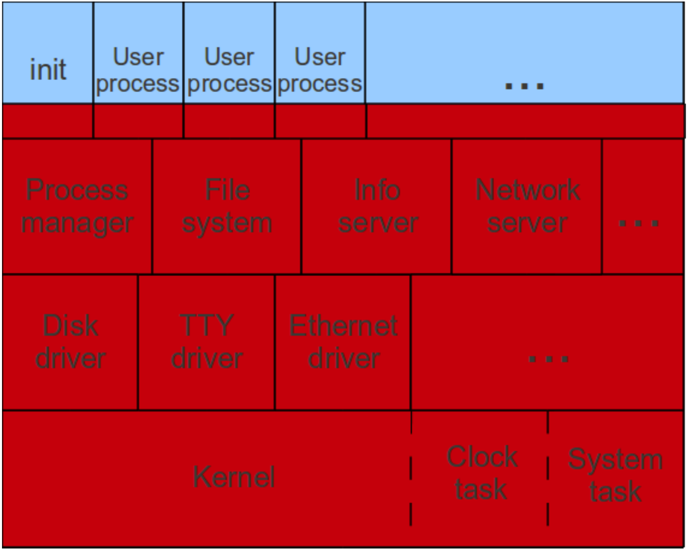
- multiprogramming,
- message - passing,
- mechanizmy dla obsługi urządzeń - drivery mogą pracować w trybie użytkownika.

Procesy użytkownika NIGDY nie pracują w trybie jądra.

Micro kernel - Minix



Monolithic kernel analogue



Outline

Tanenbaum - Torvalds debate

The Tanenbaum-Torvalds Debate, Open Sources: Voices from the Open Source Revolution, O'Reilly 1999

<u>ast:</u> `... LINUX is a monolithic style system. This is a giant step back into the 1970s. That is like taking an existing, working C program and rewriting it in BASIC. To me, writing a monolithic system in 1991 is a truly poor idea...`

<u>torvalds:</u> `... True, linux is monolithic, and I agree that microkernels are nicer. With a less argumentative subject, I'd probably have agreed with most of what you said. From a theoretical (and aesthetical) standpoint linux looses. If the GNU kernel had been ready last spring, I'd not have bothered to even start my project: the fact is that it wasn't and still isn't. Linux wins heavily on points of being available now...`

Tanenbaum - Torvalds debate

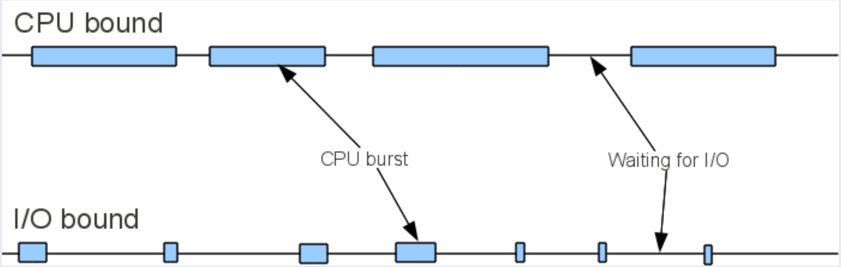
ast: ... Of course 5 years from now that will be different, but 5 years from now everyone will be running free GNU on their 200 MIPS, 64M SPARCstation-5...

PART II

Tanenbaum-Torvalds Debate: Part II, Tanenbaum's web page

Zachowanie procesów.

CPU-bound / I/O-bound



Kategorie procesów/schedulerów

- interaktywne
- wsadowe
- procesy czasu rzeczywistego

Kiedy scheduler wkracza do akcji?

Konieczna decyzja

- działający proces się blokuje
- działający proces się kończy

Możliwa zmiana procesu

- utworzenie nowego procesu
- przerwanie I/O
- przerwanie zegarowe (nonpreemptive / preemptive)

Wymagania

Wszystkie rodzaje

- Fairness - podobne procesy są podobnie traktowane
- Policy enforcement - uwzglednienie specyfiki procesów (priorytety itp.)
- Balance - wszystkie części systemu równo obciążone

Systemy wsadowe

- Throughput - maksymalizacja zadań na godzinę
- Turnaround time - minimalizacja czasu pomiędzy przyjściem a zakończeniem zadania (najlepiej znormalizowana wielkością zadania)
- CPU utilization - maksymalizacja obciążenia procesora

Wymagania - c.d.

Systemy interaktywne

- Response time - szybka reakcja na bodźce
- Proportionality - opóźnienie reakcji systmu proporcjonalne do czasu trwania zadania

Systemy czasu rzeczywistego

- Meeting deadlines
- Predictability

Outline

Batch systems: First-Come First-Served

- Procesy obsługiwane w kolejności przyjścia.
- Zablokowane procesy trafiają na koniec kolejki kiedy przechodzą w stan gotowości.

Wady

- Słabe wskaźniki: Turnaround time, CPU utilization, Throughput.
- Wiele procesów może pracować współbieżnie - duże obciążenie pamięci.
- Jeśli wszystkie procesy są zablokowane na I/O to dodawane są nowe - czas oczekiwania na I/O się wydłuża.

Batch systems: Shortest Job First

- Zakładamy, że potrafimy oszacować czas wykonania zadania (totalny czas, nie tylko CPU).
- Wybieramy proces o najkrótszym czasie.

Zalety

- Off line: Optymalny z punktu widzenia Turnaround time

Wady

- On line: nie optymalny. Zadania A,B,C,D,E o wagach 2,4,1,1,1 przychodzą w chwilach 0,0,3,3,3. Średnia 4.6. Jeśli wykonać B,C,D,E,A to średnia 4.4.
- Nie ma mechanizmów równoważenia procesów (CPU-bound/ I/O-bound)

Batch systems: Shortest Remaining Time Next

- Wybieramy proces o najkrótszym czasie pozostałym do zakończenia.
- Jeśli pojawia się nowy proces to dopuszczalne są wywłaszczenia.

Zalety

- Krótkie zadania mają krótki czas realizacji.
- Można uwzględnić blokowanie.
- Optymalny ze względu na Turnaround time w wersji on-line. Po zaniedbaniu kosztu zmiany kontekstu, schedulera, i ewentualnej korzyści z równoważenia.

Wady

- Nie ma mechanizmów równoważenia procesów (CPU-bound/ I/O-bound).
- Pesymistyczny Turnaround-time może być bardzo duży.

Batch systems: Three-Level Scheduling

Admission scheduler

- Wybiera zadania które zaczną być wykonywane współbieżnie.
- Równoważenie zadań CPU-bound / I/O-bound.
- Minimalizacja normalizowanego Turnaround time.

Memory scheduler

- Działa jeśli procesy nie mieszczą się w pamięci.
- Wybiera procesy, które zostaną swapowane na dysk.
- Równoważenie zadań CPU-bound / I/O-bound.

CPU scheduler

- Wybiera następne zadanie do wykonania.
- Równoważenie zadań CPU-bound / I/O-bound.

Outline

Interactive systems: Round robin

- Każdy proces dostaje kwant czasu (quantum).
- Procesy ustawione są w kolejce.
- Proces może obliczać tak długo jak trwa jego kwant.
- Proces który zużyje kwant, jest przerywany, dostaje nowe kwant i trafia na koniec kolejki.
- Proces, który się blokuje przed wykorzystaniem quantu, po przejściu w stan Ready, trafia na początek kolejki z pozostałym czasem.

Ważny parametr: długość kwantu

- kwant za duży - system może mieć długi czas reakcji.
- kwant za mały - duży odsetek pracy procesora tracony na zmiany kontekstu.
- w praktyce 20-50 msec.

Wady

- Wszystkie procesy są równo traktowane.

Interactive systems: Priority scheduling

Uogólnienie Round-robin. Wiele możliwości:

- Kwanty mogą zależeć od priorytetu.
- Procesy o wyższym priorytecie mają pierwszeństwo.
- Dla każdego priorytetu lista procesów - Round - robin.
- Dynamiczne/statyczne przydzialanie priorytetu (np. jeśli proces zużył $1/f$ swojego kwantu to dostaje priorytet f).
- Narastające kwanty w niższych priorytetach.

Ważny parametr: długość kwantu

- kwant za duży - system może mieć długi czas reakcji.
- kwant za mały - duży odsetek pracy procesora tracony na zmiany kontekstu.
- w praktyce 20-50 msec.

Interactive systems

Guaranteed scheduling / Fair Share scheduling

- Każdy proces/użytkownik ma zagwarantowane $x\%$ procesora.
- Scheduler wybiera procesy tak aby wypełnić zobowiązanie.

Lottery scheduling

- Każdy proces dostaje kilka żetonów z puli.
- Scheduler wybiera losowo żeton i przydziela procesor właścicielowi.
- Współpracujące procesy mogą sobie przekazywać żetony.

Outline

- hard real time / soft real time.
- periodyczne, aperiodyczne zdarzenia.
- szeregowanie statyczne/dynamiczne.

Schedulable system

C_i - czas obsługi zdarzenia i
P_i - częstotliwość zdarzenia i

$$\sum_i \frac{C_i}{P_i} \leqslant 1$$

Outline

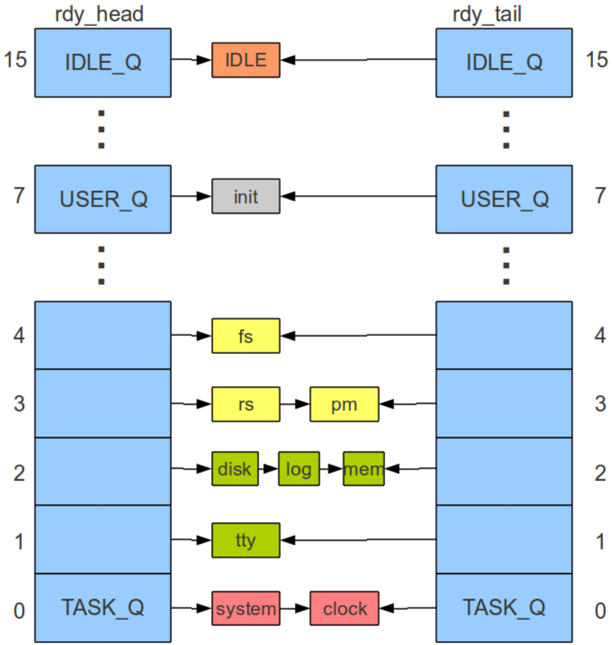
Scheduling in MINIX

Multilevel/Priority scheduling

- Dla każdego priorytetu kolejka.
- W obrębie kolejki round-robin.
- W kolejkach tylko procesy w stanie `ready`.
- Jeśli proces przechodzi w stan `ready` to:
  - Trafia na początek kolejki, jeśli nie wykorzystał całego czasu (kwantu) przed zablokowaniem,
  - wpp - trafia na koniec kolejki z nowym kwantem.

Wybór kolejnego procesu:

Weź pierwszy proces z niepustej kolejki o największym priorytecie.


rdy_head
rdy_tail
15
IDLE_Q
IDLE
IDLE_Q
15
7
USER_Q
init
USER_Q
7
4
fs
4
3
rs
pm
3
2
disk
log
mem
2
1
tty
1
0
TASK_Q
system
clock
TASK_Q
0

Degradacja priorytetu

+1 Proces zużył cały kwant i dwa razy z rzędu został mu przydzielnony procesor.

-1 wpp.

- Nie dotyczy zadań jądra (kernel tasks) oraz procesu `IDLE`.
- Priorytet nie może być większy niż `IDLE_Q-1`.
- Priorytet nie może być mniejszy niż maksymalny priorytet zależny od rodzaju procesu.

```
PUBLIC int _syscall(int who, int syscallnr, message * msgptr){
  int status;              /*lib/other/syscall.c*/

  msgptr->m_type = syscallnr;
  status = _sendrec(who, msgptr);
  if (status != 0) {
      /* 'sendrec' itself failed. */
      msgptr->m_type = status;
  }
  if (msgptr->m_type < 0) {
      errno = -msgptr->m_type;
      return(-1);
  }
  return(msgptr->m_type);
}

__sendrec:                    ! lib/i386/rts/_ipc.s
     push ebp
     mov ebp, esp
     push ebx
     mov eax, SRC_DST(ebp)   ! eax = dest-src
     mov ebx, MESSAGE(ebp)   ! ebx = message pointer
     mov ecx, SENDREC        ! _sendrec(srcdest, ptr)
     int SYSVEC              ! trap to the kernel
     pop ebx
     pop ebp
     ret
```

```
_s_call:                ! kernel/mpx386.s
_p_s_call:
    cld                 ! set direction flag to a known value
    sub esp, 6*4    ! skip RETADR, eax, ecx, edx, ebx, est
    push ebp         ! stack already points into proc table
    push esi
    push edi
    o16 push ds
    o16 push es
    o16 push fs
    o16 push gs
    mov dx, ss
    mov ds, dx
    mov es, dx
    incb (_k_reenter)
    mov esi, esp    ! assumes P_STACKBASE == 0
    mov esp, k_stktop
    xor ebp, ebp    ! for stacktrace
                        ! end of inline save
                        ! now set up parameters for sys_call()
    push ebx     ! pointer to user message
    push eax     ! src/dest
    push ecx     ! SEND/RECEIVE/BOTH
    call _sys_call ! sys_call(function, src_dest, m_ptr)
        ! caller is now explicitly in proc_ptr
    mov AXREG(esi), eax ! sys_call MUST PRESERVE si
```

```
! Fall into code to restart proc/task running.
_restart:                    ! kernel/mpx386.s

! Restart the current process or the next process if it is set.
    cmp (_next_ptr), 0      ! see if another process is scheduled
    jz 0f
    mov  eax, (_next_ptr)
    mov (_proc_ptr), eax    ! schedule new process
    mov (_next_ptr), 0
0:  mov esp, (_proc_ptr)    ! will assume P_STACKBASE == 0
    lldt P_LDT_SEL(esp)  ! enable process' segment descriptors
    lea eax, P_STACKTOP(esp) ! arrange for next interrupt
    mov (_tss+TSS3_S_SP0), eax ! to save state in process table
restart1:
    decb (_k_reenter)
    o16 pop gs
    o16 pop fs
    o16 pop es
    o16 pop ds
    popad
    add esp, 4      ! skip return adr
    iretd           ! continue process
```

```c
PUBLIC int sys_call(call_nr, src_dst, m_ptr)
int call_nr;        /* system call number and flags */
int src_dst;        /* src to receive from or dst to send to */
message *m_ptr;       /* pointer to message in the caller's space */
{
  /.../
  int function = call_nr & SYSCALL_FUNC;  /* get system call function
  /...// check several conditions
  switch(function) {
  case SENDREC:
      /* A flag is set so that notifications cannot interrupt SENDREC
      priv(caller_ptr)->s_flags |= SENDREC_BUSY;
      /* fall through */
  case SEND:
      result = mini_send(caller_ptr, src_dst, m_ptr, flags);
      if (function == SEND || result != OK) {
          break;          /* done, or SEND failed */
      }               /* fall through for SENDREC */
  case RECEIVE:
      if (function == RECEIVE)
          priv(caller_ptr)->s_flags &= ~SENDREC_BUSY;
      result = mini_receive(caller_ptr, src_dst, m_ptr, flags);
      break;
  /...// NOTIFY, ECHO, default
  }
  return(result);
}
```

```c
PRIVATE int mini_send(caller_ptr, dst, m_ptr, flags)
register struct proc *caller_ptr; /* who is trying to send a message? */
int dst;        /* to whom is message being sent? */
message *m_ptr;       /* pointer to message buffer */
unsigned flags;       /* system call flags */
{
/* Send a message from 'caller_ptr' to 'dst'. */
  register struct proc *dst_ptr = proc_addr(dst);
  register struct proc **xpp;
  register struct proc *xp;
  /.../* Check for deadlock by 'caller_ptr' and 'dst' sending to each other. */

  /* Check if 'dst' is blocked waiting for this message. The destination's
   * SENDING flag may be set when its SENDREC call blocked while sending.
   */
  if ( (dst_ptr->p_rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
       (dst_ptr->p_getfrom == ANY || dst_ptr->p_getfrom == caller_ptr->p_nr)) {
       /* Destination is indeed waiting for this message. */
       CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr, dst_ptr->p_messbuf);
       if ((dst_ptr->p_rts_flags &= ~RECEIVING) == 0) enqueue(dst_ptr);
  } else if ( ! (flags & NON_BLOCKING)) {
       /* Destination is not waiting.  Block and dequeue caller. */
       caller_ptr->p_messbuf = m_ptr;
       if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
       caller_ptr->p_rts_flags |= SENDING;
       caller_ptr->p_sendto = dst;
       /* Process is now blocked.  Put in on the destination's queue. */
       xpp = &dst_ptr->p_caller_q;    /* find end of list */
       while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
       *xpp = caller_ptr;     /* add caller to end */
       caller_ptr->p_q_link = NIL_PROC; /* mark new end of list */
  } else {
       return(ENOTREADY);
  }
  return(OK);
}
```

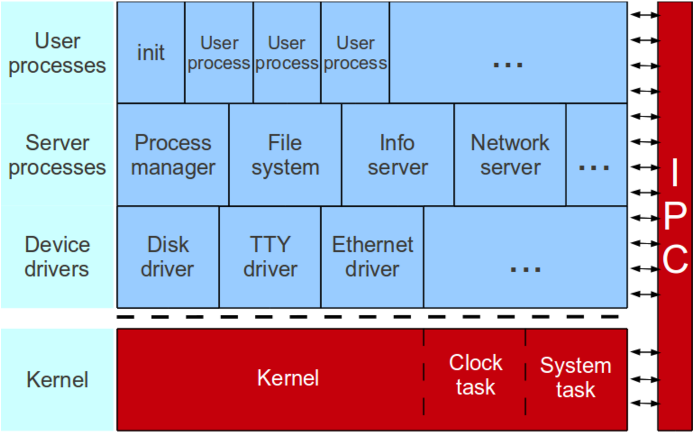
```c
PRIVATE void enqueue(rp)
register struct proc *rp; /* this process is now runnable */
{
  int q;           /* scheduling queue to use */
  int front;            /* add to front or back */

  /* Determine where to insert to process. */
  sched(rp, &q, &front);

  /* Now add the process to the queue. */
  if (rdy_head[q] == NIL_PROC) {     /* add to empty queue */
      rdy_head[q] = rdy_tail[q] = rp;      /* create a new queue */
      rp->p_nextready = NIL_PROC;    /* mark new end */
  }
  else if (front) {        /* add to head of queue */
      rp->p_nextready = rdy_head[q];     /* chain head of queue */
      rdy_head[q] = rp;         /* set new queue head */
  }
  else {           /* add to tail of queue */
      rdy_tail[q]->p_nextready = rp;     /* chain tail of queue */
      rdy_tail[q] = rp;         /* set new queue tail */
      rp->p_nextready = NIL_PROC;    /* mark new end */
  }

  /* Now select the next process to run. */
  pick_proc();
}
```

```c
/*===========================================================================*
 *          pick_proc              *
 *===========================================================================*/
PRIVATE void pick_proc()
{
/* Decide who to run now.  A new process is selected by setting 'next_ptr'.
 * When a billable process is selected, record it in 'bill_ptr', so that the
 * clock task can tell who to bill for system time.
 */
  register struct proc *rp;     /* process to run */
  int q;           /* iterate over queues */

  /* Check each of the scheduling queues for ready processes. The number of
   * queues is defined in proc.h, and priorities are set in the image table.
   * The lowest queue contains IDLE, which is always ready.
   */
  for (q=0; q < NR_SCHED_QUEUES; q++) {
      if ( (rp = rdy_head[q]) != NIL_PROC) {
          next_ptr = rp;       /* run process 'rp' next */
          if (priv(rp)->s_flags & BILLABLE)
              bill_ptr = rp;      /* bill for system time */
          return;
      }
  }
}
```

```c
PRIVATE void sched(rp, queue, front)
register struct proc *rp;      /* process to be scheduled */
int *queue;          /* return: queue to use */
int *front;          /* return: front or back */
{
  static struct proc *prev_ptr = NIL_PROC;  /* previous without time */
  int time_left = (rp->p_ticks_left > 0); /* quantum fully consumed */
  int penalty = 0;        /* change in priority */

  if ( ! time_left) {        /* quantum consumed ? */
      rp->p_ticks_left = rp->p_quantum_size;  /* give new quantum */
      if (prev_ptr == rp) penalty ++;   /* catch infinite loops */
      else penalty --;        /* give slow way back */
      prev_ptr = rp;        /* store ptr for next */
  }

  /* Determine the new priority of this process. The bounds are determined
   * by IDLE's queue and the maximum priority of this process. Kernel task
   * and the idle process are never changed in priority.
   */
  if (penalty != 0 && ! iskernelp(rp)) {
      rp->p_priority += penalty;    /* update with penalty */
      if (rp->p_priority < rp->p_max_priority)  /* check upper bound */
          rp->p_priority=rp->p_max_priority;
      else if (rp->p_priority > IDLE_Q-1)     /* check lower bound */
          rp->p_priority = IDLE_Q-1;
  }

  /* If there is time left, the process is added to the front of its queue,
   * so that it can immediately run. The queue to use simply is always the
   * process' current priority.
   */
  *queue = rp->p_priority;
  *front = time_left;
}
```

Outline

IPC - message passing



Message Passing Primitives

- SEND=1
- RECEIVE=2
- SENDREC=3
- NOTIFY=4

Kernel trap

```
push  ebp
mov ebp, esp
push  ebx
mov eax, SRC_DST(ebp)  ! eax = dest-src
mov ebx, MESSAGE(ebp)  ! ebx = message pointer
mov ecx, {SEND|RECEIVE|SENDREC|NOTIFY}
int SYSVEC         ! 33
pop ebx
pop ebp
ret
```

```c
PUBLIC int sys_call(call_nr, src_dst, m_ptr)
int call_nr;        /* system call number and flags */
int src_dst;        /* src to receive from or dst to send to */
message *m_ptr;     /* pointer to message in the caller's space */
{
  /.../
  int function = call_nr & SYSCALL_FUNC;  /* get system call function
  /...// check several conditions
  switch(function) {
  case SENDREC:
      /* A flag is set so that notifications cannot interrupt SENDREC
      priv(caller_ptr)->s_flags |= SENDREC_BUSY;
      /* fall through */
  case SEND:
      result = mini_send(caller_ptr, src_dst, m_ptr, flags);
      if (function == SEND || result != OK) {
          break;          /* done, or SEND failed */
      }                   /* fall through for SENDREC */
  case RECEIVE:
      if (function == RECEIVE)
          priv(caller_ptr)->s_flags &= ~SENDREC_BUSY;
      result = mini_receive(caller_ptr, src_dst, m_ptr, flags);
      break;
 case NOTIFY:
      result = mini_notify(caller_ptr, src_dst);
      break;
  /...// ECHO, default
```

Message Passing Primitives

- `SEND`,`RECEIVE`,`SENDREC` - są blokujące (randez-vous). Uwaga na deadlock!
- `NOTIFY` - nie blokujące, nie buforowane
- Wiadomości wychodzące z jądra (kernel tasks) nie przechodzą przez `sys_call`.

Testy w `sys_call`

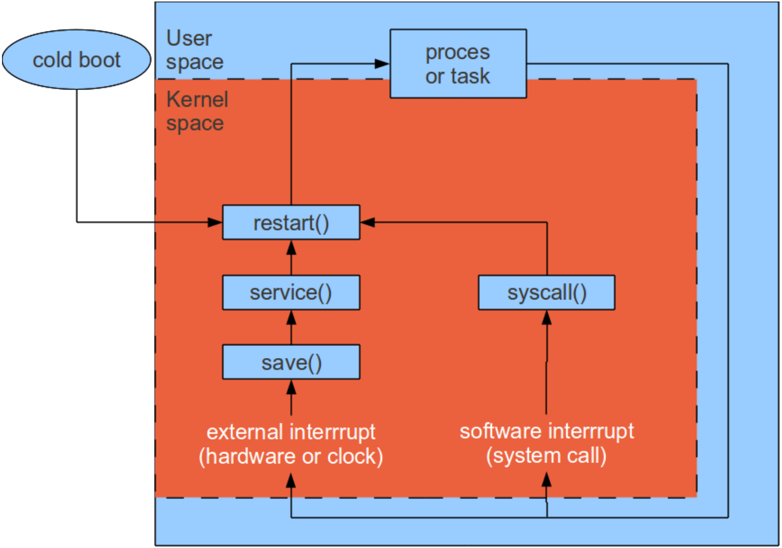
- Wiadomości mogą być wysyłane do zadań jądra wyłącznie przez `SENDREC`.
- Każdy proces ma maskę operacji IPC które może wykonywać (w strukturze `priv`) - procesy użytkownika mogą używać jedynie `SENDREC`.
- Argument `src_dst` musi być prawidłowym identyfikatorem procesu/zadania lub `ANY` dla `RECEIVE`.
- Wskaźnik do wiadomości musi się znajdować w pamięci procesu wołającego.
- Każdy proces ma maskę procesów do których może wysyłać (w strukturze `priv`) - procesy użytkownika mogą wysyłać do `PM`, `FS`, `RS`.
- Proces wskazywany przez `src_dst` musi być żywy lub `ANY`.

```
PRIVATE int mini_notify(caller_ptr, dst)
register struct proc *caller_ptr; /* sender of the notification */
int dst;          /* which process to notify */
{
  register struct proc *dst_ptr = proc_addr(dst);
  int src_id;         /* source id for late delivery */
  message m;          /* the notification message */

  /* Check to see if target is blocked waiting for this message. A process
   * can be both sending and receiving during a SENDREC system call.
   */
  if ((dst_ptr->p_rts_flags & (RECEIVING|SENDING)) == RECEIVING &&
      ! (priv(dst_ptr)->s_flags & SENDREC_BUSY) &&
      (dst_ptr->p_getfrom == ANY || dst_ptr->p_getfrom == caller_ptr->p_nr)) {

      /* Destination is indeed waiting for a message. Assemble a notification
       * message and deliver it. Copy from pseudo-source HARDWARE, since the
       * message is in the kernel's address space.
       */
      BuildMess(&m, proc_nr(caller_ptr), dst_ptr);
      CopyMess(proc_nr(caller_ptr), proc_addr(HARDWARE), &m,
          dst_ptr, dst_ptr->p_messbuf);
      dst_ptr->p_rts_flags &= ~RECEIVING; /* deblock destination */
      if (dst_ptr->p_rts_flags == 0) enqueue(dst_ptr);
      return(OK);
  }

  /* Destination is not ready to receive the notification. Add it to the
   * bit map with pending notifications. Note the indirectness: the system id
   * instead of the process number is used in the pending bit map.
   */
  src_id = priv(caller_ptr)->s_id;
  set_sys_bit(priv(dst_ptr)->s_notify_pending, src_id);
  return(OK);
}
```

```
PRIVATE int mini_receive(caller_ptr, src, m_ptr, flags)
register struct proc *caller_ptr; /* process trying to get message */
int src;            /* which message source is wanted */
message *m_ptr;          /* pointer to message buffer */
unsigned flags;          /* system call flags */
{
    /.../
    if (!(caller_ptr->p_rts_flags & SENDING)) {

    /* Check if there are pending notifications, except for SENDREC.
    if (! (priv(caller_ptr)->s_flags & SENDREC_BUSY)) {
        map = &priv(caller_ptr)->s_notify_pending;
        for (chunk=&map->chunk[0]; \
            chunk<&map->chunk[NR_SYS_CHUNKS]; \
            chunk++) {
          /* Find a pending notification from the requested source.
          /.../
          /* Found a suitable source,
          *  deliver the notification message. */
          BuildMess(&m, src_proc_nr, caller_ptr); /* assemble messa
          CopyMess(src_proc_nr, proc_addr(HARDWARE), \
            &m, caller_ptr, m_ptr);
          return(OK);         /* report success */
        }
    }
...
```

```c
...
    /* Check caller queue. Use pointer pointers to keep code simple. *
    xpp = &caller_ptr->p_caller_q;
    while (*xpp != NIL_PROC) {
        if (src == ANY || src == proc_nr(*xpp)) {
      /* Found acceptable message. Copy it and update status. */
            CopyMess((*xpp)->p_nr,*xpp,(*xpp)->p_messbuf,caller_ptr,m
            if (((*xpp)->p_rts_flags &= ~SENDING) == 0) enqueue(*xpp);
            *xpp = (*xpp)->p_q_link;     /* remove from queue */
            return(OK);          /* report success */
  }
  xpp = &(*xpp)->p_q_link;     /* proceed to next */
    }
  }
...
```

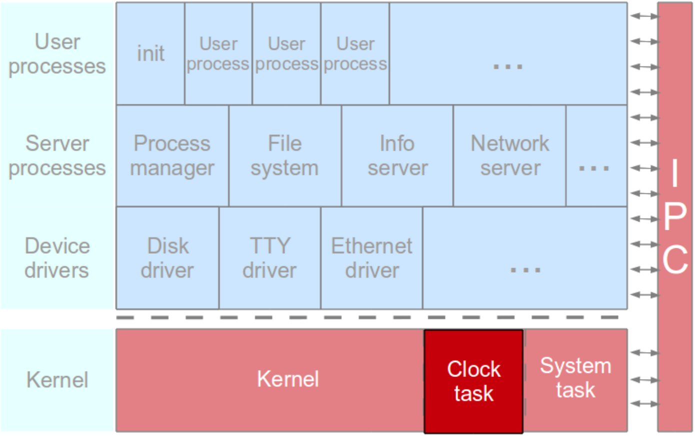
```
...
  /* No suitable message is available or the caller couldn't
   * send in SENDREC.
   * Block the process trying to receive,
   * unless the flags tell otherwise.
   */
  if ( ! (flags & NON_BLOCKING)) {
        caller_ptr->p_getfrom = src;
        caller_ptr->p_messbuf = m_ptr;
        if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
        caller_ptr->p_rts_flags |= RECEIVING;
        return(OK);
  } else {
        return(ENOTREADY);
  }
}
```


cold boot
User space
proces or task
Kernel space
restart()
service()
syscall()
save()
external interrupt (hardware or clock)
software interrupt (system call)

```c
PRIVATE int generic_handler(hook)
irq_hook_t *hook;
{
/* This function handles hardware interrupt in a simple and generic
 * way. All interrupts are transformed into messages to a driver.
 * The IRQ line will be reenabled if the policy says so.
 */

  /* As a side-effect, the interrupt handler gathers random
   * information by timestamping the interrupt events.
   * This is used for /dev/random.
   */
  get_randomness(hook->irq);

  /* Add a bit for this interrupt to the process' pending interrupts.
   * When sending the notification message, this bit map will be
   *magically set as an argument.
   */
 priv(proc_addr(hook->proc_nr))->s_int_pending |=(1<<hook->notify_id);

  /* Build notification message and return. */
  lock_notify(HARDWARE, hook->proc_nr);
  return(hook->policy & IRQ_REENABLE);
}
```

Outline

Clock task


User processes
init
User process
User process
User process
...
Server processes
Process manager
File system
Info server
Network server
...
Device drivers
Disk driver
TTY driver
Ethernet driver
...
Kernel
Kernel
Clock task
System task
IPC

Funkcje zegara

clock_handler() Obsługa przerwań zegara - aktualizacja czasu, “księgowość”, sprawdzanie alarmów i potrzeby wywłaszenia.

clock_task() Wywłaszczanie procesów których czas się wyczerpał, uruchamianie alarmów.

inne Funkcje dotyczące zegara:

- `get_uptime()`
- `set_timer()`
- `reset_timer()`
- `read_clock()`
- `clock_stop()`

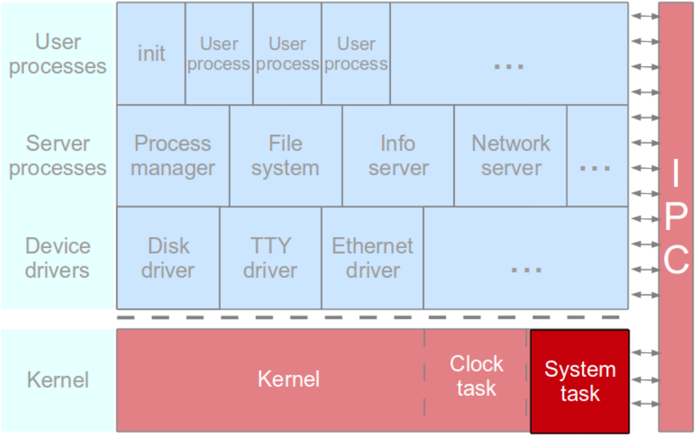
```c
PRIVATE int clock_handler(hook)
irq_hook_t *hook;
{
  register unsigned ticks;

  /* Acknowledge the PS/2 clock interrupt. */
  if (machine.ps_mca) outb(PORT_B, inb(PORT_B) | CLOCK_ACK_BIT);

  /* Get number of ticks and update realtime. */
  ticks = lost_ticks + 1;
  lost_ticks = 0;
  realtime += ticks;

  proc_ptr->p_user_time += ticks;
  if (priv(proc_ptr)->s_flags & PREEMPTIBLE) {
      proc_ptr->p_ticks_left -= ticks;
  }
  if (! (priv(proc_ptr)->s_flags & BILLABLE)) {
      bill_ptr->p_sys_time += ticks;
      bill_ptr->p_ticks_left -= ticks;
  }

  /* Check if do_clocktick() must be called. Done for alarms and scheduling.
   * Some processes, such as the kernel tasks, cannot be preempted.
   */
  if ((next_timeout <= realtime) || (proc_ptr->p_ticks_left <= 0)) {
      prev_ptr = proc_ptr;           /* store running process */
      lock_notify(HARDWARE, CLOCK);  /* send notification */
  }
  return(1);               /* reenable interrupts */
}
```

```
PUBLIC void clock_task()
{
/* Main program of clock task. If the call is not HARD_INT it is an e
 */
  message m;        /* message buffer for both input and output */
  int result;       /* result returned by the handler */

  init_clock();        /* initialize clock task */

  /* Main loop of the clock task.  Get work, process it. Never reply.
  while (TRUE) {

       /* Go get a message. */
       receive(ANY, &m);

       /* Handle the request. Only clock ticks are expected. */
       switch (m.m_type) {
       case HARD_INT:
            result = do_clocktick(&m);  /* handle clock tick */
            break;
       default:            /* illegal request type */
            kprintf("CLOCK: illegal request %d from %d.\n", m.m_type,m.
       }
  }
}
```

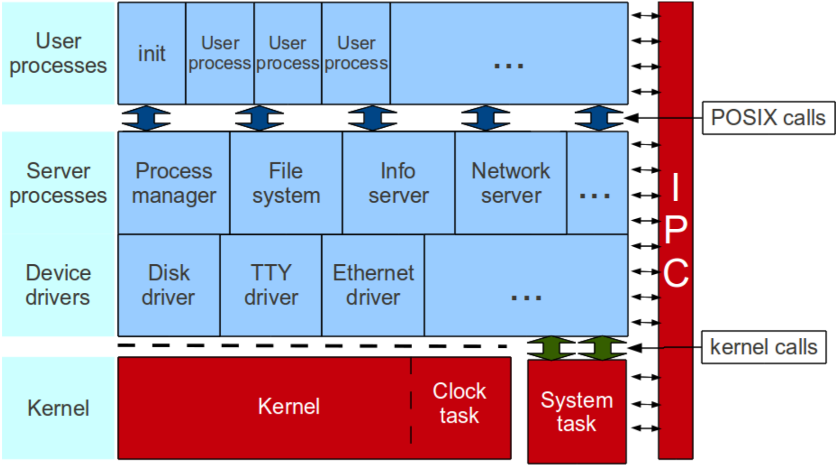
```c
PRIVATE int do_clocktick(m_ptr)
message *m_ptr;          /* pointer to request message */
{
  if (prev_ptr->p_ticks_left <= 0 &&
   priv(prev_ptr)->s_flags & PREEMPTIBLE) {
      lock_dequeue(prev_ptr);    /* take it off the queues */
      lock_enqueue(prev_ptr);    /* and reinsert it again */
  }

  /* Check if a clock timer expired and run its watchdog function. */
  if (next_timeout <= realtime) {
    tmrs_exptimers(&clock_timers, realtime, NULL);
    next_timeout = clock_timers == NULL ?
          TMR_NEVER : clock_timers->tmr_exp_time;
  }

  /* Inhibit sending a reply. */
  return(EDONTREPLY);
}
```

Outline

System task


User processes
init
User process
User process
User process
...
Server processes
Process manager
File system
Info server
Network server
...
Device drivers
Disk driver
TTY driver
Ethernet driver
...
Kernel
Kernel
Clock task
System task
IPC

System task


User processes
init
User process
User process
User process
...
Server processes
Process manager
File system
Info server
Network server
...
Device drivers
Disk driver
TTY driver
Ethernet driver
...
Kernel
Kernel
Clock task
System task
IPC
POSIX calls
kernel calls

Kernel calls

Message type	From	Meaning
sys_fork	PM	A process has forked.
sys_exec	PM	Set stack pointer after EXEC call
sys_exit	PM	A process has exited.
sys_nice	PM	Set scheduling priority.
sys_trace	PM	Carry out on operation of PTRACE call.
sys_kill	PM, FS, TTY	Send signal to a process after KILL call. (notify)
sys_getksig	PM	PM is checking for pending signals.
sys_endksig	PM	PM has finished processing signals.
sys_sigsend	PM	Send a signal to a process.
sys_sigreturn	PM	Cleanup after completion of a signal.
sys_newmap	PM	Set up a process memory map.
sys_memset	PM	Write char to memory area.
sys_times	PM	Get uptime and process times.
sys_setalarm	PM, FS, Drivers	Schedule synchronous alarm.
sys_abort	PM, TTY	Panic: MINIX is unable to continue

Kernel calls

Message type	From	Meaning
sys_privctl	RS	Set or change priviliges.
sys_irqctl	Drivers	Enable, disable or configure interrupt.
sys_devio	Drivers	Read from or write to I/O port.
sys_sdevio	Drivers	Read or write string from/to I/O port.
sys_vdevio	Drivers	Carry out vector of I/O requests.
sys_int86	Drivers	Do a real-mode BIOS call
sys_segctl	Drivers	Add segment and get selector (far data access).
sys_umap	Drivers	Convert virtual address to physical addr.
sys_vircopy	FS, Drivers	Copy using pure virt. addressing.
sys_physcopy	Drivers	Copy using physical addressing.
sys_virvcopy	Any	Vector of VCOPY requests.
sys_physvcopy	Any	Vector of PHYSCOPY requests.
sys_getinfo	Any	Requests system information.

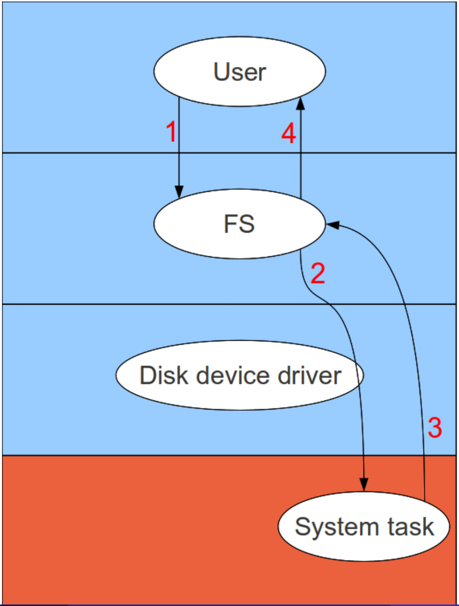
```
PUBLIC void sys_task()
{
/* Main entry point of sys_task.  Get the message and dispatch on type. */
  /.../

  initialize();

  while (TRUE) {
      /* Get work. Block and wait until a request message arrives. */
      receive(ANY, &m);
      call_nr = (unsigned) m.m_type - KERNEL_CALL;
      caller_ptr = proc_addr(m.m_source);

      /* See if the caller made a valid request and try to handle it. */
      if (! (priv(caller_ptr)->s_call_mask & (1<<call_nr))) {
    kprintf("SYSTEM: request %d from %d denied.\n", call_nr,m.m_source);
    result = ECALLDENIED;      /* illegal message type */
      } else if (call_nr >= NR_SYS_CALLS) {    /* check call number */
    kprintf("SYSTEM: illegal request %d from %d.\n", call_nr,m.m_source);
    result = EBADREQUEST;      /* illegal message type */
      }
      else {
          result = (*call_vec[call_nr])(&m);  /* handle the kernel call */
      }

      /* Send a reply, unless inhibited by a handler function. Use the kernel
       * function lock_send() to prevent a system call trap. The destination
       * is known to be blocked waiting for a message.
       */
      if (result != EDONTREPLY) {
     m.m_type = result;        /* report status of call */
          if (OK != (s=lock_send(m.m_source, &m))) {
              kprintf("SYSTEM, reply to %d failed: %d\n", m.m_source, s);
          }
      }
  }
}
```

```c
PRIVATE void initialize(void)
{
  register struct priv *sp;
  int i;

  /* Initialize IRQ handler hooks. Mark all hooks available. */
  for (i=0; i<NR_IRQ_HOOKS; i++) {
      irq_hooks[i].proc_nr = NONE;
  }

  /* Initialize all alarm timers for all processes. */
  for (sp=BEG_PRIV_ADDR; sp < END_PRIV_ADDR; sp++) {
    tmr_inittimer(&(sp->s_alarm_timer));
  }

  /* Initialize the call vector to a safe default handler. */
  for (i=0; i<NR_SYS_CALLS; i++) {
      call_vec[i] = do_unused;
  }

  /* Process management. */
  map(SYS_FORK, do_fork);     /* a process forked a new process */
  map(SYS_EXEC, do_exec);   /* update process after execute */
  /.../
}
```

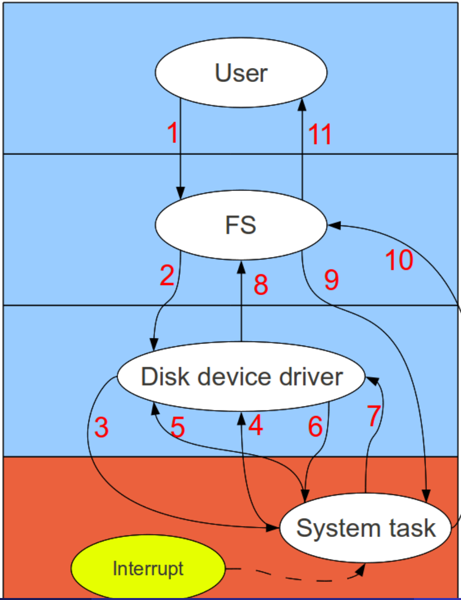
```c
PUBLIC int do_setalarm(m_ptr)
message *m_ptr;      /* pointer to request message */
{
/* A process requests a synchronous alarm, or wants to cancel its alarm. */
  /.../
  /* Extract shared parameters from the request message. */
  exp_time = m_ptr->ALRM_EXP_TIME;   /* alarm's expiration time */
  use_abs_time = m_ptr->ALRM_ABS_TIME;  /* flag for absolute time */
  proc_nr = m_ptr->m_source;     /* process to interrupt later */
  rp = proc_addr(proc_nr);
  if (! (priv(rp)->s_flags & SYS_PROC)) return(EPERM);

  /* Get the timer structure and set the parameters for this alarm. */
  tp = &(priv(rp)->s_alarm_timer);
  tmr_arg(tp)->ta_int = proc_nr;
  tp->tmr_func = cause_alarm;

  /* Return the ticks left on the previous alarm. */
  uptime = get_uptime();
  if ((tp->tmr_exp_time != TMR_NEVER) && (uptime < tp->tmr_exp_time) ) {
      m_ptr->ALRM_TIME_LEFT = (tp->tmr_exp_time - uptime);
  } else {
      m_ptr->ALRM_TIME_LEFT = 0;
  }

  /* Finally, (re)set the timer depending on the expiration time. */
  if (exp_time == 0) {
      reset_timer(tp);
  } else {
      tp->tmr_exp_time = (use_abs_time) ? exp_time : exp_time + get_uptime();
      set_timer(tp, tp->tmr_exp_time, tp->tmr_func);
  }
  return(OK);
}
```

```c
PRIVATE void cause_alarm(tp)
timer_t *tp;
{
/* Routine called if a timer goes off and the process requested
 * a synchronous alarm.
 * The process number is stored in timer argument 'ta_int'.
 * Notify that process with a notification message from CLOCK.
 */
  int proc_nr = tmr_arg(tp)->ta_int;      /* get process number */
  lock_notify(CLOCK, proc_nr);         /* notify process */
}
```

read()



read()


User
1
11
FS
2
8
9
10
Disk device driver
3
5
4
6
7
System task
Interrupt

Minix kernel