

# Zadanie 1 SOI

## Wywołania systemowe

Poruszanie się po deskryptorach procesów (**DESKRYPTORY PROCESÓW a nie PROCESY**)

tworzenie grup procesów, ścieżki rodzic → dziecko

## ▼ Robienie wywołań systemowych - tutorial

pre wszystko:

\*zwiększyć nrproc do 64 żeby móc mieć więcej działających procesów - w `usr/include/minix/config.h`

\* każdy przykład - robić na oddzielnym obrazie, bo tu dodajemy 2 wywołania, tu 2 - robimy to osobno

## KOMPILACJA JADRA 🐻🍪

idziemy do katalogu `/usr/src/tools` → kompilujemy i instalujemy nowe jądro - **make hdboot** → restartujemy minixa → mamy nowe jądro

`make clean` - usunięcie plików pośrednich kompilacji

\*warto zrobić backup obrazu, gdybyśmy coś zepsuli

Sukces:

```
install -S 512w fs
exec make - image
exec cc -O -D_MINIX -D_POSIX_SOURCE init.c -o init
install -S 192w init
installboot -image image ../kernel/kernel ../mm/mm ../fs/fs init
  text    data    bss    size
58928    10136   42736   111800  ../kernel/kernel
14736     1168   30420   46324   ../mm/mm
29312     2408  107356  139076  ../fs/fs
 6992     2084    1356   10432   init
-----
109968   15796  181868  307632  total
exec sh mkboot hdboot
cp image /dev/c0d0p0s0:/minix/2.0.3r0
Done.
#
```

## DODANIE WYWOŁANIA SYSTEMOWEGO 🤔👤👉

1. zmieniamy liczbe wywołań systemowych z 78 na 80, dodajemy własne 2 wywołania

\* w pliku jak definiujemy funkcje, zamiar var powinno byc variable

[/usr/include/minix/callnr.h](#)

W pliku: `/usr/include/minix/callnr.h`

- Dodać stałe identyfikujące wywołania systemowe.
- Zwiększyć stałą `N_CALLS` określającą liczbę wywołań systemowych (zmieniona wartość ze 78 na 80).

```
#define NCALLS      80 /* number of system calls allowed */

#define EXIT        1
#define FORK        2
#define READ        3
// ...
#define REBOOT      76
#define SVRCTL      77

#define GETVAR      78
#define SETVAR      79
```

2. dodaliśmy że istnieją takie wywołania systemowe, ale system nie wie jak je wykonać → musimy napisać funkcje (do\_getvar - funkcja wykonująca w.s. getvar)

2.1. **dodajemy nagłówki funkcji** (prototypy, deklarujemy) w pliku

[/usr/src/mm/proto.h](#)

```

/* Function prototypes. */

struct mproc;
struct stat;

_PROTOTYPE( int do_getvar, (void) );
_PROTOTYPE( int do_setvar, (void) );

/* alloc.c */
_PROTOTYPE( phys_clicks alloc_mem, (phys_clicks clicks)

```

\*modyfikujemy kod serwera mm (message management)

2.2. musimy gdzieś **dodać ciała funkcji** - do pliku z serwera mm, np. do [/usr/src/mm/misc.c](#)

```

#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include <sys/svrctl.h>
#include "mproc.h"
#include "param.h"

static int variable = 100;

PUBLIC int do_getvar()
{
    return variable;
}

PUBLIC int do_setvar()
{
    variable = mm_in.m1_i1;
    return variable;
}

```

2.3. musimy jeszcze powiedzieć serwerom mm i fs co mają robić jak dostaną takie wywołanie systemowe

- filesystemowi - mówimy no\_sys, null, nie obsługuj

plik [/usr/src/fs/table.c](#)

```

#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "super.h"

PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
    no_sys,          /* 0 = unused */
    do_exit,         /* 1 = exit */
    do_fork,         /* 2 = fork */
    //...
    do_svrctl,       /* 77 = SVRCTL */
    no_sys,          /* 78 = getvar */
    no_sys,          /* 79 = setvar */
};
/* This should not fail with "array size is negative": */
extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 :
-1];

```

- memorymanagerowi - mówimy jaką funkcję ma wywołać (formalnie: podajemy *wskazanie na funkcję*)

plik [/usr/src/mm/table.c](#)

```

#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/* Miscellaneous */
char core_name[] = "core"; /* file name where core images are produced
*/

_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
    no_sys,          /* 0 = unused */
    do_mm_exit,      /* 1 = exit */
    do_fork,         /* 2 = fork */
    //...
    do_svrctl,       /* 77 = svrctl */
    do_getvar,       /* 78 = getvar */
    do_setvar,       /* 79 = setvar */
};
/* This should not fail with "array size is negative": */
extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 :
-1];

```

UWAGA: w tych wszystkich plikach - pusta linia a koncu

modyfikujemy te pliki → robimy nowe jądro → restartujemy minixa →  
wywołujemy plik c

**PROGRAMY TESTUJĄCE DODANE WYWOŁANIA SYSTEMOWE** 🔍🐜😄😡

**na końcu każdego pliku źródłowego (również nagłówkowego) musi być  
pusta linia !!!!!**

1. napisać w czymś plik C
2. winscp, przerzucić na minixa (przerzuciłam do usr/minix, a potem move plik1  
plik2 /usr/my\_files)
3. jeśli plikom zepsuje się formatowanie to poprawić je w vimie
  - nie może być polskich znaków
  - nie może być komentarzy //
  - na końcu pliku musi być pusta linia
  - musi mieć uprawnienia write w danym folderze, pliku
  - nazwa czasami jest za długa?
  - **JEŚLI** będzie to dziwne^M na końcu linii → w vimie `:%s/^M//g`
4. kompilujemy  
cc <nazwapliku> -o <plikwynikowy>  
np. cc getvar.c -o getvar
5. uruchamiamy  
./getvar

```
# cc setvar.c -o setvar
# ls
getvar  getvar.c  setvar  setvar.c
# ./getvar
syscall return: 100
# ./setvar 50
set variable 50
syscall return: 50
# ./getvar
syscall return: 50
```

## JESZCZE O TESTUJĄCYCH

programy testujące - sety - pobierające wartość od użytkownika:

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

int main( int argc, char ** argv )
{
    int value;
    message m;
    int ret;
    if( argc != 2)
        return 0;
    value = atoi(argv[1]);
    printf( "set variable %d\n", value);
    m.m1_i1 = value;
    ret = _syscall( MM, SETVAR, & m );
    printf( "syscall return: %d\n", ret );
    return 0;
}
```

→ gdy używamy tego typu programu → ./setvar 50

gety - po prostu tak:

```

#include <stdio.h>
#include <lib.h>
int main()
{
    message m;
    int ret = _syscall( MM, GETVAR, & m );
    printf( "syscall return: %d\n", ret );
    return 0;
}

```

→ ./getvar

## podsumowanie które pliki edytujemy

- [/usr/include/minix/callnr.h](#) - (zmienić liczbę wyw.sys.), dodać stałe (liczby) reprezentujące wyw.systemowe
- [usr/src/mm/proto.h](#) - dodajemy nagłówki funkcji obsługujących wywołania
- [usr/src/mm/misc.c](#) - definicja, ciało tych funkcji
- [usr/src/mm/table.c](#), [usr/src/fs/table.c](#) - mówimy serwerom co mają robić jak pojawi się wywołanie systemowe

▼ podsumowanie



## Kompilacja jądra

```
/usr/src/tools → make hdbboot
```

## Pokazywanie nrów linii

```
:set nu
```

## Usuwanie ^M z plików

```
:%s/^M//g
```

## ▼ Działanie na pidach - tutorial

```
int childrenCount( int proc_nr )
{
    int children = 0;
    int i = 0;
    for ( i = 0; i < NR_PROCS; ++i)
        if ((mproc[i].mp_flags & IN_USE) && proc_nr != i && (mproc[i].mp_parent == proc_nr))
            ++children;
    return children;
}
```



```

}

void maxChildren( int * children, pid_t * who )
{
    int maxChildren = -1;
    pid_t found = -1;
    int count = 0;
    int proc_nr = 0;
    for (proc_nr = 0; proc_nr < NR_PROCS; ++proc_nr)
    {
        if (mproc[proc_nr].mp_flags & IN_USE)
        {
            count = childrenCount( proc_nr );
            if (count > maxChildren)
            {
                maxChildren = count;
                found = mproc[proc_nr].mp_pid;
            }
        }
    }
    *children = maxChildren;
    *who = found;
}

PUBLIC int do_maxChildren()
{
    int children = -1;
    pid_t who = -1;
    maxChildren( & children, & who );
    return children;
}

PUBLIC int do_whoMaxChildren()
{
    int children = -1;
    pid_t who = -1;
    maxChildren( & children, & who );
    return who;
}

```

dostęp przez wartosc, referencje, wskaznik

tutaj (& children) - przez wskazanie - mamy adres zmiennej

**if ((mproc[i].mp\_flags & IN\_USE)**

to sprawdzenie - sprawdzenie czy ten deskryptor jest używany  
& to OR bitowy

**proc\_nr != i**

sprawdzenie czy proces nie jest swoim rodzicem (init tak ma)

jak zrobimy już te 2 funkcje to chcemy je przetestować, więc zrobimy funkcję która tworzy proces z ilomaś dziećmi, potem sprawdzamy czy wynik się zgadza

for - każde przejście tworzy dziecko głównego procesu

\*jesli rodzic umrze, staje się zombie; rodzicem dzieci staje się init

\*init ma zawsze najwięcej dzieci

\*te zadania możemy robić rekursywnie, zazwyczaj unikac

\*używamy sleep

## ▼ Zadanie moje

**Zwrócić pid procesu mającego największą liczbę potomków (dzieci, wnuków ...), zwrócić liczbą potomków dla tego procesu. Pomiąć proces o podanym w parametrze identyfikatorze pid.**

- jakos stworzyć dzieci
- jeśli to ważne, dopilnować żeby rodzic nie umarł przed dziećmi, bo wtedy pid rodzica tych dzieci zmieni się na 0(init) i wszystko pójdzie w gwizdek - sleep
- zrobić rekurencyjnie (tylko uważnie)

./zadanie & → F1 → kill pidnr

for proces in tablica deskryptorów:

biore pierwszego, patrzę jaki ma pid

na koniec, jeśli dzieci więcej niż aktualny max, aktualizuje

- ```
# ./test_basic 0
Ignored pid 0 goal |Ignored pid 0 goal |Scanning pid 1 |CURRMAX children 5 proc
1 |Scanning pid 19 |Scanning pid 22 |Scanning pid 15 |Scanning pid 16 |Scan
ning pid 39 |Ignored pid 37 inact |Ignored pid 24 inact |Ignored pid 0 inact |I
gnored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inac
t |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0
inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pi
d 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignore
d pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ig
nored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact
|Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 i
nact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid
0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored
pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ign
ored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact
|Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 in
act |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid
0 inact |Ignored pid 0 inact |Ignored pid 0 inact |Ignored pid 0 inact |found: 1
#
```

- ```
if (argc != 2)
    return 0;
pid_to_ignore = atoi(argv[1]);
m.m1_i1 = pid_to_ignore;

max = _syscall( MM, MAXOFFSPRINGS, & m );
printf( "Max nr of offsprings: %d\n", max );
m.m1_i1 = pid_to_ignore;

who = _syscall( MM, WHOMAXOFFSPRINGS, & m );
printf( "Who: %d\n", who );
return 0;
```

## SHOWCASE

```
#
# ./basic_test 0
Max nr of offsprings: 5
Who: 1
#
# ./basic_test 1
Max nr of offsprings: 1
Who: 19
#
# ./custom_test 0 0
creating 0 children
Max offsprings: 5
Who: 1
#
# ./custom_test 0 3
creating 3 children
Max offsprings: 8
Who: 1
#
# ./custom_test 1 3
creating 3 children
Max offsprings: 4
Who: 19
#
#
```

basic\_test param1

szuka procesu o największej liczbie potomków oraz liczby tych potomków, wypisuje je

nie bierze pod uwagę procesu o podanym w argumencie pidzie

custom\_test param1 param2

program tworzy #param2 podprocesów, a następnie tak samo jak wyżej

dzięki tworzeniu określonej liczby dzieci i ignorowaniu pidu inita, można przetestować działanie wywołań systemowych

- wykonane rekurencyjnie

## ▼ Wnioski zadanie 1

\*możemy używać printa w minixie!

Na ost. zajęciach mowiliśmy że ppid inita to 1 (on sam), ALE w tej wersji/ogólnie jest to jednak ppid 0 - brak rodzica

pid = 0, ppid = 0 → proces systemowy, specjalny (w zadaniach nie bierzemy ich pod uwagę)

ps - wyświetla aktywne procesy

ps -le - więcej informacji

np.

init pid1 ppid 0

sh (shell) pid19 ppid 1

ps -le (wywołane przed chwilą polecenie) pid 29 ppid 19

minix → kliknięcie F1/F2 → tabelki z deskryptorami procesów

kolumna user - liczba kwantów czasu, jakie dostał dany proces

./a.out & - wykonaj proces w tle

kill 38 - zabij proces o pidzie 38