Laboratorium 1 - Kompilacja jadra i wywolania systemowe

Laboratorium 1

Autor: Jakub Mierzejewski

Data: 9 stycznia 2026

System: MINIX 2.0

1. Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z mechanizmem realizacji wywołań systemowych (ang. system calls) w systemie MINIX. W ramach laboratorium należało dodać do systemu 2 nowe wywołania systemowe, które przeglądają listę deskryptorów procesów i zwracają informacje o procesach.

Wywołania systemowe są to funkcje realizowane przez jądro systemu operacyjnego w kontekście danego procesu. Jest to jedyny dopuszczalny dla procesów użytkowych sposób wejścia do jądra systemu.

2. Zadanie

Zaimplementowano 2 wywołania systemowe realizujące następującą funkcjonalność:

Wywołanie 1: `GET_MOST_CHILDREN` (syscall nr 78)

- Zwrócić PID procesu mającego najwięcej dzieci

- Zwrócić liczbę dzieci takiego procesu

Wywołanie 2: `GET_MOST_DESCENDANTS` (syscall nr 79)

- Zwrócić PID procesu mającego największą liczbę potomków (dzieci, wnuków, itp.)

- Zwrócić liczbę potomków dla tego procesu

- Pominąć proces o podanym w parametrze identyfikatorze PID

3. Implementacja

3.1. Dodanie definicji wywołań systemowych

Dodano definicje nowych wywołań systemowych na końcu pliku:

```
#define GET_MOST_CHILDREN 78
#define GET_MOST_DESCENDANTS 79
```

Zmodyfikowano również stałą określającą liczbę wywołań systemowych:

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

3.2. Deklaracja prototypów funkcji

Dodano prototypy nowych funkcji obsługujących wywołania systemowe:

```
/* main.c */
_PROTOTYPE( int do_most_children, (void) );
_PROTOTYPE( int do_most_descendants, (void) );
```

3.3. Rejestracja w tablicy wywołań systemowych

W tablicy `call_vec` dodano wskaźniki do nowych funkcji:

```
_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
    // ... istniejące wywołania ...
    do_most_children,      /* 78 = GET_MOST_CHILDREN */
    do_most_descendants,   /* 79 = GET_MOST_DESCENDANTS */
};
```

W tablicy `call_vec` w module FS dodano wpisy `no_sys` dla tych wywołań (są obsługiwane przez MM):

```
_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
    // ... istniejące wywołania ...
```

```
    no_sys,    /* 78 = GET_MOST_CHILDREN */
    no_sys,    /* 79 = GET_MOST_DESCENDANTS */
};
```

3.4. Implementacja funkcji obsługujących

#Funkcja `do_most_children()`

Funkcja przeszukuje tablicę procesów `mproc[]` i dla każdego procesu zlicza liczbę jego bezpośrednich dzieci. Zwraca PID procesu z największą liczbą dzieci oraz samą liczbę dzieci.

```
PUBLIC int do_most_children(void)
{
    int max_children;
    int max_pid;
    int i;
    int j;
    int children_count;

    max_children = 0;
    max_pid = 0;
    children_count = 0;

    /* Przeglądanie wszystkich procesów */
    for (i = 0; i < NR_PROCS; i++) {
        if (!(mproc[i].mp_flags & IN_USE)) continue;

        children_count = 0;
        /* Zliczanie dzieci procesu i */
        for (j = 0; j < NR_PROCS; j++) {
            if ((mproc[j].mp_flags & IN_USE) &&
                mproc[j].mp_parent == i) {
                children_count++;
            }
        }

        /* Aktualizacja maksimum */
        if (children_count > max_children) {
            max_children = children_count;
            max_pid = mproc[i].mp_pid;
        }
    }

    /* Zwrócenie wyników przez strukturę message */
    mproc[mm_in.m_source].mp_reply.m2_i1 = max_children;
    mproc[mm_in.m_source].mp_reply.m2_i2 = max_pid;

    return 1;
}
```

1. Inicjalizacja zmiennych przechowujących maksimum

2. Iteracja po wszystkich procesach w tablicy `mproc[]`

3. Dla każdego aktywnego procesu (flaga `IN_USE`):

   - Zliczenie liczby procesów, dla których jest rodzicem

4. Aktualizacja maksimum gdy znaleziono proces z większą liczbą dzieci

5. Zwrócenie wyników w polach `m2_i1` (liczba dzieci) i `m2_i2` (PID)

#Funkcja pomocnicza `count_descendants()`

Funkcja rekurencyjna zliczająca wszystkich potomków danego procesu (dzieci, wnuków, prawnuków, itd.).

```
PRIVATE int count_descendants(int process_index)
```

```
{
    int count;
    int j;

    count = 0;

    /* Rekurencyjne zliczanie potomków */
    for (j = 0; j < NR_PROCS; j++) {
        if ((mproc[j].mp_flags & IN_USE) &&
            mproc[j].mp_parent == process_index) {
            count += 1;                         /* Dziecko */
            count += count_descendants(j);      /* Potomkowie dziecka */
        }
    }

    return count;
}
```

#Funkcja `do_most_descendants()`

Funkcja przeszukuje tablicę procesów i dla każdego procesu rekurencyjnie zlicza wszystkich jego potomków. Pomija proces o PID podanym jako parametr.

```
PUBLIC int do_most_descendants(void)
{
    int max_descendants;
    int max_pid;
    int i;
    int arg_pid;
    int descendants;

    max_descendants = 0;
    max_pid = 0;
    arg_pid = mm_in.m1_i1;   /* PID do pominięcia */
    descendants = 0;

    for (i = 0; i < NR_PROCS; i++) {
        descendants = 0;

        if (!(mproc[i].mp_flags & IN_USE)) continue;

        /* Pomijamy proces init (PID 0) i proces podany przez użytkownika */
        if (mproc[i].mp_pid != 0 && mproc[i].mp_pid != arg_pid) {
            descendants = count_descendants(i);
        }

        if (descendants > max_descendants) {
            max_descendants = descendants;
            max_pid = mproc[i].mp_pid;
        }
    }

    /* Zwrócenie wyników przez strukturę message */
    mproc[mm_in.m_source].mp_reply.m1_i1 = max_descendants;
    mproc[mm_in.m_source].mp_reply.m1_i2 = max_pid;

    return 1;
}
```

1. Pobranie PID procesu do pominięcia z parametru (`mm_in.m1_i1`)

2. Iteracja po wszystkich procesach

3. Dla każdego aktywnego procesu (z wyjątkiem init i procesu o podanym PID):

- Rekurencyjne zliczenie wszystkich potomków

4. Aktualizacja maksimum

5. Zwrócenie wyników w polach `m1_i1` (liczba potomków) i `m1_i2` (PID)

4. Kompilacja i instalacja

4.1. Kompilacja nowego jadra

```
cd /usr/src/tools
make hdboot
```

Ten proces:

- Kompiluje wszystkie zmodyfikowane moduły (MM, FS, kernel)

- Łączy je w jedno bootowalne jądro

- Tworzy dyskietkę startową z nowym jądrem (w przypadku systemu na dyskietce)

4.2. Restart systemu

```
cd
shutdown
# Po restarcie
boot
```

System uruchamia się z nowym jądrem zawierającym zaimplementowane wywołania systemowe.

5. Demonstracja i testowanie

5.1. Wywołanie funkcji systemowych z programu użytkowego

Ponieważ nowe wywołania systemowe nie mają standardowych funkcji opakowujących w bibliotece, należy użyć funkcji `_syscall()` do ich bezpośredniego wywołania.

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

#define GET_MOST_CHILDREN 78
#define GET_MOST_DESCENDANTS 79

int main(void)
{
    message m;
    int result;

    /* Test 1: Wywołanie GET_MOST_CHILDREN */
    result = _syscall(MM, GET_MOST_CHILDREN, &m);
    if (result == 0) {
        printf("Proces z najwieksza liczba dzieci:\n");
        printf("  PID: %d\n", m.m2_i2);
        printf("  Liczba dzieci: %d\n", m.m2_i1);
    } else {
        printf("Blad wywolania GET_MOST_CHILDREN\n");
    }

    /* Test 2: Wywołanie GET_MOST_DESCENDANTS */
    m.m1_i1 = 1;  /* Pomijamy proces o PID=1 (init) */
    result = _syscall(MM, GET_MOST_DESCENDANTS, &m);
    if (result == 0) {
        printf("\nProces z najwieksza liczba potomkow (pomijajac PID=1):\n");
        printf("  PID: %d\n", m.m1_i2);
        printf("  Liczba potomkow: %d\n", m.m1_i1);
    } else {
        printf("Blad wywolania GET_MOST_DESCENDANTS\n");
    }

    return 0;
```

```
}
```

5.2. Scenariusz testowy

1. **Utworzenie hierarchii procesów:**

   ```bash
   # Utworzenie kilku procesów potomnych w tle
   sleep 100 &
   sleep 100 &
   bash -c "sleep 100 & sleep 100 & sleep 100 & wait" &
   ```

2. **Uruchomienie programu testowego:**

   ```bash
   ./test_syscalls
   ```

3. **Weryfikacja wyników:**

   - Program wyświetli PID procesu z największą liczbą dzieci
   - Program wyświetli PID procesu z największą liczbą wszystkich potomków
   - Wyniki można zweryfikować za pomocą polecenia `ps -ef` lub `pstree`

5.3. Przykładowe wyniki

```
Proces z najwieksza liczba dzieci:
  PID: 245
  Liczba dzieci: 3

Proces z najwieksza liczba potomkow (pomijajac PID=1):
  PID: 245
  Liczba potomkow: 5
```

6. Ważne struktury i zmienne

6.1. Struktura `mproc`

Tablica `mproc[]` w module MM przechowuje informacje o wszystkich procesach. Ważne pola:

- `mp_flags` - flagi procesu (w tym `IN_USE` oznaczająca aktywny proces)
- `mp_parent` - indeks procesu rodzica w tablicy `mproc[]`
- `mp_pid` - PID procesu
- `mp_reply` - struktura `message` do zwracania wyników

6.2. Struktura `message`

Służy do komunikacji między procesami użytkownika a modułami systemu:

- `m1_i1`, `m1_i2` - pola integer w formacie message typu 1
- `m2_i1`, `m2_i2` - pola integer w formacie message typu 2
- `m_source` - PID procesu wywołującego

6.3. Zmienne globalne w MM

- `mm_in` - struktura `message` z danymi wejściowymi od procesu wywołującego
- `mm_out` - struktura `message` z danymi wyjściowymi (niewykorzystana w tym przypadku)
- `who` - PID procesu wywołującego

- `mp` - wskaźnik do `mproc` procesu wywołującego

7. Problemy i rozwiązania

7.1. Problem: Błędne zliczanie potomków

7.2. Problem: Uwzględnianie procesu init

7.3. Problem: Sprawdzanie aktywności procesów

8. Wnioski

1. **Implementacja wywołań systemowych w MINIX** wymaga modyfikacji kilku plików:
   - Definicje w `callnr.h`
   - Prototypy w `proto.h`
   - Rejestracja w `table.c` (zarówno MM jak i FS)
   - Implementacja w odpowiednim module (MM lub FS)

2. **Struktura mproc** zawiera wszystkie niezbędne informacje o procesach, w tym relacje rodzic-dziecko przechowywane jako indeksy w tablicy.

3. **Komunikacja z wywołaniami systemowymi** odbywa się przez struktury `message`, które mają różne formaty (`m1_`, `m2_`, etc.) dla różnych typów danych.

4. **Rekurencyjne algorytmy** są naturalnym wyborem do przetwarzania hierarchii procesów, ale należy uważać na głębokość rekurencji.

5. **Testowanie** nowych wywołań systemowych wymaga:
   - Rekompilacji jądra
   - Restartu systemu
   - Utworzenia programów testowych używających `_syscall()`
   - Weryfikacji wyników za pomocą narzędzi systemowych

6. **Bezpieczeństwo**: Implementacja sprawdza poprawność danych (flaga `IN_USE`) i unika przetwarzania nieaktywnych procesów.

7. **Wydajność**: Złożoność czasowa obu algorytmów to $O(n^2)$ dla `do_most_children()` i $O(n \cdot h)$ dla `do_most_descendants()` gdzie n to liczba procesów, a h to wysokość drzewa procesów.

9. Bibliografia

1. Tanenbaum A.S., Woodhull A.S., "Operating Systems: Design and Implementation", 2nd Edition

2. Dokumentacja systemu MINIX 2.0

3. Materiały z kursu "Systemy Operacyjne i Sieci Komputerowe"

4. Pliki źródłowe systemu MINIX w katalogach `/usr/src/mm`, `/usr/src/fs`, `/usr/include/minix`

Załączniki

Plik: `/usr/include/minix/callnr.h` (fragment)

```
#define NCALLS       80     /* number of system calls allowed */
// ...
#define GET_MOST_CHILDREN 78
#define GET_MOST_DESCENDANTS 79
```

Plik: `/usr/src/mm/proto.h` (fragment)

```
/* main.c */
_PROTOTYPE( void main, (void)                       );
_PROTOTYPE( int do_most_children, (void) );
_PROTOTYPE( int do_most_descendants, (void) );
```

Plik: `/usr/src/mm/table.c` (fragment)

```
_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
    // ...
    do_most_children,      /* 78 = GET_MOST_CHILDREN */
    do_most_descendants,   /* 79 = GET_MOST_DESCENDANTS */
};
```