Sisteme de Operare

- Note de laborator -

Facultatea de Automatică şi Calculatoare
Universitatea Politehnica Bucureşti
Grupa 332, Seria CA
Anton Puiu
anton.puiu@email.com

Contents

1 Laborator 1 - Introducere

1.1 Linux

1.1.1 Apeluri de sistem

Un **apel de sistem** este un punct de intrare **controlat** în **kernel**, permiţînd unui proces să facă o cerere sistemului de operare, astfel încît acesta să efectueze o serie de acţiuni în numele său. Sistemul de operare pune la dispoziţie o serie de funcţionalităţi accesibile proceselor prin intermediul acestor **apeluri de sistem**, totalitatea acestora fiind înglobată în **API**. Serviciile pe care le oferă **sistemul de operare** sunt, de exemplu, crearea unui nou proces, operaţii de intrare/ieşire, crearea unui **pipe** utilizat pentru comunicaţia dintre procese.

Caracteristici generale

- Un **apel de sistem** schimbă starea procesorului din **user-mode** în **kernel-mode**, astfel încît procesorul să poată accesa zona protejată de memorie în care se află **sistemul de operare**.
- API-ul **apelurilor de sistem** este fix. Fiecare apel de sistem este identificat printr-un număr unic.
- Fiecare **apel de sistem** poate avea un set de argumente ce specifică informaţii ce trebuie transferate din *user-space* în *kernel-space*.

Realizarea unui apel de sistem
Din punct de vedere programatic, un apel de sistem poate fi asociat cu apelul unei funcţii. Totuşi, în practică, se execută mult mai mulţi paşi în timpul execuţiei unui apel de sistem. Spre exemplu, pentru arhitectura **x86**, paşii sunt următorii:

1. Programul aplicaţie efectuează apelul de sistem prin invocarea unei funcţii *wrapper* din biblioteca standard **C**.
2. Funcţia *wrapper* trebuie să pună la dispoziţie **rutinei de tratare a apelului de sistem** toate argumentele necesare apelului de sistem. Aceste argumente sunt pasate către funcţia *wrapper* prin intermediul stivei, dar acestea trebuie depozitate în anumite registre. Astfel, funcţia va copia valorile de pe stivă în respectivele registre ale procesorului.
3. Întrucît toate apelurile de sistem intră în **kernel** prin acelaşi loc, acesta necesită o metodă de identificare a apelului de sistem. Pentru a permite acest lucru, funcţia **wrapper** copiază numărul apelului de sistem în registrul **eax**.
4. Funcţia wrapper execută o **întrerupere software** prin intermediul instrucţiunii **int 0x80**, iar astfel procesorul îşi schimbă starea de lucru din **user-mode** în **kernel-mode** şi execută codul aflat la locaţia **0x80** din **vectorul de tratare a întreruperilor**.
5. Drept consecinţă a execuţiei instrucţiunii aflate la adresa **0x80**, **kernel**-ul îşi invocă rutina *system_call()* pentru a trata întreruperea. Această rutină execută următoarele acţiuni:
   - Salvează valoarea registrelor pe **stiva sistemului de operare**.
   - Verifică corectitudinea numărului apelului de sistem.

- Apelează rutina de tratare a apelului de sistem asociată cu numărul apelului de sistem. Pentru a realiza acest lucru, se utilizează o tabelă de mapare **număr apel de sistem-rutină tratare apel de sistem**, în care numărul apelului de sistem joacă rol de **index**. Dacă rutina de tratare a apelului de sistem are argumente, întîi se verifică validitatea acestora, după care rutina execută operaţia cerută, care ar putea implica modificarea anumitor valori la adresele specificate drept argument sau transferul de date între **memoria sistemului de operare** şi **memoria procesului**. În final, rutina întoarce un rezultat interpretat drept **status**.
- Se restaurează valorile registrelor de pe **stiva sistemului de operare** şi se pune în vîrful stivei aplicaţiei valoarea de retur a apelului de sistem.
- Se revine în funcţia *wrapper* şi în acelaşi timp procesorul comută în starea **user-mode**.

6. Dacă valoarea de retur a rutinei apelului de sistem indică o eroare, funcţia *wrapper* modifică valoarea variabilei globale **errno** utilizînd această valoare. Funcţia *wrapper* returnează către funcţia apelantă o valoare indicînd succesul sau eşecul apelului de sistem.

Concluzii
Avînd în vedere seria de operaţii necesare astfel încît un apel de sistem să fie executat, se observă faptul că un apel de sistem este o operaţie **costisitoare**, iar pentru a obţine performanţă maximă, atunci cînd un apel de sistem este efectuat, acesta să fie efectuat pentru cît mai multe cazuri posibile, astfel încît să se reducă numărul de apeluri de sistem.

1.1.2 Funcţii din biblioteci

Scopul acestor funcţii este diversificat şi include task-uri precum deschiderea unui fişier, conversia timpului la un format *human-readable*, compararea a două şiruri de caractere, etc.

Multe astfel de funcţii nu utilizează apeluri de sistem. Pe de altă parte, anumite funcţii din biblioteca standard **C** sunt definite peste un apel de sistem, în primul rînd pentru a oferi o interfaţă mai uşoară de lucru, dar poate cel mai important motiv este **portabilitatea**. Astfel, dacă o astfel de bibliotecă există şi maschează un apel de sistem, o aplicaţie ce utilizează respectiva bibliotecă poate fi compilată de asemenea pe orice alt sistem de operare pentru care există o implementare a respectivei biblioteci.

1.1.3 Tratarea erorilor din apeluri de sistem şi funcţii din biblioteci

Aproape orice apel de sistem şi funcţie din biblioteca standard returnează o valoare ce indică starea finală a respectivului apel. Această stare trebuie **întotdeauna** verificată. În cazul unei erori, programul **trebuie** să afişeze un mesaj de eroare. Acest principiu poate salva în prealabil multe ore de debugging.

Tratarea erorilor apelurilor de sistem
Pentru fiecare apel de sistem, se regăseşte, în pagina de manual asociată, valorile posibile returnate. În general, o funcţie din biblioteca standard întoarce valoarea -1, semnalînd astfel o eroare. Pentru a obţine detalii despre eroarea apărută, putem verifica variabila globală *errno*, setată ori de căte ori un apel de sistem eşuează, iar prin intermediul acesteia se poate identifica cauza erorii. De asemenea, în pagina de manual a fiecărui apel de sistem se regăsesc valorile posibile pe care variabila *errno* le poate avea în cazul în care respectivul apel de sistem eşuează.

Este de menţionat faptul că în general un apel de sistem nu schimbă valoarea variabilei *errno* la 0 în cazul execuţiei cu succes, aceasta fiind actualizată de la o **eroare la alta**. Astfel, înainte de a verifica valoarea variabilei *errno*, trebuie verificată valoarea întoarsă de funcţia apelată din biblioteca standard.

Anumite apeluri de sistem pot întoarce valoarea -1 în cazul execuţiei cu succes. Avînd în vedere acest fapt, ajungem la concluzia că înainte de a utiliza orice apel de sistem, trebuie verificată pagina de manual a acestuia. În cazul în care apelul de sistem întoarce -1 la succes, putem reseta valoarea variabilei *errno* la 0 înaintea apelului, urmînd ca ulterior să verificăm dacă respectiva valoare a fost modificată.

O practică comună după eşuarea unui apel de sistem este afişarea anumitor mesaje de eroare, bazate pe valoarea variabilei *errno*. Funcţiile *perror* şi *strerror* sunt puse la dispoziţie pentru acest scop. Funcţia *perror* afişează mesajul trimis ca argument, urmat de mesajul asociat variabilei *errno*. Pentru a obţine mesajul asociat unei anumite valori a variabilei *errno*, se poate utiliza funcţia *strerror*, care primeşte ca argument o valoare posibilă a variabilei *errno* şi întoarce mesajul de eroare asociat. Merită să fie menţionat faptul că şirul de caractere returnat de funcţia *strerror* ar putea fi alocat static, iar astfel la apeluri repetate acesta să fie suprascris.

Tratarea erorilor apelurilor funcţiilor din biblioteci
Funcţiile din biblioteci pot fi împărţite în următoarele categorii:

- Anumite funcţii întorc informaţii legate de erori la fel ca şi apelurile de sistem: valoarea întoarsă -1 şi *errno* indicînd eroarea. Erorile generate de astfel de funcţii se pot diagnostica în acelaşi mod ca şi erorile generate de apeluri de sistem.
- Anumite funcţii pot întoarce o valoare diferită de -1, dar schimbă valoarea variabilei *errno*. Funcţiile *perror* şi *strerror* pot fi utilizate în acest caz.
- Alte funcţii nu folosesc deloc variabila *errno*. Pentru a diagnostica aceste funcţii, trebuie consultată pagina de manual. Utilizarea variabilei *errno* sau a oricărei funcţii care utilizează această variabilă reprezintă o eroare.

1.2 Windows

1.2.1 Principii Windows

Windows necesită un anumit coding style şi tehnică. Anumite caracteristici ale Windows sunt prezentate în continuare:

- Majoritatea resurselor sistemului sunt reprezentate drept **obiecte kernel**, identificate şi referite printr-un **handle**.
- Obiectele kernel trebuie manipulate prin intermediul **API**-ului Windows.
- Obiectele kernel sunt, spre exemplu, fişiere, procese, thread-uri, pipe-uri pentru comunicaţia între procese, mapare de memorie, evenimente, etc. Obiectele au atribute de securitate. Practic orice cerere a sistemului de operare ce are drept consecinţă crearea unor obiecte în kernel, respectivele obiecte pot fi referite prin acest **handle**.
- Windows pune la dispoziţie o interfaţă flexibilă. În primul rînd, conţine multe funcţii ce realizează aceleaşi operaţii, aceste funcţii fiind definite drept *primitive*; există de asemenea funcţii ce grupează un set de funcţii primitive într-un singur apel, astfel fiind creată o **funcţie wrapper**. În al doilea rînd, o funcţie are în general mulţi parametri, dar o bună parte din aceşti parametri pot fi ignoraţi.
- În Windows unitatea de bază de execuţie este **thread**-ul. Un proces poate conţine mai multe thread-uri.

- Numele funcţiilor în windows sunt lungi şi descriptive şi există o convenţie pentru denumirea acestora.
- Numele tipurilor predefinite, utilizate de către API-ul Windows, sunt scrise cu majuscule şi sunt de asemenea descriptive.
- Tipurile predefinite evită operatorul '*' şi fac distincţii între LPTSTR (definit ca TCHAR *) şi LPCTSTR (definit ca const TCHAR *).
- Există de asemenea convenţii pentru numele variabilelor din cadrul prototipurilor funcţiilor. De exemplu, lpszFileName ar putea reprezenta *pointer la un tip de date* ***long*** *către un şir de caractere care se termină cu valoarea zero reprezentînd un nume de fişier.* Această notaţie se numeşte **notaţie Ungurească**. Similar, dwAccess reprezintă un cuvînt dublu (32 de biţi) conţinînd permisiuni de fişier; 'dw' reprezintă **double word**.

Avînd în vedere faptul că Windows API a fost creat de la zero, a fost proiectat să fie compatibil cu Windows 3.1 Win16 API. Acest fapt aduce împreună cu avantajul compatibilităţii anumite efecte neplăcute.

- Există tipuri de date duplicate.
- WIN32 poate apărea uneori în denumiri de tip macro, cum ar fi WIN32_FIND_DATA , chiar dacă aceste macro-uri sunt utilizate uneori în Win64.
- Există o serie de funcţii definite pentru a opera cu 16 biţi, care poartă denumiri sugestive, iar aceste denumiri sunt ocupate. De exemplu, OpenFile este o astfel de funcţie; pentru a deschide un fişier existent, se utilizează CreateFile .

Spre deosebire de Linux, care pentru un fişier asociază un descriptor de fişier, fiind reprezentat printr-un întreg obţinut prin incrementarea unei valori, HANDLE-urile nu sunt întregi alocaţi într-o anumită ordine. De asemenea, în Windows nu există diferenţă în reprezentarea identificatorilor de proces şi descriptorilor de fişier, sau mai general, obiectelor kernel, acestea fiind reprezentate prin intermediul HANDLE-urilor. O distincţie importantă între cele două platforme se regăseşte în contextul conceptului de proces. Procesele în Windows nu au o relaţie tată-fiu, acestea fiind organizate în obiecte de tip **job**. O altă deosebire între aceste două sisteme de operare este reprezentată de terminatorul de linie: Linux utiilizează caracterul LF, în timp ce Windows utilizează CR-LF.

2 Laborator 2 - Operaţii de intrare/ieşire

2.1 Linux

Toate apelurile de sistem ce efectuează operaţii de intrare/ieşire se referă la deschiderea fişierelor utilizînd un **descriptor de fişier**. Un descriptor de fişier este reprezentat de un întreg pozitiv. Acesta este utilizat pentru a referi orice tip de fişier, inclusiv pipe-uri, cozi, socketi, terminale, dispozitive şi fişiere obişnuite.

Fiecare proces are propriul set de descriptori de fişiere

Prin convenţie, orice program se aşteaptă să poată utiliza trei descriptori de fişier:

- **intrarea standard** - fişierul din care un proces va putea **citi** date - valoarea 0;
- **ieşirea standard** - fişierul în care un proces va putea **scrie** date de ieşire - valoarea 1;
- **ieşirea standard de eroare** - fişierul în care un proces va potea **scrie** date de eroare - valoare 2.

Aceşti descriptori de fişier sunt **moşteniţi** de către procesul copil din partea procesului părinte. Dacă, spre exemplu, se doreşte redirectarea rezultatelor sau schimbarea fişierului de intrare, se pot schimba valorile variabilelor **stdin, stdout, stderr** prin intermediul funcţiei freopen() .

2.1.1 Universalitatea I/O

În UNIX există conceptul de universalitate pentru orice operaţie de intrare/ieşire. Acest concept se referă la faptul că orice operaţie de intrare/ieşire se realizează prin intermediul a patru apeluri de sistem:

- open(pathname, flags, mode).
- read(fd, buffer, count).
- write(fd, buffer, count).
- close(fd).

Drept consecinţă, dacă spre exemplu se dezvoltă un program utilizînd doar aceste apeluri de sistem, acel program va funcţiona pe orice tip de fişier.

Acest concept se bazează pe faptul că orice sistem de fişiere şi orice driver pentru dispozitiv implementează acelaşi set de apeluri de sistem pentru operaţii de intrare/ieşire. Întrucît detaliile specifice sistemelor de fişiere sau dispozitivelor sunt interpretate de către sistemul de operare, în general putem ignora caracteristicile dispozitivelor sau sistemelor de fişiere în momentul în care se dezvoltă un program aplicaţie. Atunci cînd este important să se efectueze anumite operaţii în legătură cu dispozitivul pe care se scrie sau sistemul de fişiere, un program poate utiliza apelul de sistem ioctl(), care pune la dispoziţie o interfaţă diferită de modelul universal de intrare/ieşire.

open()

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

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

Acest apel de sistem fie deschide un fişier existent, fie crează fişierul şi îl deschide. Fişierul ce trebuie deschis este identificat de argumentul *pathname*. Dacă acesta este un link simbolic, este dereferenţiat. La succes, acest apel de sistem întoarce un descriptor de fişier asociat fişierului deschis. Dacă se petrece o eroare, open() întoarce -1 şi setează valoarea variabilei errno.

Argumentul flags este o mască de biţi ce specifică modul de acces al fişierului, utilizînd una din constrîngerile următoare:

- O_RDONLY
- O_WRONLY
- O_RDWR

Atunci cînd acest apel de sistem este utilizat pentru a crea un nou fişier, argumentul mode specifică permisiunile respectivului fişier. Dacă apelul open() nu specifică O_CREAT, mode poate fi omis.

Argumentul mode poate fi specificat drept număr (în general în baza opt), sau, de preferat, utilizînd macro-uri definite în manualul apelului de sistem.

```
/* Open existing file for reading */
fd = open(``startup'', O_RDONLY);
if (fd == -1) {
  exit(-1);
}

/*
 * Open new or existing file for reading and writing, truncating to zero
 * bytes; file permissions read+write for owner, nothing for all others
 */
fd = open(``myfile'', O_RDWR | O_CREAT | O_TRUNG, S_IRUSR | IWUSR);
if (fd == -1) {
  exit(-1);
}

/*
 * Open new or existing file for writing; writes should always
 * append to end of file
 */
fd = open(``w.log'', O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, S|IRUSR | S_IWUSR);
if (fd == -1) {
  exit(-1);
}
```

open() flags

Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if *pathname* is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on *read()* (since Linux 2.6.8)	
O_NOCTTY	Don't let *pathname* become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3
O_SYNC	Make file writes synchronous	v3

Există următoarele grupuri din care fac parte macro-urile ce pot fi utilizate în cadrul argumentului flags:

- **Flag-uri de acces**: O_RDONLY, O_WRONLY şi O_RDWR descrise anterior.
- **Flag-uri de creare**: Sunt acele flag-uri din a doua secţiune din tabelul anterior.
- **Flag-uri de deschidere**: Aceste flag-uri specifică **cum** vor fi realizate operaţiile de intrare/ieşire asupra fişierelor.

Detalii legate de constantele flag sunt date în continuare.

- O_APPEND: Scrierile sunt mereu efectuate la sfîrşitul fişierului.
- O_ASYNC: Se generează un semnal atunci cînd se pot efectua operaţii de intrare/ieşire pe descriptorul de fişier întors de open(). Acest flag, denumit **signal-driven I/O**, este valabil doar pentru anumite tipuri de fişiere, cum ar fi cozi şi socketi. Pentru a utiliza acest flag, trebuie efectuat apelul de sistem fcntl() F_SETFL.
- O_CLOEXEC: Este util în programe paralele pentru a evita anumite probleme de sincronizare. Aceste probleme pot apărea în momentul în care un thread deschide un descriptor de fişier şi după încearcă să îl marcheze close-on-exec în acelaşi timp în care un alt thread apelează fork() şi apoi exec() pentru un program arbitrar. Astfel, evitînd un alt apel de sistem necesar pentru a seta flag-ul close-on-exec al unui descriptor de fişier, sau primitive de sincronizare, se poate utiliza acest flag în schimb.
- O_CREAT: Dacă fişierul nu există deja, va fi creat drept fişier nou, gol. Acest flag este util chiar dacă fişierul este deschis doar pentru citire. Dacă se specifică acest flag, atunci trebuie specificat şi argumentul mode în cadrul apelului de sistem; altfel, permisiunile noului fişier vor fi setate drept valori arbitrare luate din stivă.

- O_DIRECT: Permite operaţiilor de intrare/ieşire să fie scrise ocolind zonele tampon. Pentru a utiliza acest flag, trebuie definit macro-ul _GNU_SOURCE.
- O_DIRECTORY: Întoarce eroare dacă primul argument nu este un director. Acest flag este o extensie proiectată pentru implementarea apelului de sistem opendir().Pentru a utiliza acest flag, trebuie definit macro-ul _GNU_SOURCE.
- O_DSYNC: Efectuează scrierile în fişier în manieră **synchronized I/O cu integritate de date**.
- O_EXCL: Acest flag este utilizat împreună cu O_CREAT pentru a indica dacă un fişier există deja, acesta nu ar trebui deschis. Astfel, apelul de sistem trebuie să eşueze, cu errno setat la EEXIST. Astfel, se asigură faptul că procesul curent este cel care crează fişierul respectiv.
- O_NOATIME: Nu se actualizează timpul de acces al fişierului atunci cînd se citeşte din respectivul fişier. Pentru a utiliza acest flag, utilizatorul care rulează programul trebuie să fie deţinătorul fişierului, sau procesul trebuie să fie privilegiat.Pentru a utiliza acest flag, trebuie definit macro-ul _GNU_SOURCE. Acest flag a fost creat pentru programe de backup. Utilizarea sa poate reduce activitatea memoriilor externe, întrucît nu trebuie modificate metadatele fişierelor.
- O_NOCTTY: Dacă fişierul ce urmează să fie deschis este un terminal, acesta nu va fi terminalul de control.
- O_NOFOLLOW: Acest flag este util atunci cînd se doreşte deschiderea unui link.Pentru a utiliza acest flag, trebuie definit macro-ul _GNU_SOURCE.
- O_SYNC: Deschide fişierul pentru **synchronous I/O cu integritate de fişiere**.
- O_TRUNC: Dacă fişierul există deja drept fişier obişnuit, atunci va şterge conţinutul acestuia.

Erori întoarse de open

- EACCES: Permisiunile nu pot fi setate sau nu există permisiuni pentru deschiderea fişierului.
- EISDIR: Fişierul specificat este un director şi se încearcă scrierea acestuia.
- EMFILE: S-a atins limita de descriptori de fişiere deschişi în cadrul procesului.
- ENFILE: S-a atins limita de descriptori de fişiere deschişi în cadrul sistemului.
- ENOENT: Fişierul specificat nu există şi nu se intenţionează crearea acestuia, sau un director din calea fişierului specificată nu există.
- EROFS: Fişierul specificat este read-only şi se încearcă scrierea acestuia.
- ETXTBSY: Fişierul specificat este un fişier executabil în execuţie. Nu este permisă modificarea executabilelor asociate unui program în execuţie.

read()

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

Argumentul *count* specifică numărul maxim de octeţi ce trebuie citiţi. Pointer-ul *buffer* trebuie să indice o zonă de memorie de cel puţin *count* octeţi. Un apel reuşit al *read()* întoarce numărul de octeţi citit, sau zero în cazul în care se ajunge la sfîrşitul fişierului.

Un apel *read* ar putea citi mai puţin decît numărul cerut de octeţi. Pentru un fişier obişnuit, un motiv posibil este apropierea de sfîrşitul acestuia.

Atunci cînd *read()* este aplicat pe alte tipuri de fişiere, există o serie de motive pentru care acesta să citească mai puţini octeţi. De exemplu, dacă se citeşte de la un terminal, citirea se opreşte în momentul în care se întîlneşte caracterul terminator de linie.

De asemenea, apelul *read()* nu inserează octetul nul necesar semnalării sfîrşitului de şir de caractere. Acest lucru se întîmplă întrucît acest apel de sistem este utilizat pentru a citi orice tip de fişier. În anumite cazuri, în respectivele fişiere putem regăsi text, sau date binare.

write()

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

Similar ca şi în cazul apelului *read()*, argumentele au aceeaşi semnificaţie şi octeţii scrişi pot fi mai puţini. Spre deosebire de acesta, în cazul apelului *write()* nu este garantat că datele au ajuns în memoria în care rezidă fişierul, întrucît sistemul de operare efectuează optimizări prin intermediul zonelor tampon.

close()

```
#include <unistd.h>

int close(fd);
```

Acest apel de sistem închide un descriptor de fişier. Atunci cînd un proces se termină, toţi descriptorii de fişier ai acestuia sunt automat închişi.

În general este recomandată închiderea descriptorilor de fişiere neutilizaţi, întrucît astfel se sporeşte lizibilitatea codului şi fiabilitatea în contextul modificărilor ulterioare. Mai mult decît atît, descriptorii de fişier reprezintă resurse finite, iar astfel ocuparea acestora ar putea conduce la cazul în care un proces să aibă nevoie de un nou descriptor de fişier, dar limita să fie atinsă, întrucît alte procese au deschis un număr semnificativ de descriptori de fişier.

lseek()

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

2.1.2 Operaţii în afara modelului universal I/O: ioctl

2.2 Windows

3 Laborator 3

4 Laborator 4

5 Laborator 5

6 Laborator 6

7 Laborator 7

8 Laborator 8

9 Laborator 9

10 Laborator 10

11 Laborator 11

12 Laborator 12