Recapitulare

Laboratorul constă în subiecte în stilul celor de la examen, însoţite de mici bucăţi de cod complete sau aproape complete pe care le puteţi rula pentru a vă convinge. Discuţia pe marginea subiectelor cu asistentul sau cu colegii este încurajată. Subiectele sunt suficient de multe încât nu pot fi acoperite într-un singur laborator.

Exerciţii

Utilizaţi arhiva recap.tar.gz aferentă laboratorului.

1. Demand-paging, Copy-on-Write. Intraţi în directorul `01-cow-dp`.
   - Rulaţi comanda:make && make run
   - **Explicaţi** apariţia evenimentelor de Demand-paging şi de Copy-on-Write.
   - Pentru testare, utilizaţi fişierele: `fault/fault` şi `fault2/fault`.
   - De ce la ultima pagină din buffer (din fişierul `fault/fault`) nu se mai face Demand-paging?
   - **Hints**:
     - paginile **gri** reprezintă pagini muliple nemapate reprezentate într-un interval
     - paginile **verzi** reprezintă pagini ce aparţin unui singur proces
     - paginile **roşii** sunt pagini partajate de cel puţin două procese
     - la **click** pe o pagină se centrează pagina respectivă şi toate paginile legate de ea
     - după ce a a apărut un eveniment de tipul Copy-on-Write **un** click duce la pagina la care este mapată acum, **următorul** click va duce la pagina veche la care era mapată
2. Subsistemul IO. Intraţi în directorul `02-io` şi inspectaţi fişierul `splice.c`.
   - Închipuiţi-vă scenariul în care programul rulează repede şi un alt scenariu în care rulează greu (din punctul de vedere al timpului).
   - **Hints**:
     - Ce operaţii se fac la trunchierea fişierului?
     - Care este rolul fişierului /proc/sys/vm/drop_caches
     - Ce fel de mecanism implementează apelul de sistem `splice`?
     - **Atenţie**: Asiguraţi-vă că sistemul dispune de suficientă memorie (RAM) liberă pentru a putea aduce tot fişierul în RAM (i.e. dimensiunea memorie liberă > dimensiunea fişierului).
     - Utilizaţi scriptul `run.sh`: sudo ./run.sh
     - **Ignoraţi mesajul**:Command terminated by signal 13
3. Threads counter (LD): Un proces crează N thread-uri şi un thread apelează fork(); câte threaduri, atât în procesul părinte cât şi în procesul copil, există imediat după acest moment? Studiaţi conţinutul directorului `03-threads-counter`.În ce director din `/proc/$PID/` se află informaţiile despre thread-urile procesului? De ce acestea au asignat un pid?
4. Function hijacking (LD): De ce în cazul folosirii LD_PRELOAD se pot intercepta apeluri de funcţii? Intraţi în directorul `04-functions-hijacking`, compilaţi (`make`) şi rulaţi (`./test` şi `LD_PRELOAD=./libnative.so ./test`. Cum explicaţi output-ul diferit?
5. Iniţializări (VD): Intraţi în directorul `05-init`. Compilaţi programul `initializari`. De ce durează mai mult prima iniţializare? De asemenea, încercaţi să explicaţi dimensiunea mare a executabilului.
6. Spaţiu de adresă (AF): Intraţi în directorul `06-addrspc` şi inspectaţi fişierul `addrspc.c`. Ce se întâmplă cu spaţiul de adresă al procesului în momentul schimbării de context?
   - **Hints**:
     - Threadurile rulează alternativ, scriind la stdout atunci când încep rularea
     - Monitorizaţi `/proc/$PID/maps`
7. Reentranţă / thread-safety (AF): Catalogaţi funcţiile din `07-reentr/reentr.c` ca fiind reentrante / thread-safe. Puteţi modifica programul încât să puneţi în evidenţă aceste aspecte.
8. Stack growth (VD): Scrieţi o secvenţă de cod care să decidă dacă stiva creşte în sus sau în jos. Un asemenea exemplu îl aveţi în directorul `08-stack-growth`.
9. Apeluri de sistem (DB): Câte apeluri de sistem se efectuează în urma rulării secvenţei de mai jos:
```
for (i = 0; i < 10; i++) {
        getpid();
        write(1, "A", 1);
}
```
   - Intraţi în directorul `09-syscall`.
   - Analizaţi conţinutul fişierului `pid_test.c`.
   - Urmăriţi apelurile de system efectuate rulând comanda `strace`. strace ./pid_test
   - **Suplimentar**
     - Analizaţi conţinutul fişierului `fork_test.c`.
     - Rulaţi programul având pentru `SYS_TYPE` pe rând valorile `SYS_NATIVE` şi `SYS_GLIBC`.
     - Explicaţi rezultatul obţinut. Citiţi secţiunea `NOTES` din `getpid(2)`.

10. Creare procese (DB): Desenaţi arborele de procese rezultat în urma execuţiei programului `fork.c` din directorul `10-fork`.
11. Descriptori de fisier (OB): Un proces dispune de o tabelă de descriptori de fişiere cu 1024 de intrări. În codul său, procesul deschide un număr mare de fişiere folosind open. Totuşi, al 964-lea apel open se întoarce cu eroare, iar errno are valoarea EMFILE (maximum number of files open). Care este o posibilă explicaţie? Vă puteţi gândi şi la alte explicaţii decât cea din exemplu?
12. File descriptor (BD): Completaţi zona punctată de mai jos cu cod Linux(posix) astfel incat sa conduca la afişarea mesajului "alfa" la ieşirea standard (standard output) şi mesajul "beta" la ieşirea de eroare standard (standard error), fara sa alteraţi simbolurile standard fputs (functie), stderr şi stdout (FILE *):
    - /* de completat */
      ...
      fputs("alfa", stderr);
      fputs("beta", stdout);
    - Intrati in directorul `12-file`, si rulati comanda ./file >out 2>err
13. Calloc/Malloc (IS): Intraţi in directorul `13-access`. Rulaţi cele 2 programe atât cu macro-ul TEST_ACCESS definit cât şi fără acesta. Măsuraţi timpii de rulare folosind *time* şi încercaţi să explicaţi diferenţa.
14. Apeluri de sistem (RD): În urma rulării secvenţei de mai jos: int page_size = getpagesize();

        for (i = 0; i < 1000; i++)
            ptr[i] = malloc(page_size); se obţine un număr redus de apeluri de sistem, de ordinul zecilor (apelul de sistem folosit de malloc este `brk`). Cum explicaţi?
    - Intraţi în directorul `14-malloc-syscalls`.
    - Urmăriţi conţinutul fişierului `malloc-syscalls.c`.
    - Compilaţi fişierul (folosiţi `make`).
    - Rulaţi comanda `strace -e brk ./malloc-syscall 2>&1 > /dev/null | wc -l`.
    - Rulaţi comanda `ltrace -e malloc ./malloc-syscall 2>&1 > /dev/null | wc -l`.
    - **Suplimentar**
        - Actualizaţi macro-ul `DO_FREE` la valoarea 1.
        - Câte apeluri `brk` au loc? Cum explicaţi? Ce rol are apelul de sistem `brk`?
15. Page fault-uri (RD): Câte page fault-uri se obţin în urma rulării secvenţei de mai jos? char *p;

```
int status;
size_t i;
int page_size = getpagesize();
char value;

p = mmap(NULL, NUM_PAGES * page_size, PROT_READ | PROT_WRITE,
        MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
if (p == MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
}

for (i = 0; i < NUM_PAGES; i++)
    p[i*page_size] = i;

switch (fork()) {
    case -1:    /* handle error */
        perror("fork");
        exit(EXIT_FAILURE);

    case 0:     /* child process */

        for (i = 0; i < NUM_PAGES / 2; i++)
            value = p[i*page_size];

        for (i = NUM_PAGES; i < NUM_PAGES; i++)
            p[i*page_size] = page_size-i;

        exit(EXIT_SUCCESS);
        break;

    default:
        break;
```

```
        }

        wait(&status);
        for (i = 0; i < NUM_PAGES; i++)
                p[i*page_size] = i;
```

- Intraţi în directorul `15-faults`.
- Urmăriţi conţinutul fişierului `fork-faults.c`.
- Compilaţi fişierul (folosiţi `make`).
- Instalaţi pachetul `sysstat`. Pachetul conţine utilitarul `pidstat` care permite monitorizarea page fault-urilor unui proces (prin intermediul argumentului `-r`).
- Rulaţi programul `fork-faults`. Folosiţi `ENTER` pentru a continua programul.
- Folosiţi comanda `pidstat -r -T ALL -p $PID` pentru a urmări page fault-urile. Rulaţi comanda pentru fiecare secvenţă de program.
  - `$PID` reprezintă pid-ul unui proces. Puteţi afla atât pid-ul procesului părinte cât şi pe cel al procesului copil cu o comandă de forma `ps -ef | grep fork-faults`.
- Urmăriţi evoluţia numărului de page fault-uri pentru cele două procese: părinte şi copil. Page fault-urile care apar în cazul unui copy-on-write în procesul copil vor fi vizibile ulterior şi în procesul părinte.