Listă proiecte sisteme de operare

1 Regulament

- Implementarea trebuie urcată pe moodle de fiecare student până la data de 17.01.2020
- Proiectul valorează maxim 20 de puncte
- Toți membrii echipei primesc același punctaj
- Fiecare membru trebuie să își prezinte contribuția (afectând nota finală corespunzător)

2 Proiecte

1. **Shell** (2–3 studenți) – Scrieți un shell care să implementeze elementele de bază găsite în orice linie de comandă (ex. istoric comenzi, pipe, expresii logice, suspendarea unui program).
2. **ListPIDs** (1–2 studenți) – Implementați o funcție sistem care să ofere date (PID, stare, statistici legate de execuție etc.) despre toți descendenții unui proces dat (în ordine DFS). Scrieți programe în userland care să afișeze arborii rezultați folosind noua funcție de sistem.
3. **EventSync** (1–2 studenți) – Scrieți un set de funcții de sistem care să permită sincronizarea mai multor procese legate de apariția unui evenmient. Procesele își opresc execuția până un terț semnalează apariția evenimentului dorit. Funcții necesare `eventopen`, `eventclose`, `eventwait`, `eventsignal`.
4. **MutexPolicy** (3–4 studenți) – Scrieți un daemon în userland care să decidă politica de acces la mutecșii creați cu ajutorul unui set nou de funcții de sistem implementat de dumneavoastră: `mtxopen`, `mtxclose`, `mtxlock`, `mtxunlock`, `mtxlist` și `mtxgrant`. Ultimele două funcții sunt folosite de daemon pentru a stabili care proces obține acces la mutex. Mutecșii sunt vizibili de orice proces din sistem, dar fiecare proces păstrează o listă de mutecși folosiți (precum descriptorii de fișiere).
5. **UserSched** (2–3 studenți) – Implementați un user weighted round-robin scheduler: algoritmul parcurge lista de utilizatori cu procese gata de execuție în ordine round-robin. O dată ales un utilizator, algoritmul alege un proces aparținând acestuia și îl lasă să se execute un timp finit. Utilizatorii au o pondere asociată care dictează timpul de execuție permis.

6. **SchedSim** (2–3 studenți) – Scrieți un simulator de procese care să fie folosit pentru a evalua performanța algoritmilor de scheduling (cu prioritate și în timp real) oferind utilizatorilor indicatori standard precum utilizarea procesorului, timpul de așteptare etc.

7. **Alloc** (2 studenți) – Creați o bibliotecă care să ofere operații de alocare, realocare și eliberare a memoriei, dedesubt având grijă să păstreze eficient blocurile de date. Funcțiile implementate trebuie să poată fii apelate concomitent de mai multe procese din sistem. Scrieți programe de test care să demonstreze punctual corectitudinea diferitelor aspecte de implementare.

8. **UserFS** (2–3 studenți) – Scrieți un pseduo-sistem de fișiere care să afișeze utilizatorii activi din sistem și procesele asociate. Când este montat, în rădăcină se vor găsi directoare corespunzătoare fiecărui utilizator activ. În fiecare director se va găsi un fișier `procs` ce conține lista aferentă de procese active.

9. **ProcFS** (2–3 studenți) – Scrieți un pseduo-sistem de fișiere care să urmeze structura arborescentă a proceselor unui sistem. Când este montat, directoarele și subdirectoarele corespund PID-urilor și sunt numite ca atare. Fiecare director conține un singur fișier `stats` cu date statistice legate de proces.

10. **ContainerFS** (3 studenți) – Scrieți un sistem de fișiere minimal în userland folosind FUSE. Implementarea poate urmării un arhivator clasic de fișiere precum zip sau tar. Când sistemul este montat, rădăcina va conține directoarele și fișierele aferente arhivei.

11. **LockCheck** (3 studenți) – Scrieți un algoritm care să primească un fișier în care sunt folosite diferite mecanisme de sincronizare (ex. mutex, semafor) și la ieșire să prezinte un raport în care să arate dacă și unde există posibilitatea apariției unui fenomen de *d*eadlock sau *s*tarvation

12. **Lockers** (2–3 studenți) – Implementați propria bibliotecă care să ofere obiecte de sincronizare: mutecși, semafoare, rwlocks fără a folosi pe cele existente în biblioteci precum *p*thread. Pentru implementare puteți folosi doar primitive atomice precum *c*ompare-and-swap.

13. **Monitor** (2–3 studenți) – Implementați un obiect de sincronizare tip monitor. Folosiți variabile condiționale pentru a permite mai multor procese să coexiste în interiorul monitorului la un moment dat.

14. **CopyService** (3–4 studenți) – Scrieți un daemon în userspace care să permită copierea asincronă de fișiere.

    Daemon-ul trebuie să fie însoțit de o librarie care să expună următoarele funcționalități: (1) creerea unui job nou de copiere, dintr-o sursa data catre destinatie data (ambele sunt locale, pe masina care ruleaza daemonul) (2) anularea unui job de copiere (3) suspendarea unui job de copiere (4) listarea progresului si a starii unui job de copiere (5) listarea tuturor joburilor de copiere existente.

    Daemon-ul de copiere va putea fi configurabil, prin intermediul unui fisier de configurare "config" localizat langa daemon. Acesta va specifica cate fire de executie maxime poate folosi daemon-ul si cate job-uri maxime poate primi.

    API-ul recomandat librăriei, pentru:

```
(1) Creearea unui job de copiere:
    copyjob_t copy_createjob(const char * src, const char * dst);

(2) Anularea unui job de copiere:
    int copy_cancel(copyjob_t job);

(3) Întreruperea unui job de copiereȘ
    in copy_pause(copyjob_t job);

(4) Obținerea de informații despre un job de copiere:
    int copy_stats(copyjob_t job, struct copyjob_stats * stats);
    int copy_progress(copyjob_t job);

(5) Enumerarea joburilor de copiere:
    int copy_listjobs(copyjob_stats ** statslist, unsigned int * jobscount);
    void copy_freestats(copyjob_stats * stats);
```

15. **Telemetry** (3–4 studenți) – Scrieți un daemon în userspace, însoțit de o librărie care să expună funcționalitatea acestuia, care sa implementeze distribuția de mesaje de lungime scurtă, organizate pe canale de distribuție, între mai mulți participanți.

    Canalele de distribuție vor fi organizate ierahic, de așa natură incât, un mesaj trimis pe un canal părinte, va fi recepționat de toți cei inregistrați la canalul respectiv împreună cu toți aceia inregistrati la canelele copil ale acestuia. Canalele părinte vor fi create automat în momentul in care daemonul semnaleaza înregistrarea a cel puțin unui participant la comunicarea pe un canal copil.

    Există două tipuri de participanți: cei care trimit mesajele, numiți "publishers" si cei care le receptionează, numiți "subscribers".

```
Exemplu:

Un set de canale posibile la un moment dat, ar putea fi:
A: /comm/messaging/channel_a
B: /comm/messaging/channel_b
C: /comm/messaging
D: /comm

Pentru acest exemplu, avem următoarele afirmații corecte:
1) Un mesaj trimis catre canalul (A) va fi receptionat doar de (A)
2) Un mesaj trimis catre (C) va fi receptionat de (A) si (B)
3) Un mesaj trimis catre (D) va fi receptionat de (A),(B),(C) si (D).
4) Când canalul (A) este creat, automat vor fi disponibile canalele părinte (C) si (D)

API-ul recomandat librăriei, pentru:

(1) Înregistrarea unui participant fie "subscriber", fie "publisher":
    #define TLM_PUBLISHER 0x1
```

```
#define TLM_SUBSCRIBER 0x2
#define TLM_BOTH 0x3

tlm_t tlm_open(
    int type, // publisher, subscriber or both
    const char * channel_name
);
```

(2) Înregistrarea unei funcții "callback" (pentru notificare în timp real):

```
int tlm_callback(
    tlm_t token,
    void (* message_callback)(
        tlm_t token,
        const char * message)
);
```

(3) Obținerea ultimului mesaj primit (pentru aplicații care fac "polling"):

```
const char * tlm_read(
    tlm_t token,
    unsigned int * message_id
    // opțional, repr. numărul de mesaje trimise pe acest canal
);
```

(4) Postarea unui mesaj scurt pe un canal anume:

```
int tlm_post(tlm_t token, const char * message);
```

(5) Deînregistrarea participanților:

```
void tlm_close(tlm_t token);
```

(6) Alte funcții ajutătoare:

```
int tlm_type(tlm_t token); // returns TLM_* flag
```

16. **Supervisor** (3–4 studenți) – Scrieți un daemon in userspace, însoțit de o librărie care să expună funcționalitatea sa, care să execute, administreze și monitorizeze rularea mai multor servicii. Daemon-ul joaca rolul supervizorului care execută și instrumentează serviciile create de el, în numele apelanților librăriei.

    Serviciile sunt reprezentate de o sumă de programe care raman rezistente in memorie pe o perioadă indefinită de timp și pot îndeplini diverse funcționalități.

    Sistemul trebuie să expună urmatoarele funcționalități:

    (1) să creeze și să închidă un canal de comunicare cu supervizorul:

    ```
    supervisor_t supervisor_init();
    int supervisor_close(supervisor_t);
    ```

    (2) să creeze un serviciu nou, cu argumente parametrizabile:

```
    #define SUPERVISOR_FLAGS_CREATESTOPPED 0x1
    #define SUPERVISOR_FLAGS_RESTARTTIMES(times) ((times & 0xF) << 16)

    service_t service_create(
        supervisor_t supervisor,
        const char * servicename,
        const char * program_path,
        const char ** argv,
        int argc,
        int flags
    );

    int service_close(service_t service);
```

(3) să monitorizeze starea unui serviciu existent (running, closed):

```
    service_t service_open(
        supervisor_t supervisor,
        const char * servicename
    );

    #define SUPERVISOR_STATUS_RUNNING 0x1
    #define SUPERVISOR_STATUS_PENDING 0x2
    #define SUPERVISOR_STATUS_STOPPED 0x4

    int service_status(service_t service);
```

(4) să suspende și/sau să oprească un serviciu:

```
    int service_suspend(service_t service);
    int service_resume(service_t service);
```

(5) să oprească și să șteargă un serviciu:

```
    int service_cancel(service_t service);
    int service_remove(service_t service);
```

(6) să aibe opțiunea de recreere automată dacă serviciul iese din memorie neprevăzut:

```
    // a se vedea SUPERVISOR_FLAGS_RESTARTTIMES
```

(7) să listeze toate procesele instrumentate de supervizor:

```
    int supervisor_list(
        supervisor_t supervisor,
        char *** service_names,
        unsigned int * count
    );

    int supervisor_freelist(
        char ** service_names,
        int count
    );
```

17. **DiskAnalyzer** (3–4 studenți) –

Creați un daemon care analizează spațiul utilizat pe un dispozitiv de stocare începând de la un cale dată, și construiți un program utilitar care permite folosirea acestei functionalități din linia de comandă.

Daemonul trebuie să analizeze spațiul ocupat recursiv, pentru fiecare director conținut, indiferent de adâncime.

Utilitarul la linia de comanda se va numi "da" și trebuie sa expună următoarele funcționalități:

(1) Crearea unui job de analiză, pornind de la un director părinte și o prioritate dată

(a) - prioritățile pot fi de la 1 la 3 și indică ordinea analizei în raport cu celelate joburi (1-low, 2-normal, 3-high)

(b) - un job de analiză pentru un director care este deja parte dintr-un job de analiză, nu trebuie să creeze task-uri suplimentare

(2) Anularea/ștergerea unui job de analiză

(3) Întreruperea și restartarea (pause/resume) unui job de analiză

(4) Interogarea stării unui job de analiză (preparing, in progress, done)

```
Usage: da [OPTION]... [DIR]...
Analyze the space occupied by the directory at [DIR]

   -a, --add           analyze a new directory path for disk usage
   -p, --priority   set priority for the new analysis (works only with -a argument)
   -S, --suspend <id>  suspend task with <id>
   -R, --resume <id>  resume task with <id>
   -r, --remove <id>  remove the analysis with the given <id>
   -i, --info <id>  print status about the analysis with <id> (pending, progress, d
   -l, --list   list all analysis tasks, with their ID and the corresponding root p
   -p, --print <id>  print analysis report for those tasks that are "done"
```

Exemplu de folosire:

```
$> da -a /home/user/my_repo -p 3
Created analysis task with ID '2' for '/home/user/my_repo' and priority 'high'.

$> da -l
ID  PRI Path  Done Status            Details
2 *** /home/user/my_repo  45% in progress       2306 files, 11 dirs

$> da -l
ID  PRI Path  Done Status            Details
2 *** /home/user/my_repo  75% in progress       3201 files, 15 dirs

$> da -p 2
```

```
Path  Usage  Size Amount
/home/user/my_repo 100%   100MB  ########################################
|
|-/repo1/ 31.3%  31.3MB #############
|-/repo1/binaries/ 15.3%  15.3MB ######
|-/repo1/src/ 5.7%   5.7MB ##
|-/repo1/releases/ 9.0%   9.0MB  ####
|
|-/repo2/ 52.5%  52.5MB ####################
|-/repo2/binaries/ 45.4%  45.4MB ##################
|-/repo2/src/ 5.4%   5.4MB ##
|-/repo2/releases/ 2.2%   2.2MB #
|
|-/repo3/ 17.2%  17.2MB ########
[...]

$> da -a /home/user/my_repo/repo2
Directory 'home/user/my_repo/repo2' is already included in analysis with ID '2'

$> da -r 2
Removed analysis task with ID '2', status 'done' for '/home/user/my_repo'

$> da -p 2
No existing analysis for task ID '2'
```

18. **Encriptor** (1 student) – Să se implementeze un encriptor/decriptor care primește un fișier de intrare cu diferite cuvinte. Programul mapează fișierul de intrare în memorie și pornește mai multe procese care vor aplica o permutare random pentru fiecare cuvânt. Permutările vor fi scrise într-un fișier de ieșire. Programul poate primi ca argument doar fișierul de intrare, în acest caz va face criptarea cuvintelor; sau va primi fișierul având cuvintele criptate și permutările folosite pentru criptare, caz în care va genera fișierul de output având cuvintele decriptate.

19. **SimulareCabinet** (1 student) – Să se implementeze un program care simulează activitatea dintr-un cabinet medical: vor fi create un număr dat de thread-uri pacienți care vor aștepta pentru eliberarea unor resurse reprezentând doctorii (pot fi niște structuri iar consultația să consiste blocarea acelei structuri și afișarea id-ului doctorului). Clienții vor ocupa resursa doctor pentru o perioadă random care să nu depășească o limită dată. Fiecare pacient va fi generat la un interval aleator pentru o perioadă data de timp. După consultație, pacientul își va afișa timpul de așteptare și timpul consultației.

20. **ContribOS** (1 student) – Orice contribuție tehnică adusă (acum sau în trecut) unui sistem de operare open-source.