Lucrare 1

- La începutul cursului 4:
  - 11.03.2019, seria CA
  - 13.03.2019, seria CB
  - 13.03.2019, seria CC

3CA, varianta 1

1. Explicaţi de ce apelul `printf(”aici”)` înainte de o instrucţiune care accesează o zonă de memorie invalidă nu afişează nimic la consolă.
   - **Răspuns**: Biblioteca va stoca local mesajul aici pentru a-l trimite la consolă atunci când bufferul se umple sau utilizatorul scrie (i.e. \n). Atunci când procesul este întrerupt în urma accesului invalid la memorie, bufferul nu mai este scris la consolă.
2. Explicați semnificația stării RUNNING în viața unui proces și tranzițiile posibile din această stare.
   - **Răspuns**: Procesul se află în execuție pe procesor. Dacă îi expiră cuanta va merge în starea READY; dacă face un apel blocant va tranziționa în starea WAITING. Dacă își încheie execuția va ajunge în starea TERMINATED.
3. Ce valori va afișa execuția codului următor atunci când fork reușește?

```
int i = 0;
if (fork()==0)
    i++;
else sleep(1);

printf(”%d\n”,i);
```

   - **Răspuns**: Va afișa 1 (scris de copil) urmat de 0 (scris de părinte).

3CA, varianta 2

1. Explicați de ce apelul `fwrite(..)` este mai rapid decât `write(..)` atunci când facem multe scrieri.
   - **Răspuns**: fwrite scrie într-un buffer în memoria procesului și execută write cu mai multe date, reducând numărul de apeluri de sistem. Apelurile de sistem sunt costisitoare, iar reducerea lor creste performanța.
2. Explicați semnificația stării READY în viața unui proces și tranzițiile către această stare.
   - **Răspuns**: Procesul este gata să ruleze însa așteaptă să fie planificat. Un proces ajunge în READY după ce este pornit, dacă este RUNNING si este preemptat, sau dacă este WAITING și operația pe care o asteptă se termină.
3. Câte procese noi vor fi create la execuția codului următor atunci când fork reușește?

```
int i = 0;
for (i = 0;i<5;i++)
  if (fork()>0)
     break;
```

- **Răspuns**: Vor fi create 5 procese noi; de fiecare dată părintele își încheie execuția și copilul va continua bucla pâna când i devine 5.

3CB, varianta 1

1. Numiți un rol al bibliotecilor în sistemul de operare și exemplificați.
    - **Răspuns**: Permit refolosirea codului de mai multe procese; o singură bibliotecă poate să fie mapată în memoria mai multor procese ocupând mai puțină memorie fizică. De asemenea, o bibliotecă poate fi o interfață între nucleu și procese.
2. Fie secvența `int fd2 = dup(fd)`. Execuția apelului de sistem `write` pe fd2 va influența în vreun fel apelurile ce folosesc ca parametru **fd**? Explicați.
    - **Răspuns**: Da, deoarece file offset-ul din structura alocată în kernel se va modifica prin apelul write pe fd2.
3. Care este diferența între un proces zombie și unul orfan?
    - **Răspuns**: Un proces orfan este un proces al cărui părinte a murit, iar unul zombie este un proces care a murit, dar la care părintele nu a făcut wait.

3CB, varianta 2

1. Care este motivul pentru care există noțiunea de "apel de sistem" într-un sistem de operare? De ce nu se fac pur și simplu apeluri simple de funcție?
    - **Răspuns**: Pentru izolarea proceselor, anumite funcționalități care pot influența alte procese sau integritatea sistemului au nevoie de un schimb de context către kernel unde se verifică dacă procesul are permisiunile necesare pentru efectuarea operației.
2. Fie secvența `int fd2 = dup(fd)`. Execuția apelului de sistem `write` pe fd va influența în vreun fel apelurile ce folosesc ca parametru **fd2**? Explicați.
    - **Răspuns**: Da, deoarece file offset-ul din structura alocată în kernel se va modifica prin apelul write pe fd.
3. Numiți un avantaj și un dezavantaj al structurării ierarhice al proceselor într-un sistem de operare (în comparație de exemplu cu o structură liniară). Explicați.
    - **Răspuns**: Reprezentarea proceselor într-o ierarhie ajută la partajarea de date între procesul părinte și procesul copil, însă implementarea este mai dificilă, deoarece trebuie păstrate mereu legături cu procesul părinte (ex. Procesele orfane sunt adoptate de init).

3CC, varianta 1

1. De ce preferăm să folosim un apel de bibliotecă care NU realizează în spate un apel de sistem în locul unui apel de bibliotecă ce face apel de sistem? De exemplu `fwrite()` în loc de `write()`.
    - **Răspuns**: Un apel de sistem are un overhead mai mare decât un apel simplu de funcție întrucât se face schimbarea nivelului de privilegiu la apel și revenirea din apel, care consumă timp. Din acest motiv, dacă avem de ales, optăm pentru funcții de bibliotecă ce nu fac apel de sistem ca să reducem overhead-ul.
2. Fie apelul `close(fd)` executat cu succes. Observăm că structura de fișier deschis referită de descriptorul fd (acum închis) nu este eliberată din memorie. De ce?

- **Răspuns**: pelul `close(fd)` invalidează descriptorul fd și decrementează contorul de referință din structura de fișier deschis. Structura este dezalocată în momentul în care contorul de referință este 0. Dacă după `close(fd)` structura de fișier deschis nu a fost dezalocată înseamnă că mai există un descriptor care o referă, descriptor care a fost obținut printr-un apel `dup()` sau `dup2()`. Un apel `open()` nu este un răspuns corect, pentru că acela creează o structură nouă de fișier deschis.

3. Un proces zombie devine și proces orfan: procesul său părinte își încheie execuția. Ce se întâmplă acum cu procesul zombie rămas orfan?
    - **Răspuns**: Un proces orfan, indiferent dacă este zombie sau nu este adoptat de init. Apoi init face "reaping" în procesele zombie, așteptându-le și eliminându-le astfel complet din sistemul de operare.

3CC, varianta 2

1. Dați exemplu de acțiune care poate fi executată doar în modul privilegiat (kernel mode). Justificați de ce acea acțiune nu poate fi executată în modul neprivilegiat (user mode).
    - **Răspuns**: Acțiuni care pot fi executate doar în kernel mode sunt: lucrul cu I/O, forme de IPC (Inter-Process Communication), alocarea memorie. Acestea sunt privilegiate pentru a garanta integritatea sistemului și izolarea proceselor. Dacă ar fi executate în mod neprivilegiat, un proces ar avea acces la datele altui proces, i-ar putea suprascrie și corupe informații. Cu atât mai mult cu cât dorim separație între procese privilegiate (ce aparțin utilizatorului root) și procese neprivilegiate.
2. Ce câmpuri din structura de fișier deschis și din structura de fișier de pe disc (FCB: File Control Block) modifică apelul `write()`? Argumentați răspunsul.
    - **Răspuns**: Apelul `write()` modifică pointer-ul/cursorul de fișier, îl crește cu numărul de octeți scriși în fișier; cursorul de fișier este reținut în structura de fișier deschis. În plus, dacă în urma scrierii octeților se trece de limita fișierului, atunci se modifică și dimensiunea fișierului; dimensiunea fișierului este reținută în structura de fișier de pe disc (FCB).
3. Ce efect are, la nivel de implementare, folosirea operatorului `&` în shell?
    - **Răspuns**: Atunci când folosim operatorul `&` în shell, shell-ul nu mai așteaptă încheierea procesului nou creat. Adică shell-ul nu apelează wait() și nu se blochează în așteptarea încheierii execuției procesului. Din acest motiv shell-ul poate primi noi comenzi de la terminal.

Greșeli frecvente

- Confuzie între procesele **zombie** și procesele **orfan**. Un proces zombie este un proces care și-a terminat execuția, dar pe care nu l-a "așteptat" părintele, iar un proces orfan este un proces al cărui părinte și-a încheiat execuția.
- Confuzie între kernel mode și utilizator privilegiat. Chiar dacă utilizatorul root este privilegiat, este doar un utilizator. În continuare se face diferențierea între user mode și kernel mode.
- Greșeli legate de utilizarea `&` în shell. În shell, `&` are rolul de a duce un proces în background, nu de a face o dereferențiere (ca de exemplu în C) sau de operator logic *AND* între comenzi (precum `&&`).
- Confuzie între procesele trimise în background și daemoni. Unui proces trimis în background **NU** i se închid `stdin`, `stdout` și `stderr`.

Lucrări foarte bune

- APOSTOL Teodor-Petruț, 336CA
- ŞENDRE Mihai-Alin, 332CC
- RADU Valentin-Gabriel, 333CC
- BĂLĂNICĂ Darius, 334CC
- COCOŞ Vlad, 334CC
- DUMITRU Philip, 334CC
- VIŞAN Anamaria, 334CC
- FOLEA Rareş, 336CC
- ILIE Vlad-Florin 331CB
- POPESCU Teodor-Constantin 333CB
- SMĂDU Răzvan-Alexandru 335CB
- FIRUŢI Bogdan-Cristian, 331CA
- IVAN Vlad, 331CA
- VASILE Cristian-Ştefan, 331CA
- ANDREI Rareş, 332CA
- DRAGOMIR Horia-Alexandru, 333CA
- POPA Bogdan, 333CA
- BUCUR Adrian-Cătălin, 336CA
- MARICA Andreea-Mădălina, 336CA

Lucrare 2

- La începutul cursului 7:
    - 01.04.2019, seria CA
    - 03.04.2019, seria CB
    - 03.04.2019, seria CC

3CA, varianta 1

1. Indicați trei surse de overhead care apar la schimbarea de context între două procese.
    - **Răspuns**: Context save/restore; TLB flushes; cache misses for app data, scheduler overheads.
2. Cum poate un proces să își modifice propria tabelă de pagini?
    - **Răspuns**: Doar invocând apeluri de sistem; procesul nu își poate scrie singur tabela de pagini pentru că ar permite breșe de securitate.
3. Explicați cum funcționează mecanismul copy-on-write pentru memoria virtuală.
    - **Răspuns**: Se mapează diferite secțiuni din executabil în memorie, inclusiv sectiunea text. Această mapare presupune adăugarea unor intrări în tabela de pagini, acestea sunt marcate ca fiind invalide, dar încărcabile de pe disk. Atunci când se accesează o adresă dintr-o astfel de pagină, MMU va genera un fault care va fi prins de SO; atunci SO va citi datele de pe disk într-un frame liber și va updata intrarea din tabela de pagini; instrucțiunea care a generat fault va fi repornită.

3CA, varianta 2

1. Enumerați pașii efectuați de SO pentru schimbarea de context între două procese.
    - **Răspuns**: Save context; flush TLB; change PTBR; select proces to run from READY queue; load context.
2. Descrieți pașii prin care o adresă virtuală este translată la o adresa fizică într-un sistem cu paginare.
    - **Răspuns**: MMU identifică pagina dorita prin inspectarea unui număr predefinit (e.g. 20) din biții cei mai semnificativi din adresa virtuală. Acest număr este căutat în TLB; dacă există o intrare, se înlocuiește pagina cu frame-ul asociat și se obține adresa fizică. Altfel MMU folosește PTBR pentru a localiza tabela de pagini activă și găsește frame-ul asociat la adresa PTBR + page * page_entry_size.
3. Explicați mecanismul prin care SO încarcă în memorie codul unui executabil exact atunci când acel cod urmează să fie folosit.
    - **Răspuns**: Mai multe pagini (din mai multe procese) vor fi mapate la același frame, însă intrările respective vor fi marcate R/O. Atunci când se accesează una din aceste pagini, MMU generează trap care este rezolvat de SO prin duplicarea frameului, maparea paginii faulty la frame-ul nou cu permisiuni R/W și reluarea instrucțiunii faulty.

3CB, varianta 1

1. Ce parametri ai planificatorului trebuie să modificăm și cum pentru a avea un sistem cu productivitate mai mare?
    - **Răspuns**: Pentru a avea un sistem cu productivitate mai mare putem mări cuanta de timp alocată unui proces din starea running. O cuantă de timp mai mare înseamnă un număr mai mic de context switch-uri, deci overhead mai mic, ceea ce duce la productivitate mai mare a sistemului per total.
2. Menționați un avantaj și un dezavantaj al paginării față de segmentare. Explicați avantajul, respectiv dezavantajul.
    - **Răspuns**: Un avantaj al paginării față de segmentare este acela că paginarea nu duce la fragmentare externă. Deoarece paginile au dimensiuni fixe, nu vor exista situații în care să avem goluri ce nu pot fi refolosite/sunt greu de refolosit între două pagini alocate, față de cazul segmentării.
      Un dezavantaj al paginării față de segmentare este acela că paginarea duce la fragmentare internă. Deoarece paginile au dimensiuni fixe, în momentul în care avem de alocat o secțiune cu dimensiune mică relativă la dimensiunea paginii, se va aloca o pagină întreagă, iar memoria rămasă liberă în cadrul paginii nu poate fi refolosită de alt proces.
3. La execuția unei instrucțiuni se generează un page-fault dar procesul care a rulat acea instrucțiune continuă să ruleze fără eroare. Motivați/descrieți de ce nu s-a încheiat execuția procesului.
    - **Răspuns**: Apariția unui page-fault nu generează neapărat un eveniment care să ducă la încheierea forțată a execuției programului. De exemplu, dacă instrucțiunea care a generat page fault-ul încercă să citească memorie dintr-o pagină care se află în swap, se generează un page-fault pentru aducerea paginii din swap. De asemenea, dacă instrucțiunea respectivă încearcă să scrie într-o pagină marcată ca fiind copy-on-write, se generează un page-fault pentru duplicarea paginii.

3CB, varianta 2

1. Ce parametri trebuie să modificăm și cum pentru a avea un sistem cât mai responsive?

- **Răspuns**: Pentru a avea un sistem cu responsivitate mai mare putem micșora cuanta de timp alocată unui proces din starea running. O cuantă de timp mai mică înseamnă un număr mai mare de context switch-uri, adică procesele ajung mai des și mai repede să ruleze pe procesor oferind un grad mai ridicat de responsivitate.

2. Ce avantaj aduce noțiunea de ierarhie în paginare?
   - **Răspuns**: Un avantaj al paginării ierarhice este reducerea spațiului ocupat de tabela/tabelele de pagini.
3. Fie următoarele 2 secvențe de cod:

```
a) a = mmap(...., n * 1024 * sizeof(int),....);
   for(i = 0; i < n * 1024; i = i + 1024)
      a[i] = i;
b) b = mmap(...., n * 1024 * sizeof(int),....);
   for(j = 0; j < n; j++)
      b[j] = j;
```

   Care dintre secvențe este mai rapidă și de ce?

   - **Răspuns**: Snippet-ul de cod de la punctul b) se va executa mai rapid decât snippet-ul de cod de la punctul a) deoarece vor fi mai puține page-fault-uri. Numarul de iterații este același în ambele cazuri (n), dar în cazul a) se indexează relativ la 1024, generând page-fault la fiecare instrucțiune.

3CC, varianta 1

1. De ce procesele I/O intensive primesc, în general, o cuantă de timp de rulare mai mare?
   - **Răspuns**: Procesele I/O intensive accesează des dispozitive I/O și execută des operații blocante. Aceasta înseamnă că un proces I/O intensive proaspăt planificat se va bloca repede și va trece din starea RUNNING în starea BLOCKED/WAITING. Șansele sunt foarte mici ca un astfel de proces să își încheie cuanta de timp în starea RUNNING. De aceea i se oferă o cuantă de timp mai mare, pe care o va folosi în mai multe runde de planificare, fiecare rundă de planificare încheindu-se, în general, cu tranziția sa în starea BLOCKED/WAITING din cauza unei operații blocante.
2. Un sistem pe 32 de biți are pagini de 4KB. Presupunând că sistemul folosește schemă de paginare simplă, pe un singur nivel (adică nu ierarhică/multi-nivel), care este dimensiunea aproximativă (ordinul de mărime) al unei tabele de pagini?
   - **Răspuns**: Pe un sistem pe 32 de biți, spațiul virtual de adrese are 2^32 = 4GB. Dacă o pagină are 4KB, avem 4GB / 4 KB = 2^20 pagini. O tabelă de pagini simplă, pe un singur nivel avea o intrare pentru fiecare pagină. Dacă o intrare în tabela de pagini ocupă 4-8 octeți (aproximativ) atunci o tabelă de pagini va ocupa 2^20 * (4/8) octeți, adică undeva la 4MB-8MB de memorie.
3. Fie instrucțiunea `*a = 3;` În ce situație executarea acestei instrucțiuni va conduce la evacuarea unei pagini în spațiul de swap (swap out)?
   - **Răspuns**: Instrucțiuna `*a = 3` înseamnă un acces la memorie în pagina virtuală ce conține adresa indicată de pointerul a. Dacă acea pagină virtuală nu are pagina fizică rezidentă (adică în memoria fizică / RAM), pagina fizică nu a fost alocată încă (demand paging) sau se găsește pe spațiul de swap sau a apărut duplicarea ei ca urmare a copy-on-write. Dacă nu aveam nici o pagină disponibilă, atunci este nevoie de evacuarea unei pagini (swap out) și folosirea paginii eliberate.

3CC, varianta 2

1. De ce este util ca prioritatea proceselor să fie dinamică? Care este neajunsul folosirii priorităților statice?
    - **Răspuns**: În cazul unui sistem cu priorități statice, un proces care este CPU-intensive și care are o prioritate mai bună va fi planificat foarte des: îi expiră cuanta, este trecut în coada READY cu prioritate foarte bună, apoi este replanificat. În această situație un proces care are o prioritate mai slabă va fi planficat foarte rar, posibil deloc, adică să ajungă la situația de starvation. Pentru a compensa aceste dezavantaj, folosim priorități dinamice, care permit proceselor cu prioritate mai slabă să câștige, în timp, priorități mai bune.
2. De ce, în general, TLB-ul (Translation Lookaside Buffers) este golit (flushed) la schimbări de context?
    - **Răspuns**: Intrările din TLB sunt cache-uri din tabela de pagini a procesului curent, intrări folosite recent. La o schimbare de context are loc schimbarea proceselor adică și a tabelelor de pagini. Fiind vorba de o tabelă de pagini nouă, intrările din TLB sunt nevalide și atunci sunt golite (flushed) pentru a fi populate cu intrări din tabela de pagini nouă.
3. Dați un exemplu de situație în care un page fault conduce la livrarea unei excepții de acces de memorie către un proces (de tipul *Segmentation fault*) și un exemplu de situație care nu conduce la livrarea unei excepții de acces de memorie.
    - **Răspuns**: Un page fault este generat de MMU când o pagină a fost marcată ca nevalidă în tabela de pagini. Dacă acea pagină nu a fost rezervată de proces înseamnă că nu este accesibilă din spațiul de adrese al procesului și atunci sistemul de operare va livra o excepție de acces la memorie (Segmentation fault) procesului. Altfel, dacă acea pagină este marcată în informațiile sistemului de operare ca fiind copy-on-write sau rezervată dar nealocată (demand paging) sau evacuată pe disc (swapped out), se va trata page fault-ul fără ca sistemul de operare să livreze excepție de acces la memorie procesului.

Greșeli frecvente

- Multe răspunsuri conform cărora unui proces I/O i se dă cuantă mai mare pentru că este mai lent și are nevoie de mai mult timp pentru a termina operația de I/O.
- Multe răspunsuri conțin doar calcule, fără explicații prin care să prezinte ce reprezintă numerele scrise. Multe răspunsuri în care nu se prezintă unitatea de măsură (ex. *dimensiunea tabelei este 4* sau *2^20*).
- Se rotunjesc puterile lui 2 la puterile lui 10 (ex. 1KB == 1000B).
- Multe răspunsuri conform cărora, prin folosirea priorităților statice, procesele care așteaptă date (ex. input de la utilizator) consumă timp inutil pe procesor.
- Multe răspunsuri care presupun că malloc nu alocă memorie, doar o rezervă (folosește mmap întotdeauna).

Lucrări foarte bune

- CEBERE Ioan-Tudor, 331CC
- ILIESCU Valentina-Florentina, 331CC
- ION Horia-Paul, 331CC
- TĂZLĂUANU Bianca, 334CC
- VIŞAN Anamaria, 334CC
- FOLEA Rareş, 336CC

- SULIMAN Anca, 336CC
- DANCIU Andra-Maria, 335CB
- MĂRGINEANU Cristian, 336CB
- ANDREI Rareș, 332CA
- MARICA Andreea-Mădălina, 336CA
- MARIN Cristian-Alexandru 333CB
- TRIFU Andrei-Ștefan 331CB
- SECUIU Ana 336CB

Lucrare 3

- La începutul cursului 10:

- 22.04.2019, seria CA
- 24.04.2019, seria CB
- 24.04.2019, seria CC

3CA, varianta 1

1. Un programator dorește să citească aleator dintr-un fișier de 1GB pe un sistem cu 2GB de memorie RAM. Ajutați-l să aleagă între memory mapped I/O și standard I/O.
    - **Răspuns**: Memory-mapped I/O este preferabil din cauza accesului aleator și pentru că există destul RAM pentru a ține fișierul în memorie; după încărcarea inițiala programul va merge la viteza memoriei.
2. Cum poate fi suprascrisă adresa de retur a unei funcții dintr-o aplicație de tip server folosind un stack buffer overflow atunci când stack canaries sunt folosite pe un sistem cu 64 de biți? Pentru fiecare client, serverul se va clona folosind fork(), și va apela o funcție de tratare a cererii.
    - **Răspuns**: Se poate ataca canarul octet cu octet atunci când există un server care folosește fork.
3. Scrieți pseudocod care implementează un semafor (up neblocant și down blocant atunci când `sem == 0`) folosind suportul hardware oferit de funcția compare_and_swap (cas): `int compare_and_swap(int* reg, int oldval, int newval);`
    - **Răspuns**:

```
int sem = 0;
void up(){
  while (1) {
    int o = sem;
    if (cas(&sem, o, o+1)) break;
  }
}

void down() {
  while (1) {
    int o;
    while ((o = sem) == 0);
    if (cas(&sem, o, o-1)) break;
```

```
    }
  }
```

3CA, varianta 2

1. Un programator dorește să citească secvențial un fisier de 10GB pe un sistem cu 1GB de memorie RAM. Ajutați-l să aleagă între memory mapped I/O și standard I/O.
    - **Răspuns**: Standard I/O este preferabil pentru că accesul este secvențial și caching-ul datelor în memorie nu este necesar; de asemenea memoria este de dimensiune mică.
2. Cum poate fi gasită o adresă validă pe un sistem pe 64 biți cu ASLR atunci când există un stack buffer overflow? Presupunem că programul atacat este un server. Pentru fiecare client, serverul se va clona folosind `fork()`, și va apela o funcție de tratare a cererii.
    - **Răspuns**: La un server care folosește fork se poate ataca ASLR încercând octet cu octet adresa de retur a funcției, până când se găsește o adresă validă în zona de text.
3. Scrieți pseudocod care implementează un spinlock (lock și unlock) folosind suportul hardware oferit de funcția compare_and_swap(cas):

```
int compare_and_swap(int* reg, int oldval, int newval);
```

- **Răspuns**:

```
void unlock(){
  lock = 0;
}

void lock() {
  while (!cas(&lock,0,1));
}
```

3CB, varianta 1

1. Ați executat comanda pmap pe un proces ce rulează și ați observat că secțiunea *.text* și secțiunea *.rodata* se află mapate în aceeași pagină cu atributele read și execute. De ce NU este o problemă de securitate faptul că secțiunea *.rodata* are aceste permisiuni?
    - **Răspuns**: Chiar dacă secțiunea .rodata este mapată în aceeași zonă ca secțiunea .text și ambele au permisiunile pentru citire și execuție, acest lucru nu este o problemă de securitate deoarece, în general, în secțiunea .rodata nu vom găsi cod executabil, și zona nefiind mapată cu drepturi de scriere, un posibil atacator nu ar putea adăuga cod executabil în acea secțiune.
2. Când preferăm folosirea proceselor în detrimentul thread-urilor într-un sistem multi procesor? Oferiți un exemplu.
    - **Răspuns**: Preferăm să folosim procese în detrimentul thread-urilor atunci când dorim să asigurăm robustețea programului (atunci când în cadrul unui thread se generează o excepție sau o eroare și se încheie execuția întregului program; în cazul proceselor, dacă un proces se termină brusc, nu afectează celelalte procese).De asemenea, preferăm să folosim procese în detrimentul thread-urilor și în cazurile în care dorim să întărim securitatea aplicației (în cazul în care un proces este compromis, nu se compromite întreaga aplicație). Un astfel de exemplu este un browser web. Dacă pentru

fiecare tab se crează un proces nou, în cazul în care unul dintre tab-uri crapă/are o eroare, nu se va închide întreaga aplicație, ci doar tab-ul respectiv.

3. Într-un sistem multi-procesor, mai multe thread-uri execută

```
int a=10, b = 10000; // variabile globale
thread_func() {
  a--;
  if ( a > 0) {
    b +=  b / a;
  }
}
```

La un moment dat, execuția programului este întreruptă de o eroare. Identificați/explicați problema și propuneți o soluție.

- **Răspuns**: Variabila a este accesată într-un mod care nu este thread-safe. Între intrarea în if și execuția împărțirii din cadrul threadului X, un alt thread Y decrementează a, și aceasta devine 0. Thread-ul X va face împărțire la 0 și primește semnalul SIGFPE care va încheia execuția programului. Trebuie adăugat un mutex la intrarea și ieșirea din funcție.

3CB, varianta 2

1. Un server este compilat cu flag-ul `-fstack-protector`. Implementarea se bazează pe un pool de procese care va deservi clienții. Workerii sunt creați printr-un apel `fork()`, urmat de un apel `execve()` ce va încărca un binar (compilat la fel ca serverul) care tratează cererile clienților. Când un worker moare, altul va fi creat în locul său. Funcția de prelucrare a cererilor conține următoarele 2 linii:

```
char a[10];
read(socket, a, 100);
```

Poate fi acesta exploatat printr-un atac de tipul buffer overflow? Motivați

- **Răspuns**: Workerii sunt creați prin fork() urmat de execve(). Prin urmare, valoarea de canary de pe stivă va fi aleatoare pentru fiecare worker. Prin urmare, NU putem ghici canary-ul octet cu octet, deci șansele sunt de 1 la 4 miliarde. Atacul nu este fezabil.

2. De ce este mai rapid un context switch între două thread-uri ale aceluiași proces față de context switch-ul între două procese diferite?
   - **Răspuns**: Față de procese, thread-urile împart același spațiul de adresă. Astfel, un context switch între două thread-uri este mai rapid decât un context switch între două procese deoarece în primul caz nu se face flush la TLB.
3. Descrieți o situație când `a = a + 5` nu este atomică.
   - **Răspuns**: Declarăm variabila a global, și o modificăm intermitent în alt thread. În cazul în care arhitectura e x86, sistemul trebuie să NU fie single-core.

3CC, varianta 1

1. De ce Address Space Sanitizer (ASAN) se folosește doar în faza de dezvoltare, nu și în producție?

- **Răspuns**: ASAN are un overhead foarte mare (200%). Așa ceva face nepractică folosirea ASAN în producție. Poate fi însă folosit în dezvoltare pentru a detecta atunci probleme în cod.

2. De ce este preferat ca la o schimbare de context de execuție să schimbăm un thread al unui proces cu un alt thread al aceluiași proces?
    - **Răspuns**: La schimbarea contextului de execuție între două thread-uri ale aceluiași proces nu schimbăm și spațiul de adresă. Întrucât nu îl schimbăm, nu este nevoie de schimbarea tabelei de pagini și de flush de TLB. Sistemul devine în felul acesta mai productiv.
3. De ce implementarea internă a unui **mutex** are nevoie de un **spinlock**?
    - **Răspuns**: În implementarea sa internă un mutex are o variabilă internă ce reține starea (locked / unlocked) și o coadă în care sunt menținute thread-urile care așteaptă eliberarea mutexului. Protejarea accesului la variabila internă și la coadă necesită o primitivă de sincronizare, adică un spinlock.

3CC, varianta 2

1. Ce tip de mecanism protejează împotriva atacurilor de tip *code injection*, dar nu și împotriva atacurilor de tip *code reuse*?
    - **Răspuns**: Data Execution Prevention (DEP) nu permite executarea zonelor care au fost scrise. Adică atunci când injectăm cod, nu-l putem executa. Adică DEP oferă protecție la atacurile de tip code injection. Atacurile de tip code reuse nu sunt afectate pentru că aceste atacuri se referă doar la zona read-executable, nu la zone writable.
2. De ce thread-urile de tip user level nu pot fi folosite pentru paralelizarea unui program?
    - **Răspuns**: Implementarea de thread-uri în user space nu este vizibilă la nivelul sistemului de operare (kernel-ului). Întrucât planificatorul rulează în kernel mode, acesta nu poate planifica un thread per procesor pentru paralelizare. Toate thread-urile de tip user level ajung să ruleze pe un singur procesor.
3. De ce nu este suficientă folosirea instrucțiunilor hardware de tipul *compare-and-swap* (precum `cmpxchg` pe x86) pentru implementarea unui spinlock pe sistemele multicore?
    - **Răspuns**: Operațiile de tip compare-and-swap sunt atomice la nivelul procesorului curent. Întrucât magistrala este partajată, aceste instrucțiuni se pot intercala și duce la date necoerente. De aceea, în implementarea de spinlock pentru sistemele multicore se folosește lock pe magistrală.

Greșeli frecvente

- Multe răspunsuri care spun că ASAN este vulnerabil/introduce vulnerabilități.
- Multe răspunsuri care compară spinlock cu mutex și când e bine să folosim spinlock-ul (secțiune critică mică) și când e bine să folosim mutex-ul (secțiune critică mare), în loc să spună efectiv de ce este necesar să fie folosit un spinlock în implementarea unui mutex.
- Confuzie între kernel-level threads și user-level threads.
- Confuzii legate de ce face `execve()`.

Lucrări foarte bune

- LUCHIAN Alina-Elena, 331CC

- NIŢU Ioan-Florin-Cătălin, 333CC
- MOGA Mihaela-Mădălina, 334CC
- ŞTEFAN Laurenţiu, 334CC
- SULIMAN Anca, 336CC
- MANOLE Aida-Ştefania, 331CB
- DANCIU Andra-Maria, 335CB
- MĂRGINEANU Cristian, 336CB
- HINŢIU Diana-Florina, 341C5
- CREȚU Diana, 336CA
- OLARU Paul-Stelian, 332CA
- POPA Adrian, 332CA
- MITOCARU Irina, 334CB
- ALEXANDRU Ioana, 334CB
- TIŢA Matei-Dimitrie, 332CB
- BRÎNZOI Iuliana, 336CB
- NACA Andrei, 335CB
- LIURCĂ Daniel, 335CB

Lucrare 4

- La începutul cursului 13:
    - 20.05.2019, seria CA
    - 22.05.2019, seria CB
    - 22.05.2019, seria CC

3CA, varianta 1

1. Ce mecanisme putem folosi pentru a reduce ciclii folosiți de procesor pentru I/O pentru dispozitivele care sunt accesate rar?
    - **Răspuns**: Întreruperi și DMA.
2. Cum balansează plăcile de rețea moderne întreruperile către mai multe procesoare?
    - **Răspuns**: Au mai multe perechi de ring-uri; fiecare ring RX poate genera întreruperi și trimite către oricare procesor. Pachetele sunt demultiplexate în aceste ring-uri folosind o funcție de hash sau alte filtre specificate de utilizator.
3. De ce au i-node-urile dimensiune fixă?
    - **Răspuns**: Pentru a le putea stoca într-un vector a cărui adresă de început este cunoscută și pentru a le accesa printr-o *singură citire* atunci când este nevoie.

3CA, varianta 2

1. Dați un exemplu când polling este preferat intreruperilor pentru sincronizarea cu dispozitivele de I/O.
    - **Răspuns**: Plăci de rețea de viteză mare (10Gbps sau mai mult), atunci când primesc multe pachete mici (e.g. TCP ACKs la 10Gbps = 14.88Mpps).

2. Care sunt cele două mecanisme care permit stivei de TCP să obțină performanță mai bună decât stiva UDP atunci cand bottleneck-ul este sistemul gazdă, nu rețeaua?
    - **Răspuns**: Batching la apelul funcției send (se trimit mai multe date per syscall) și batching în stivă atunci când se lucrează cu segmente mai mari decat MTU; acestea sunt sparte înainte de a fi puse pe fir folosind GSO sau TSO.
3. Un utilizator a creat 10 linkuri hard și 10 symlink-uri către un fișier existent care avea inițial 1 singur hardlink. Câte inode-uri sunt folosite în total? Care este reference count-ul inode-ului fișierului `a.txt` după această operație?
    - **Răspuns**: 11 inode-uri - 1 pentru fisier si 10 pentru symlink. Ref count 11 (10 hard links noi + 1 existent).

3CB, varianta 1

1. Ce informații oferă intrările din fișierul `/proc/interrupts`?
    - **Răspuns**: Avem mai multe coloane: prima coloană reprezintă numărul întreruperii, ce are corespondent un pin pe procesor; N coloane, una pentru fiecare procesor, ce indică câte întreruperi au fost declanșate; tipul de întrerupere; numele modulului/modulelor de kernel ce îl gestionează.
2. Sunt suficiente buffer-ele prezente pe placa de rețea (NIC) pentru a realiza o transmisie/recepție a datelor? Explicați.
    - **Răspuns**: Pentru a realiza o transmie/recepție avem nevoie și de memoria principală. Ring bufferele placii vor referi memoria principală pentru ca datele să ajungă în aceasta.
3. La **nivel de implementare**, sub ce formă diferă linkurile simbolice și hard în cadrul sistemului de fișiere?
    - **Răspuns**: Un link simbolic este un fișier în cadrul sistemului de fișiere, ce conține o cale (un șir de caractere), care poate fi folosită pentru a traversa ierarhia de fișiere, și a ajunge la destinație. Un link hard este reprezentat de un dentry în cadrul unui director (o listă de structuri dentry), care conține un număr inode care corespunde inode-ului fișierului destinație.

3CB, varianta 2

1. Se dă un sistem cu 4 core-uri (procesoare). Dorim să verificăm cum sunt distribuite întreruperile pe fiecare din core-uri pentru dispozitivul de rețea. Cum am putea proceda?
    - **Răspuns**: Consultăm fișierul `/proc/interrupts`. Ne uităm pe ultima coloană (coloana cu modulele de kernel atașate), și căutăm linia unde apare modulul care corespunde plăcii noastre de rețea (e.g. ath9k (WI-FI atheros) sau `en*`(ethernet)). Pe coloanele CPU0-CPU3 găsim numărul de întreruperi declanșate pe fiecare nucleu.
2. Ce tipuri de buffere există în NIC-uri și care este rolul lor?
    - **Răspuns**: Într-un NIC în general există 2 buffere circulare: de recepție și de transmisie. În cazul unui spike în activitatea pe rețea, aceste 2 buffere vor acționa ca un tampon în care se pot acumula date: pentru recepție, procesorul va fi întrerupt mai puțin, iar pentru transmisie, kernelul va termina mai rapid și predictibil transmisia pachetelor. Rezultatul este că se reduce utilizarea procesorului în I/O.
3. Numiți 2 mecanisme de gestionare a spațiului liber care pot fi folosite în implementarea unui sistem de fișiere și menționați un avantaj pentru fiecare din ele.

- **Răspuns**: Alocare contiguă și alocare cu liste. La alocarea contiguă nu e nevoie să facem management/tracking la blocurile unui fișier, iar la alocarea cu liste avem flexibilitate în modificarea dimensiunii unui fișier (adăugare de noi blocuri.

3CC, varianta 1

1. Numiți un avantaj al folosirii memory-mapped I/O față de port-mapped I/O.
    - **Răspuns**: Memory-mapped I/O folosește un spațiu comun de adrese cu al memoriei. Aceasta înseamnă că folosește instrucțiunile de lucru cu memoria (de tipul mov, load, store) în loc de instrucțiuni dedicate precum in sau out. Rezultă un set de instrucțiuni (ISA) mai simplu al procesorului.
2. După apelul `connect()` pe un socket TCP de tip listener se creează un nou socket TCP de tip conexiune care este folosit pentru comunicare. Ambii socketi (și alții socketi creați cu apelul `connect()`) au aceeași adresă IP și port. Cum demultipleaxează sistemul pachetele pentru a le livra socket-ului corespunzător?
    - **Răspuns**: Sistemul de operare păstrează un 5-tuplu pentru fiecare conexiune cuprinzând adresa IP sursă, portul sursă, adresa IP destinație, portul destinație, protocolul de nivel transport. Atunci când vine un pachet nou cu aceeași adresă destinație și port destinație, pe baza **adresei sursă și a portului sursă** are loc demultiplexarea și identificarea socketului care va prelua pachetul.
3. Care este diferența dintre un fișier și un director în implementarea sistemului de fișiere?
    - **Răspuns**: Blocurile de date ale unui fișier conțin date de orice fel. Blocurile de date ale unui director conțin un vector de dentry-uri (intrări de director).

3CC, varianta 2

1. De ce spunem că, de obicei, operațiile de citire/scriere de pe/pe disc NU sunt blocante?
    - **Răspuns**: Operațiile de citire și de scriere pe disc folosesc, de obicei, cache-ul subsistemului de block I/O. Atunci când are loc o scriere, se realizează apel de sistem, se transferă datele din user space în kernel space și apoi apelul de sistem se întoarce (fără să se blocheze) urmând ca datele să fie scrise (flushed) la un moment ulterior. La fel, la citire, datele sunt de obicei prezente în cache (din prefetch sau read-ahead) și sunt transferate de acolo în user space.
2. De ce este utilă folosirea TSO (TCP Segmentation Offload) în plăcile de rețea moderne?
    - **Răspuns**: TSO (TCP Segmentation Offload) permite transmiterea unui pachet de nivel transport (segment) cu dimensiune mai mare decât MTU (Maximul Transmission Unit) al plăcii de rețea. Devine responsabilitatea plăcii de rețea (NIC) segmentarea pachetelor în frame-uri de dimensiunea MTU, eliberând procesorul de această prelucrare și crescând performața.
3. Sistemul de fișiere ext4 are dimensiunea maximă a unui fișier de 16TB. Ce cauzează această limitare?
    - **Răspuns**: Limitarea este cauzată de numărul de pointeri pe care îi are inode-ul către blocuri sau blocuri cu indirectare simplă sau dublă sau triplă. Fiecare pointer poate referi un bloc, numărul total de pointeri ducând la limitarea numărului total de blocuri accesibile și, astfel, la limitarea dimensiunii maxime a fișierului.

Greșeli frecvente

- Confuzie între memory-mapped I/O și maparea fișierelor în memorie.
- Confuzie între apelurile de sistem și instrucțiuni speciale precum in sau out.

- Foarte multe răspunsuri care afirmă că operațiile de read și write nu sunt blocante deoarece se folosește DMA.
- Foarte multe răspunsuri care afirmă că operațiile blocante blochează procesorul.
- Confuzie între operații non-blocante și asincrone.
- Confuzie între indexare și indirectare.
- Foarte multe răspunsuri incomplete și nejustificate.

2018:

Lucrare 1

- La începutul cursului 4:
  - 12 martie 2018, seria CA
  - 14 martie 2018, seria CB
  - 14 martie 2018, seria CC

3CA, varianta 1

1. Explicați diferența dintre descriptorii rezultați din două apeluri de open către acelasi fisier, și un apel de open urmat de un apel dup pe descriptorul rezultat.
   - **Răspuns**: Descriptorii rezultați din două apeluri de open către același fișier vor indica spre două structuri de fișier deschis diferite, pe când în cazul unui apel `open()` urmat de un apel `dup()` descriptorii vor indica spre aceeași structură.
2. Ce va afișa la rulare următorul cod, presupunând că fork este executat cu succes?

```
int i = fork();
if (i==0)
  i = getppid();
else
  sleep(1);

printf("%d\n", i);
```

   - **Răspuns**: La rularea codului se va afișa:
     <pid-ul procesului părinte>
     <pid-ul procesului copil>

     Fork returnează 0 în cadrul procesului copil și pid-ul procesului copil în cadrul procesului părinte. Astfel, procesul copil va afla pid-ul procesului părinte și își va continua execuția prin printarea acestuia în timp ce procesul parinte face sleep de 1 secundă.
3. Scrieți secvența de cod care implementează comanda ls > /dev/null.
   - **Răspuns**:

```
pid_t pid = fork();
```

```
switch(pid) {
case 0:
  fd = open("/dev/null", O_WRONLY | O_TRUNC | O_CREAT, 0644);
  dup2(fd, STDOUT_FINENO);
  close(fd);
  execlp("ls", "ls", NULL);
default:
  waitpid(pid, NULL, 0);
}
```

De asemenea, au fost punctate și variantele care conțin variante în pseudocod ale apelurilor precum: `open("/dev/null"),exec("ls")` etc.

3CA, varianta 2

1. Dați un exemplu de funcție I/O care permite transmiterea de date către disk, rețea și dispozitive de tip caracter.
   - **Răspuns**: O funcție de I/O care permite transmiterea de date către disk, rețea și dispozitive de tip caracter este apelul de sistem write().
2. Ce va afișa la rulare următorul cod, presupunând că fork este executat cu succes?

```
int i = fork();

if (i==0)
  i++;
else
  sleep(1);

printf("%d\n", i);
```

   - **Răspuns**: La rularea codului se va afișa:
     1
     <pid-ul procesului copil>

     Fork returnează 0 în cadrul procesului copil și pid-ul procesului copil în cadrul procesului părinte. Astfel, procesul copil va afla va incrementa variabila i, și o va afișa. Părintele va face sleep de 1 secundă și va afișa pid-ul procesului copil.
3. Scrieți secvența de cod care lansează în execuție `ls | wc -l`.
   - **Răspuns**: O variantă de răspuns posibilă:

```
fd_redirect;

fd = fork();

if (fd == 0) { //child
  close(STDOUT);
  dup2(STDOUT, fd_redirect);
```

```
  exec("ls");
} else if (fd > 0) { //parent
  close(STDIN);
  dup2(STDIN, fd_redirect);
  wait(); //for child
  exec("wc -l");
} else {
  //error
}
```

3CB, varianta 1

1. Când este necesară execuția unui apel de sistem?
    - **Răspuns**: Execuția unui apel de sistem este necesară în momentul în care se interacționează cu hardware-ul.
2. Cum ați implementa redirectarea pentru următoarea comandă bash `ls >> test.out` (este de interes implementarea redirectării, în locul comenzii ls putea fi orice alta comandă)?
    - **Răspuns**: Variantă de răspuns:

```
int fd = open("test.out");
close (STDOUT_FILENO);
dup(fd);
close(fd);
lseek(STDOUT_FILENO,0, SEEK_END);
```

3. Câte procese se pot afla în starea RUNNING la un moment dat?
    - **Răspuns**: În starea RUNNING, se pot afla la un moment dat un număr de procese egal cu numărul de core-uri ale sistemului de calcul.

3CB, varianta 2

1. Indicați cel puțin un rol al nucleului (kernel-ului) într-un sistem de calcul.
    - **Răspuns**: Variante de răspuns: Nucleul are rolul de a abstractiza hardware-ului SAU rol de interfațare a hardware-ului SAU Nucleul oferă controlul accesului la resurse.
2. Ce apeluri de sistem sunt necesare pentru a porni două procese care comunică prin intermediul unui pipe (ex `ls | grep test`)?
    - **Răspuns**: O variantă de răspuns: Apelurile de sistem necesare pentru a porni două procese care comunică prin intermediul unui pipe sunt `fork()` pentru crearea celor două procese, `pipe()` pentru deschiderea unor pipe-urilor folosite de cele două procese, `close()` pentru închiderea unuia din capetele pipe-urilor create înainte, `dup2()` pentru redirectarea capetelor pipe-ului și `execve()` pentru execuție.
3. Care este cauza principală pentru care un proces este în starea WAITING?
    - **Răspuns**: În general un proces ajunge în starea de WAITING în urma efectuării unei operații blocante.

3CC, varianta 1

1. Stiva de rețea a unui sistem de operare poate fi implementată în user space sau în kernel space. De ce se preferă implementarea în kernel space?
    - **Răspuns**: Preferăm implementarea stivei de rețea în kernel space din rațiuni de performanță. Prelucrarea pachetelor se întâmplă în kernel space iar tranziția (costisitoare) către și din user space are loc doar pentru payload-ul pachetelor.
2. Dați un exemplu de apel de bibliotecă/sistem, altul decât `open()`/`fopen()` care duce la ocuparea unei intrări din tabela de descriptori de fișier a unui proces.
    - **Răspuns**: Popularea unei intrări în tabela de file descriptori se poate face cu apelul `dup()` (sau `dup2()`) pentru duplicarea unei intrări existente sau cu apeluri precum `socket()` sau `pipe()` pentru crearea unui socket sau a unui pipe care ocupă, respectiv, o intrare și două intrări.
3. Un proces execută apelul `getppid()` în două situații diferite. În prima situație apelul întoarce valoarea 2832, iar în a doua situație apelul întoarce valoarea 1. Cum explicați?
    - **Răspuns**: `getppid()` întoarce pid-ul procesului părinte. În prima situație valoarea returnată este diferită de 1 deoarece procesul are un părinte (creator) care încă rulează. În cea de-a doua situație procesul a fost adoptat de init deoarece părintele creator al procesului și-a terminat execuția și nu l-a așteptat.

3CC, varianta 2

1. Oferiți un motiv pentru care este necesară existența modului privilegiat (supervisor/kernel mode) într-un sistem de calcul.
    - **Răspuns**: Este nevoie de modul privilegiat pentru a asigura integritatea sistemului, izolarea între procese și accesul mediat la resurse. În absența modului privilegiat (kernel/supervisor mode), un proces ar putea scrie în memoria altui proces, ar putea corupe resurse sau ar putea citi informații de la resursele I/O fără să știe dacă sunt ale sale sau alte altcuiva.
2. Care este un avantaj și un dezavantaj al folosirii apelului `printf()` în locul apelului `write()` pentru afișarea unui mesaj la ieșirea standard a unui proces?
    - **Răspuns**: Avantajele folosirii `printf()` sunt: formatare, portabilitate, mai puține apeluri de sistem (când mesajul este buffered). Dezavantaje: buffer-ul ocupă memorie, nu se afișează instant mesajul (este buffered).
3. Unui proces îi expiră cuanta de rulare (time slice) în momentul în care rulează pe un procesor (se află în starea RUNNING). Ce se întâmplă cu procesul?
    - **Răspuns**: La expirarea cuantei de timp, un proces este mutat din starea RUNNING în starea READY, unde va rămâne până va fi din nou planificat: adică până atunci când va primi o altă cuantă de rulare și va fi lăsat să ruleze pe procesor.

Greșeli frecvente

Lucrări foarte bune

Lucrare 2

- La începutul cursului 7:

- 2 aprilie 2018, seria CA
- 4 aprilie 2018, seria CB
- 4 aprilie 2018, seria CC

3CA, varianta 1

1. Presupunând un sistem cu timp de access la RAM de 100ns, și știind că un context conține zece valori ale regiștrilor, estimați timpul necesar pentru o schimbare de context.
   - **Răspuns**: Worst case: 10 read * 100ns, 10 write * 100ns = 2 microsecunde. Dacă sunt în cache, timpul poate fi mai mic.
2. Un proces execută următorul cod:

```
for (i=0; i<100000; i++)
  if (fork() == 0)
    exit(1);
```

   Care este consumul total de memorie fizică folosită de toate procesele copil în plus față de procesul părinte, exceptând structurile de date din kernel?

   - **Răspuns**: Variantă răspuns: Fiecare proces copil va aloca o pagină pentru stivă prin mecanismul copy-on-write (care va fi scrisă atunci când funcția exit își inițializează stack-frame-ul, salvând base pointer). Consumul total va fi 400MB.
     Variantă de răspuns acceptată: nu se va aloca nici o pagină în copil din cauza copy-on-write, pentru că nu se modifică nici o variabilă.
3. Dați un exemplu de algoritm de înlocuire de pagini pentru care creșterea memoriei fizice nu garantează reducerea numărului de major page faults.
   - **Răspuns**: FIFO.

3CA, varianta 2

1. Într-un sistem preemptiv cu planificator round-robin, explicați dacă este posibil ca două procese ce sunt planificate alternativ pe același procesor să primească timp de procesor diferit.
   - **Răspuns**: Procesele sunt I/O bound.
2. Dați un exemplu de situație în care unei pagini din procesul părinte nu i se aplică mecanismul copy-on-write după apelul `fork()`, deși pagina respectivă este scrisă atât de copil cât și de părinte.
   - **Răspuns**: Pagina a fost mapată cu `mmap (…, MAP_SHARED,…);`
3. Dați avantaj și un dezavantaj al memory-mapped IO comparativ cu standard I/O.
   - **Răspuns**: Memory mapped I/O este mult mai rapid în general pentru că nu implică syscalls la fiecare acces; deasemenea mecanismul de paging nu încarcă în memorie decât paginile efectiv accesate. Orice eroare de I/O, de exemplu disk removed, este transmisă procesului prin semnale de genul SIGSEGV sau SIGBUF - iar programele trebuie să prindă și să trateze aceste semnale explicit. În cazul standard IO tratarea astfel de erori e mult mai simplă (-1, look at errno).

3CB, varianta 1

1. Într-un sistem doar cu procese I/O intensive, cuanta de rulare pe procesor ar trebui să fie mai mică sau mai mare decât în mod obișnuit pentru a eficientiza per ansamblu sistemul? Explicați.
    - **Răspuns**: Cuanta de rulare trebuie să fie mai mică întrucât subsistemele I/O sunt mult mai încete ca procesorul iar o cuantă de rulare mai mare nu ar face decât să blocheze procesorul fără ca acesta să execute nimic util.
2. Câte operații sunt necesare într-o arhitectură load/store pentru a copia o valoare între 2 locații de memorie RAM?
    - **Răspuns**: 2 operații (încărcare din memorie în registru și salvare din registru în memorie).
3. Alocăm un spațiu contiguu de 128MB folosind apelul mmap. Presupunând că dimensiunea paginii este de 4K, care este numărul maxim de page-fault-uri ce vor putea fi generate?
    - **Răspuns**: 128MB / 4k = 32K page fault-uri.

3CB, varianta 2

1. Într-un sistem doar cu procese CPU intensive, cuanta de rulare pe procesor ar trebui să fie mai mică sau mai mare decât în mod obișnuit pentru a eficientiza per ansamblu sistemul? Explicați.
    - **Răspuns**: Cuanta de rulare trebuie să fie mai mare întrucât la un context switch se pierde o perioadă importantă de timp, iar dacă cuanta ar fi prea mică, mai mult am sta în schimbarea de context decât să procesăm volumul mare de date pe procesor.
2. Precizați un avantaj și un dezavantaj al paginării.
    - **Răspuns**: Avantaje: memoria ocupată per proces, un proces are acces la tot spațiul virtual de memorie propriu, etc; Dezavantaje: mai multe accese la memoria fizică, prezența hardware-ului specializat (MMU, TLB), etc
3. Precizați 2 cazuri care generează un page fault atunci când accesăm o adresă validă de memorie.
    - **Răspuns**: Swapping, Demand paging.

3CC, varianta 1

1. De ce este important ca prioritățile proceselor să fie dinamice, nu statice, în contextul planificării proceselor?
    - **Răspuns**: Dacă prioritățile sunt statice, atunci procesele prioritare vor rula întotdeauna în fața celor mai puțin prioritare. Dacă sunt CPU bound, atunci acestea vor ocupa foarte mult procesorul și procesele mai puțin prioritare vor avea puțin timp să ruleze, ducând la starvation.
2. La ce este folosit PTBR (Page Table Base Register)? De ce sistemul nu ar putea funcționa fără un PTBR?
    - **Răspuns**: PTBR este folosit pentru a referi adresa fizică a tabelei de pagini a procesului curent. În abența PTBR-ului, nu am avea un mod prin care să identificăm tabele de pagini a unui proces și deci nici un mod prin care să putem face translatarea de adrese când folosim memorie virtuală.
3. Apelul de mai jos este efectuat cu succes. În ce situație duce apelul la două page fault-uri?

```
memset(a, 0, 8); /* fill a with 8 bytes of 0 */
```

    - **Răspuns**: Scrisul de 8 octeți poate genera două page fault-uri atunci când sunt “atinse” două pagini de memorie care fie sunt în demand paging fie swappate. Dacă avem, de exemplu, 4 octeți pe o pagină (la sfârșit) și 4 octeți în pagina următoare (la început) vor rezulta două page fault-uri.

3CC, varianta 2

1. De ce procesele I/O bound primesc, în general, o cuantă de timp de rulare mai mare?
    - **Răspuns**: Procesele I/O bound sunt procese care se blochează des (și generează schimbări de context voluntare). Le alocăm o cuantă de timp mai mare pentru că nu vor apuca să o consume, se vor bloca. La următoarea planificare va consuma din restul de cuantă. Un proces CPU bound primește o cuantă mai mică pentru a fi forțat să iasă de pe proces (schimbare de context nevoluntară) atunci când îi expiră cuanta.
2. De ce este utilă prezența unei zone dedicate pentru kernel în spațiul virtual de adrese al fiecărui proces, față de un spațiu virtual de adrese dedicat pentru kernel?
    - **Răspuns**: La un apel de sistem se schimbă nivelul de privilegiu al procesorului dar nu se schimbă tabela de pagini, pentru că nu se schimbă spațiul de adresă. În felul acesta nu mai facem flush la TLB și avem overhead mai redus. În cazul în care kernel-ul ar avea un spațiu virtual de adrese propriu ar trebui făcută schimbarea tabelei de pagini (și flush la TLB).
3. De ce zona de text/cod a bibliotecii standard C poate fi partajată între mai multe procese, dar nu și zona de date?
    - **Răspuns**: Zona text este o zona read-only, pe când zona de date e modificabilă. Întrucât modificările trebuie să fie vizibile doar la nivelul procesului curent, zona de date e unică per proces (și nepartajate). Zona text nu se modifică și poate fi partajată de mai multe procese.

Greșeli frecvente

Lucrări foarte bune

Lucrare 3

- La începutul cursului 11:

- 7 mai 2018, seria CA
- 9 mai 2018, seria CB
- 9 mai 2018, seria CC

3CA, varianta 1

1. Implementați o incrementare atomică a variabilei t folosind funcția atomică GCC `_sync_bool_compare_and_swap (int* t, int r, int n)`.
    - **Răspuns**:

```
do {
        int s = t+1;
} while (!_sync_bool_compare_and_swap(&t,s-1,s));
```

2. Explicați cum poate un thread din procesul P1 citi memoria procesului P2.
    - **Răspuns**: Shared memory sau meltdown pe un system fara KPTI enabled.
3. Un atacator dorește să atace un sistem cu stack canary protection pe 8 octeți. Serverul care este atacat face `fork()` pentru fiecare client și are un stack overflow exploatabil de client prin modificarea inputului. De câte încercări are nevoie atacatorul, în medie, pentru a ghici valoarea canary?

- **Răspuns**: Atacatorul poate exploata faptul ca valoarea canary ramane aceeasi intre doua conexiuni ale sale (din cauza fork). El poate ataca fiecare octet din canar independent, incepand cu primul; daca greseste o sa i se inchida conexiunea; daca nu, o sa mearga. Astfel, in medie va avea 128 de incercari (i.e. conexiuni) per octet. Dupa ce gaseste primul octet il ataca pe al doilea. In total va avea nevoie de 8 * 128 = 1024 incercari in medie.

3CA, varianta 2

1. Implementați un spinlock folosind funcția atomică GCC `_sync_bool_compare_and_swap (int* t, int r, int n)`.
    - **Răspuns**:

      ```
      while (!_sync_bool_compare_and_swap(&lock,0,1));
      ```

2. Explicați de ce este mai ieftină crearea unui thread comparativ cu crearea unui proces nou.
    - **Răspuns**: Nu este necesara duplicarea tabelei de pagini, nici a descriptorilor.
3. De câte încercări are un atacator nevoie (în medie) pentru a găsi o adresă de cod validă pe un sistem de 64 biti atunci când atacă un sistem cu ASLR pornit. Acest sistem rulează un server care face fork() pentru fiecare client și care are un stack overflow exploatabil de client prin modificarea inputului.
    - **Răspuns**: De 1024 de incercari in medie: ataca octet cu octet adresa de retur salvata pe stiva, observand daca i se inchide conexiunea (a gresit) sau nu.

3CB, varianta 1

1. Care e motivul pentru care secțiunile `.data` și `.bss` sunt separat puse în fișierul executabil?
    - **Răspuns**: `.bss` refera variabila globale care au valoarea 0. Gruparea acestora intr-o sectiune speciala, diferita de `.data`, optimizeaza consumul de spatiu intrucat nu mai este necesar sa stocam 0-uri.
2. Se crează 2 thread-uri în cadrul aceluiași proces. Se poate ca unul din thread-uri să modifice fluxul executiei celuilalt thread?
    - **Răspuns**: Da, prin coruperea stivei celuilalt thread.
3. Secventa de cod `a += 5` este atomică? (la executarea secvenței de 2 thread-uri, rezultatul final va fi același tot timpul) Explicați.
    - **Răspuns**: Da, daca avem un singur core. Nu, daca avem mai multe core-uri.

3CB, varianta 2

1. Care este motivul pentru care secțiunea de cod este stocată separat de restul datelor într-o zonă denumită `.text`?
    - **Răspuns**: Zona .text are o proprietate speciala: trebuie sa fie executata si nu trebuie sa poata fi scrisa (pentru a nu modifica ilegitim codul de executat). Astfel maparea acelei zone este diferita fata de celelalte zone din executabil.
2. Cum putem preveni un atac de tip stack buffer overflow?
    - **Răspuns**: Toate functiile ce acceseaza la scriere variabilele de pe stiva, sa fie limitate printr-un parametru sau printr-o verificare implicita la lungimea buffer-ului alocat.

3. De ce nu este atomică instrucțiunea `add [%rax], 5` cand avem 2 core-uri în sistem?
    - **Răspuns**: add [%rax], 5 - Codul de asamblare presupune un acces la memorie pentru a citi valoarea initiala. Acest acces la memorie nu este atomic (ambele core-uri pot acces memoria in acelasi timp). Avem nevoie de primitiva lock pentru asigurarea accesului exclusiv la magistrala.

3CC, varianta 1

1. De ce trecerea peste limita unui buffer (buffer overflow) nu generează, de obicei, excepție de acces la memorie (de tipul Segmentation fault)?
    - **Răspuns**: Întrucât sistemul de operare gestionează memoria la nivel de pagini. Un buffer este o parte a unei pagini. Dacă trecerea peste limita unui buffer (buffer overflow) nu duce la trecerea într-o altă pagină (nevalidă), atunci nu rezultă în excepție de acces la memorie, accesul fiind valid din punctul de vedere al sistemului de operare.
2. De ce schimbarea de context între două thread-uri user-level este mai rapidă decât schimbarea de context între două thread-uri kernel-level ale aceluiași proces?
    - **Răspuns**: Thread-urile cu implementare kernel-level fac schimbarea de context la nivelul nucleului sistemului de operare, lucru care presupune schimbarea nivelului de privilegiu din user space în kernel space și invocarea planificatorului. Thread-urile cu implementare user-level fac schimbarea de context în user space, fără apel de sistem sau tranziție în kernel space (cauzatoare de overhead) și este așadar mai rapidă.
3. De ce instrucțiunea `a++` din C poate fi atomică pe un sistem x86 single-core dar niciodată pe un sistem x86 multi-core?
    - **Răspuns**: Instrucțiunea a++ se poate traduce în add [mem_address], 1. O astfel de instrucțiune este atomică pe un singur core (nu poate fi întreruptă în mijlocul execuției). Însă, pe un sistem multi-core, instrucțiunea se “desface” în acțiuni de load, add, store și lucru cu magistrala de acces la memorie, magistrală partajată. Pe un astfel de sistem multi core, pot exista întrețeseri de acces la magistrală din parte mai multor core-uri care fac instrucțiunea neatomică și duc la rezultate nedeterministe.

3CC, varianta 2

1. Pentru a putea folosi un shellcode pe un sistem care are suport DEP (Data Execution Prevention) un atacator folosește apelul `mprotect()`. De ce și cum folosește atacatorul apelul `mprotect()`?
    - **Răspuns**: Un sistem cu suport DEP va marca zonele writable din spațiul virtual de adrese al unui proces (data, bss, stack, heap) ca neexecutabile. Un shellcode este injectat într-o zonă writable. Pentru a putea fi executat trebuie ca acea zonă să fie executabilă. Pentru aceasta folosim apelul mprotect() care adaugă permisiuni de execuție (PROT_EXEC) acelei zone în care am injectat shellcode-ul.
2. De ce schimbarea de context între două thread-uri ale aceluiași proces e mai rapidă decât schimbarea de context între două procese?
    - **Răspuns**: Schimbarea de context între două thread-uri ale aceluiași proces nu schimbă spațiul virtual de adrese, deci nu schimbă tabela de pagini, care duce la anularea intrărilor din TLB. Schimbarea de context între două thread-uri din procese diferite duce la schimbarea spațiului virtual de adrese, lucru care înseamnă schimbarea tabelei de pagini și are overhead suplimentar.
3. Cum arată în pseudo-asamblare implementarea primitivei `lock()` pentru un spinlock?

- **Răspuns**: Un spinlock este o variabilă care este actualizată atomic. Operația lock() pe spinlock este un busy waiting cât timp acea variabilă este ocupată, adică cineva a achiziționat deja spinlock-ul. Această verificare trebuie să fie atomică, lucru realizat cu operații atomice de tipul CAS (compare-and-exchange) și lock pe magistrală. Dacă valoarea 1 înseamnă spinlock liber, iar 0 spinlock ocupat, în pseudo-asamblare implementarea operației lock() va fi:

```
spinlock = 1; /* initial value is 1 - open */
lockfn:
        lock cmpxchg spinlock, 1, 0
        cmp old_value, 0 ; if old_value (stored in a register by cmpxchg)
is 0, busy wait
        je lockfn
```

Greșeli frecvente

- **Buffer Overflow (scrierea peste dimensiunea buffer-ului) este considerat a fi același lucru cu atacul de tip Stack Buffer Overflow.**
- **Buffer Overflow (trecerea peste dimesiunea unui buffer) se întâmplă doar pe stivă**. Un buffer overflow poate să apară și pe heap.
- **Codul este pus într-o zonă diferită față de celelalte date pentru a evita suprascrierea de cod**. Nu se explică faptul că diferă permisiunile, presupun că fac un buffer overflow până în cod.
- **Pentru a preveni atacuri de tip stack buffer overflow se folosesc ASLR și DEP**. ASLR și DEP sunt folosite pentru a îngreuna anumite atacuri, nu pentru a preveni buffer overflow-ul în sine. DEP previne execuția unui shellcode introdus de atacator deoarece zona respectivă nu are concomitent permisiuni de write și execute. Singura legătură cu buffer overflow-ul e că suprascriind adresa de retur a funcției să pointeze către shellcode, acesta nu va putea fi executat. ASLR pune adresele funcțiilor din libc random pentru a îngreuna apelul unor funcții de bibliotecă precum system. Însă, folosind return-to-libc (prin exploatarea unui buffer overflow), se pot leak-ui adresele funcțiilor din libc și se poate calcula offset-ul funcțiilor dorite, făcându-se bypass ASLR-ului. Astfel, ASLR nu previne buffer overflow-ul.

Lucrări foarte bune

Lucrare 4

- La începutul cursului 13:
    - 21 mai 2018, seria CA
    - 23 mai 2018, seria CB
    - 23 mai 2018, seria CC

3CA, varianta 1

1. Explicați pașii necesari pentru a transfera un bloc de la un HDD către aplicația care citește un

fișier.

- **Răspuns**: Pas 1. Driver-ul SO programează controller-ul de hard-disc să citească blocul.
  Pas 2. HDD-ul citește blocul, plasând conținutul într-un buffer intern al HDD;
  Pas 3. controller inițiază DMA pentru a transfera datele în memorie;
  Pas 4. când DMA se finalizează, se generează o întrerupere pentru a anunța procesorul că datele sunt disponibile.

2. La momentul t0 apelul send întoarce valoarea 100. Presupunând că apelurile pe socketi sunt blocante, este posibil ca apelul `recv` efectuat la celălalt capăt al conexiunii la momentul t1 > t0 să întoarcă valoarea -1?
   - **Răspuns**: Da, dacă rețeaua sau oricare dintre mașini se deconectează; succesul apelului send indică doar copierea datelor în buffer-ul nucleului transmițătorului.
3. Dorim să folosim un sistem de

fișiere FAT pentru un disc de 10TB și dimensiunea blocului de 4KB. Cât de mare va

fi tabela FAT pentru acest disc?

- **Răspuns**: Numărul total de blocuri (și de intrări în tabela FAT): 10 * 10^12 / 4 / 10^3 = 2.5*10^9. E nevoie de 4 octeți per bloc pentru a encoda FAT-ul; deci vor fi necesari în total 4 * 2.5 * 10^9B RAM= 10GB.

3CA, varianta 2

1. Dați un exemplu în care polling este preferabil în defavoarea întreruperilor pentru lucrul cu dispozitive I/O.
   - **Răspuns**: La plăcile de rețea cu viteză mare (10Gbps sau mai mult), când vin pachete mici (e.g. 64B) se vor genera milioane de întreruperi pe secundă, și fiecare întrerupere este costisitoare (salvare context, etc). Dacă folosim polling, evităm acest cost și performanța este mai bună.
2. Atunci când se transmit 10.000 octeți folosind un socket TCP, de câte apeluri `recv(fd, buf, 10000, 0)` va

nevoie pentru a primi toate datele?

- **Răspuns**: Minim 1 apel, maxim 10.000 de apeluri - teoretic SO poate întoarce orice număr de octeți per apel citire, chiar și 1 octet.

3. Pe un HDD cu capacitate de 1TB dorim să instalăm un sistem de

fișiere ext2 cu dimensiunea blocului de 4KB. Câte nivele de indirectare sunt necesare pentru a permite unui

fișier să ocupe (aproape) tot spațiul de pe HDD?

- **Răspuns**: Număr blocuri fișier maxim: 1TB/4KB = 10^12 / 4 / 10^3 = 250*10^6 blocuri. Dacă folosim 1 nivel indirectare ⇒fișierul maxim poate avea 1000 blocuri; 2 nivele indirectare ⇒1.000.000 de blocuri; 3 nivel ⇒10^9. Sunt necesare 3 nivele de indirectare.

3CB, varianta 1

1. În directorul curent există fișierul in.dat. Executăm comanda `/usr/bin/time -v cp in.dat out.dat`. Aceasta durează 1 secundă. Mai executăm o dată comanda `/usr/bin/time -v cp in.dat out.dat`. A doua oară aceasta va dura doar 0.1 secunde. Care ar putea fi cauza?
   - **Răspuns**: Page cache / buffer cache (caching-ul fișierului înainte de a fi scris pe disc)
2. Executăm linia următoare `int n = recv(s,buf,1000,0);`. Dați exemplu de un caz în care n va fi 1 și de un caz în care n va fi 1000.
   - **Răspuns**: n va fi 1 atunci când buffer-ul de receive are un singur octet și va fi 1000 când va avea cel puțin 1000 de octeți.
3. Creăm un hardlink la un fișier. Dacă ștergem acel fișier, link-ul devine invalid? Explicați mecanismul din spate.
   - **Răspuns**: Nu devine invalid. Hardlink-ul este un dentry ce pointează către același inode al fișierului. O dată șters fișierul, inode-ul nu se va șterge întrucât există o referință la acesta (hard-link-ul)

3CB, varianta 2

1. Dorim să modificăm setările unui driver în Linux. Ce apel de sistem folosim pentru acest lucru?
   - **Răspuns**: ioctl
2. Executăm linia următoare `int n = send(s,buf,2000,0);`. Dați exemplu de un caz în care n va fi 1 și de un caz în care n va fi 2000.
   - **Răspuns**: n va fi 1 atunci când buffer-ul de send este aproape plin (mai are un singur octet liber) și va fi 2000 atunci când buffer-ul este gol și are cel putin 2000 de octeți liberi.
3. Gestiunea spațiului liber într-un sistem de fișiere se face folosind o structura de tipul vector de biți sau o listă înlănțuită. Precizați câte un dezavantaj al fiecărei metode.
   - **Răspuns**: Dezavantaj vector de biți: parcurgere liniară. Dezavantaj lista înlănțuită: overhead de memorie.

3CC, varianta 1

1. Un program rulează prima dată într-un sistem, apoi este oprit. Apoi este pornit din nou. Se observă că timpul de pornire a doua oară este mult mai mic. Care este explicația?
   - **Răspuns**: Când pornește prima oară, secțiunile din fișierul executabil al programului/procesului sunt aduse prima oară în RAM de pe disc (suportul persistent), acțiune care durează. După ce procesul este oprit, mare parte din secțiuni rămân în buffer cache, astfel că a doua oară nu mai este nevoie de aducerea lor de pe disc, iar pornirea e mai rapidă.
2. În ce situație se blochează un apel `send()` pe un socket?
   - **Răspuns**: Un apel send() pe un socket se blochează în momentul în care buffer-ul de send/transmit al socket-ului este plin și nu mai există nici măcar un octet liber unde să fie transferate date din buffer-ul din user space.
3. Ce conțin blocurile de date ale unui director?

- **Răspuns**: Blocurile de date ale unui director conțin un vector de dentry-uri; fiecare dentry conține un nume și un număr de inode aferent acelui nume. Aceste dentry-uri sunt intrările din directorul respectiv, vizibile în Unix prin rularea comenzii ls.

3CC, varianta 2

1. Care este o diferență între o operație non-blocantă și o operație asincronă?
    - **Răspuns**: Operațiile non-blocante sunt operații care nu blochează execuția procesului, întorcându-se imediat; operațiile non-blocante fie întorc datele disponibile atunci (sincron), fie declanșează în spate execuția unei operații (asincron). Operațiile asincrone sunt operații care se execută separat de fluxul de execuție al procesului (asincron); rezultatul acestor operației este disponibil la încheierea operației prin notificarea procesului sau dacă procesul interoghează starea operației. În general operațiile asincrone sunt non-blocante.
2. În ce situație se blochează un apel `recv()` pe un socket?
    - **Răspuns**: Un apel recv() pe un socket se blochează în momentul în care buffer-ul de receive al socket-ului este gol și nu mai există nici măcar un octet care să fie transferat în buffer-ul din user space.
3. Cu ce diferă un hard link de un symbolic link?
    - **Răspuns**: Un hard link este un nume/dentry al unui fișier (inode). Crearea unui hard link înseamnă crearea unui dentry nou. Un symbolic link este un inode care conține calea către un alt inode. Crearea unui symbolic link înseamnă crearea unui inode. Astfel că un symbolic link poate avea unul sau mai multe hard link-uri.

Greșeli frecvente

- **Funcția recv() se blochează atunci când buffer-ul e plin**.
- **Operatiile non-blocante sunt doar cele de genul pe fișiere cu flag-ul O_NONBLOCK - dau datele imediat**. Practic a fost făcută o comparație între `O_NONBLOCK` si `ASYNC IO`.
- **Operațiile non-blocante nu notifică procesul, iar operațiile asincrone notifică procesul.**

2017:

3CA, varianta 1

1. Ce se întâmplă cu shell-ul după executarea comenzii `exec sleep 10`? Dar după executarea comenzii `sleep 10`?
    - **Răspuns**: La execuția comenzii `exec sleep 10`, imaginea shell-ului este înlocuită cu imaginea executabilului `sleep`. Adică shell-ul nu mai există ca proces. La încheierea procesului `sleep`, acesta își încheie execuția, iar shell-ul nu mai există. În cazul execuției comenzii `sleep 10` se creează un nou proces care folosește imaginea executabilului `sleep`. Procesul shell așteaptă încheierea procesului `sleep` și apoi își continuă execuția.
2. Dați două exemple de evenimente care pot determina un proces să iasă din starea `RUNNING`.
    - **Răspuns**: Un proces poate ieși din starea `RUNNING` în momentul în care:
        - își încheie execuția;
        - realizează o operație blocantă și trece în starea `WAITING`;
        - îi expiră cuanta de timp alocată pentru rulare și trece în starea `READY`;
        - dă voie de bună voie la procesor (yield) și trece în starea `READY`;
        - există un proces prioritar care este planificat pe procesor și care forțează procesul curent să treacă în starea `READY`.
3. Cu ce diferă sistemele Windows de cele Unix la crearea de procese?
    - **Răspuns**: Pe sistemele UNIX, un proces este creat prin apelul `fork()` care creează o clonă / un duplicat al procesului părinte. Pentru un proces cu imagine de executabil nouă se folosește apelul `exec()` după apelul `fork()`. În Windows procesele sunt create cu apelul `CreateProcess` care primește imaginea de executabil, ca o unificare a apelurilor `fork()` și `exec()` din UNIX.

3CA, varianta 2

1. Explicați cum lansează shell-ul în execuție comanda `ls > a.txt`.
    - **Răspuns**: Pentru lansarea comenzii `ls > a.txt` shell-ul apelează `fork()` și creează o clonă a sa. În clonă folosește `open()` și `dup()/dup2()` pentru a înlocui ieșirea standard cu fișierul `a.txt`. Ulterior apelează o funcție din familia `exec()` pentru a înlocui imaginea de executabil a clonei cu executabilul `/bin/ls` aferent comenzii `ls`.
2. Care este diferența dintre `fork` și `fork + exec`?
    - **Răspuns**: În cazul apelului `fork()` se creează un proces clonă, copie aproape identică a procesului părinte. Procesul copil și procesul părinte execută același cod. În urma `fork() + exec()` imaginea (codul executabil) aferentă procesului copil este înlocuită cu imaginea de executabil transmisă ca argument funcției `exec()` rezultând într-un cod diferit pentru procesul copil.
3. Care este diferența dintre funcțiile `fwrite` și `write` atunci când scriem într-un fișier?
    - **Răspuns**: Funcția `write()` este un apel de sistem unbuffered, în vreme ce funcția `fwrite()` este un apel de bibliotecă buffered. Atunci când apelăm `write()` se execută apel de sistem care va transfera datele din user space în kernel space. În cazul apelului `fwrite()` datele sunt transferate într-un buffer intern al bibliotecii standard C urmând ca apelul de sistem efectiv (și flush-ul datelor) să aibă loc la un moment ulterior (de exemplu la newline sau când se umple buffer-ul intern). Apelul

`fwrite()` consumă mai multă memorie pentru bufer-ul intern cu avantajul reducerii overhead-ului cauzat de apeluri de sistem.

3CB/CC, varianta 1

1. Dați un exemplu de efect negativ care s-ar întâmpla dacă nu ar exista separația kernel mode / user mode.
    - **Răspuns**: Dacă nu ar exista kernel mode, o aplicație (un proces) ar avea acces la memoria altui proces și ar putea corupe buna funcționare a acestuia. De asemenea, un proces are acces nemediat la resursele hardware putând citi datele altor procese sau putând face un atac de tipul denial-of-service pe acele resurse.
2. Dați exemplu de operație care modifică cursorul unui fișier fără a modifica dimensiunea fișierului.
    - **Răspuns**: Operația `read()` modifică cursorul de fișier. Dimensiunea nu este modificată pentru că nu ajunge să se scrie mai mult sau mai puțin. Operația `write()` modifică cursorul de fișier și nu modifică dimensiunea dacă nu scrie peste dimensiunea curentă a fișierului. Operația `lseek()` este definiția enunțului: doar modifică cursorul de fișier, fără alte acțiuni.
3. Câte procese se pot găsi în starea `RUNNING` la un moment dat în sistemul de operare?
    - **Răspuns**: În starea `RUNNING` să găsește maxim un singur proces per procesor. Într-un sistem de operare avem maxim N procese în starea `RUNNING`, unde N este numărul de procesoare.

3CC, varianta 2

1. Dați exemplu de situație în care o aplicație oarecare are nevoie să execute un apel de sistem.
    - **Răspuns**: O aplicație are nevoie să execute un apel de sistem în momentul în care dorește să interacționeze cu I/O: rețea, hard disc, terminal, tastatură. De asemenea, atunci când dorește să comunice cu alte procese (în orice formă posibilă). Nu în ultimul rând, atunci când face operații de lucru cu memoria.
2. Ce conține o intrare din tabela de descriptori a unui proces?
    - **Răspuns**: Tabela de descriptori a unui proces conține pointeri către structurile de fișier deschis. Când un fișier este deschis se creează o structură de fișier deschis, iar în tabela de descriptori, în primul slot liber, se plasează un pointer la această structură.
3. Dați două exemple de situații în care are loc o schimbare de context între două procese.
    - **Răspuns**: Schimbarea de context între două procese poate avea loc atunci când:
        - procesului care rulează îi expiră cuanta;
        - procesul care rulează efectuează o operație blocantă;
        - procesul care rulează cedează de bună voie procesorul (yield);
        - procesul care vrea să ruleze are o prioritate mai mare;
        - procesul care rulează își încheie execuția.

Greșeli frecvente

- Procese in starea RUNNING: cate PID-uri avem in sistem.
- Tabela de descriptori conține PID-uri (proces curent, copii, părinte, etc.).
- In starea RUNNING se poate afla un singur proces.
- Separatia kernel mode/user mode: teorie generala, fara a da exemplu propriu-zis.

- Context switch - se înțelege prin context de fapt environment/context general (se modifică o variabilă de mediu, se modifică memorie partajată, 2 procese deschid același fișier și își mută independent cursoarele).
- Context switch (perlă frecventă): atunci când 2 procese accesează aceeași zonă de memorie.
- Este frecventă confuzia dintre PCB și intrarea din tabela de descriptori.

Lucrări foarte bune

- RADU Toma, 333CC
- BELINGHIR Grigore Petrut, 332CC
- MARDALE Andrei, 334CB
- DORNEANU Anda-Alexandra, 333CC
- MARINICĂ Elena, 335CA
- POPA Ștefan-Adrian, 332CA
- ILIE Andra Teodora, 332CA
- PICIU Laurențiu-Gheorghe, 333CA
- VATAMANU Alexandra, 331CA
- STOICAN THEODOR, 333CA
- CURCUDEL Ioan-Răzvan, 335CA
- BUȘILĂ Ștefan, 334CA
- DURBALĂ Cătălina, 332CA
- DIACONU Cristian, 333CA
- STAN Cristiana-Ștefania, 335CC
- MUNTEAN Dan Iulian, 335CB
- UNGUREANU Remus-Dan, 335CB
- PÎRTOACĂ George Sebastian, 335CB
- PANDELEA Alexandru, 335CB
- MACARIE Roxana, 333CB
- STERPU Paul, 334CB
- JITEA Ștefan, 335CC
- BULGARU Mihaela, 332CB
- OLTEANU Robert, 331CC
- POPA Crina Elena, 333CB
- CALOIANU George, 332CC

Lucrare 2

- La începutul cursului 7:
    - luni, 3 aprilie 2017, 17:00-17:15, EC105, seria CA
    - miercuri, 5 aprilie 2017, 14:00-14:15, PR001, seriile CB/CC

3CA, varianta 1

1. Explicați în ce situație două procese pot fi planificate inechitabil pe un procesor (adică ocupă timp diferit pe procesor) de către un planificator round-robin.

- **Răspuns**: Planificatorul round-robin planifică procesele rând pe rând. Dacă un proces este CPU intensiv iar altul I/O intensiv, atunci cel CPU intensiv va consuma mai mult timp pe procesor. Fie până când îi expiră cuanta (round-robin preemptiv) fie până când își încheie execuția sau, în final, se blochează (round-robin cooperativ). Procesul I/O intensiv se va bloca mult mai repede și astfel va ocupa mai puțin timp pe procesor.

2. Într-un sistem cu paginare, cum este transformată o adresă virtuală într-o adresă fizică?
    - **Răspuns**: Din adresa virtuală se obține adresa paginii virtuale și offset-ul în cadrul paginii. Adresa paginii este folosită ca index în cadrul tabelei de pagini. În poziția aferentă din cadrul tabelei de pagini se găsește adresa paginii fizice (frame) aferente. Această adrese este apoi cuplată cu offset-ul în cadrul paginii și se obține adresa fizică finală. Pentru eficiență se folosește TLB-ul în căutarea paginii fizice aferente paginii virtuale.
3. Dați un avantaj al folosirii demand paging în sistemele cu memorie virtuală.
    - **Răspuns**: Avantajul demand paging este timpul scurt folosit pentru "alocarea" memoriei și reducerea consumului de memorie fizică. Întrucât nu se consumă memorie fizică, "alocarea" va fi mai rapidă și nu va consuma din memoria RAM, lăsând-o disponiilă pentru acțiuni urgente. Memoria RAM va fi alocată și consumată on-demand (la nevoie).

3CA, varianta 2

1. Un programator apelează pipe(..). Cum poate fi transmis descriptorul rezultat pentru citire unui alt proces?
    - **Răspuns**: Descriptorii de fișiere pot fi transferați doar între procese înrudite. Un descriptor de proces va fi transferat către procesul/procesele copil în momentul apelării fork().
2. Cum rezolvă MMU adresa fizică a tabelei de pagini a procesului curent?
    - **Răspuns**: Adresa din memoria RAM a tabelei de pagini a procesului curent este dată de un registru dedicat numit generic PTBR (Page Table Base Register). Acest registru este încărcat de sistemul de operare cu valoarea aferentă procesului curent și este interogat de MMU.
3. Explicați modul în care apelul fork() inițializează memoria procesului nou creat.
    - **Răspuns**: Procesul nou creat partajează spațiul de memorie fizică al procesului părinte. Fiecare proces are spațiu virtual propriu și tabelă de pagini proprie; pentru procesul copil se creează o tabelă de pagini nouă și se copiază conținutul tabelei de pagini a procesului părinte. Spațiile virtuale de adrese ale celor două procese sunt mapate peste același spațiu fizic, care est marcat read-only. La accesul de scriere la o pagină din partea unuia dintre cele două procese, are loc page fault și copy-on-write: pagina fizică în cauză este duplicată, este marcată read-write și apoi peste ea este mapată pagina virtuală în care a avut loc page fault-ul.

3CB/CC, varianta 1

1. De ce un planificator de procese de pe un sistem de operare modern oferă, în general, o cuantă de timp mai mare proceselor I/O intensive?
    - **Răspuns**: Un proces I/O intensiv este așteptat să se blocheze repede și astfel, să cedeze procesorul. I se poate aloca o cuantă de timp mai mare (și o prioritate) mai mare pentru a rula cât mai rapid, pentru că oricum apoi va lăsa loc altor procese (CPU intensive sau I/O intensive).
2. Care este un avantaj al paginării memoriei față de segmentare?

- **Răspuns**: Paginarea conduce la eliminarea fragmentării externe: fragmentarea în afara blocurilor alocate. Spațiile libere sunt multiplu de pagină și pot fi alocate individual. În cazul segmentării ar trebui să găsim o zonă liberă suficientă pentru segmnetul care se alocă. Un alt avantaj este managementul mai ușor al memoriei din partea sistemului de operare, întreaga memorie fiind acum administrată la nivelul unei pagini (bloc de dimensiune fixă).

3. Furnizați două exemple de situații în care un page fault nu conduce la trimiterea unei excepții de memorie (de tipul segmentation fault) procesului.
    - **Răspuns**: Situații în care un page fault nu generează segmentation fault sunt:
        - se accesează o pagină care este swappată
        - se accesează o pagină care nu este încă paginată (comisă în RAM) și care va fi paginată prin demand paging
        - se accesează pentru scriere o pagină read-only marcată copy-on-write

3CB/CC, varianta 2

1. De ce este util să avem priorități dinamice pentru planificarea proceselor, nu priorități statice?
    - **Răspuns**: Dacă am folosi doar priorități statice, care nu se modifică pe durata execuției procesului, atunci procesul cu prioritatea statică cea mai bună ar acapara pentru o perioadă nedeterminată procesorul. Și alte procese ar ajunge să folosească puțin (spre deloc) procesorul, rezultând în starvation. Folosirea priorităților dinamice duce la alterararea la rulare a priorității proceselor și, în acest fel, la creșterea prioriății proceselor care nu au rulat și scăderea priorității celor care au rulat, ducând la o folosire mai echitabilă a procesorului și, în acest fel, la evitarea fenomenului de starvation.
2. De câtă memorie RAM poate dispune un sistem cu procese având spațiu virtual de adrese de o dimensiune dată (de exemplu 4GB)?
    - **Răspuns**: Nu există o legătură directă între dimensiunea spațiului virtual de adrese al procesului. Dimensiunea spațiului virtual de adrese este dată de capacitatea de adresare virtuală, iar spațiul fizic (RAM) de capacitatea de adresare fizică (dimensiunea magistralei de adrese dintre procesor și memoria RAM). În teorie, sistemul poate dispune de oricâtă memorie RAM (cât îi permite magistrala de adrese) independent de capacitatea spațiului virtual de adrese al procesului. De exemplu, un output al comenzii cat /proc/cpuinfo: address sizes : 36 bits physical, 48 bits virtual
        - În exemplul de mai sus, pot fi maxim 2^36 = 64GB RAM și 2^48 = 256 TB spațiu virtual de adrese. Nu e obligatoriu să avem mai mulți biți pentru adresele virtuale decât pentru adresele fizice.
3. Când apare fenomenul de “memory thrashing”?
    - **Răspuns**: Fenomenul de memory thrashing apare în momentul în care set size-ul (spațiul fizic folosit) al proceselor sistemului depășește capacitatea memoriei RAM: fie multe procese, fie procese cu consum mare de memorie RAM (set size mare). În acest caz, un proces care rulează are nevoie de multe pagini în RAM și le aduce de pe swap (swap in), evacuând pagini ale altui proces (swap out); apoi acel proces este planificat și face operația similară. Rezultă așadar un transfer constant între RAM și disc care afectează performanța.

Greșeli frecvente

Lucrări foarte bune

- POPA Ștefan-Adrian, 331CA
- SAVA Iulia, 331CA
- EFTIMIE Denis, 332CA
- DIACONU Cristian, 333CA
- MARINICĂ Elena, 335CA
- PĂNĂTĂU Liana-Maria, 335CA
- COTEȚ Teodor-Mihai 333CA
- CURCUDEL Ioan-Răzvan 335CA
- ROTSCHING Cristofor 343C1
- PRIPOAE Silvia 331CA
- ȘTEFĂNESCU Andrei 333CA
- SANDU Denisa 332CA
- ELISEI Alexandru 333CC
- IONESCU-TĂUTU Mihai Andrei, 331CA
- FUSU Elian, 332CB

Lucrare 3

- La începutul cursului 10:
    - luni, 8 mai 2017, 17:00-17:15, EC105, seria CA
    - miercuri, 10 mai 2017, 14:00-14:15, PR001, seriile CB/CC

3CA, varianta 1

1. Enumerați o caracteristică a limbajelor C/C++ care permite apariția vulnerabilităților de tip buffer overflow.
    - **Răspuns**: Nu exista bounds checking.
2. Dați un exemplu de cod care folosește instrucțiunea cmpxchg pentru a implementa un mutex.
    - **Răspuns**:

```
void lock (struct mutex* m){
        while(atomic_cmpxchg(s->val,1,0)==0){
                add_proc_to_mutex_waiting_list;
                scheduler();
        }
}
void unlock(struct mutex* m){
        atomic_set(s->val,1);
        mark_waiting_processes_as_ready;
}
```

3. Explicați cum pot fi lansate atacuri de tip buffer overflow împotriva unui server care execută fork() pentru a procesa cererile fiecărui client. Se folosesc stack canaries pe 32 biți.

- **Răspuns**: Valoarea canary de pe stiva clientului va fi aceeasi dupa fiecare fork; atacatorul poate incerca toate valorile posibile (4 miliarde) executand overflow cu un canar ales pana o gaseste pe cea corecta. Pentru a minimiza nr. de incercari se poate atata octet cu octet: astfel canary value va fi gasita in 1024 de incercari.

3CA, varianta 2

1. De ce nu sunt posibile buffer overflows în Java?
   - **Răspuns**: In Java, toate accesele la memorie sunt verificate sa fie “in bounds”.
2. Dați un exemplu de cod care folosește instrucțiunea cmpxchg pentru a implementa un semafor.
   - **Răspuns**:

```
void sem_up (struct sem* s){
        while(atomic_cmpxchg(s->mutex,1,0)==0);
        m->semval++;
        atomic_set(s->mutex,1)
        wake_up_waiting_processes;
}
void sem_down(struct sem* s){
        while (1){
                while(atomic_cmpxchg(s->mutex,1,0)==0);
                if (m->semval>0){
                        m->semval--;
                        atomic_set(s->mutex,1);
                        return;
                }
                else {
                        Add_process_to_waiting_list_for_s;
                        atomic_set(s->mutex,1);
                }
        }
}
```

3. Explicați cum putem afla adresa aproximativă a segmentului de cod pe un sistem de 64 biți cu ASLR care are o vulnerabilitate de tip buffer overflow. Procesul vulnerabil execută fork() pentru a procesa fiecare cerere a clienților.
   - **Răspuns**: Adresa de intoarcere de pe stiva poate fi leaked catre atacator astfel; atacatorul poate incerca sa ghiceasca primul octet executand overflow cu o valoare aleasa de el (e.g. 0); daca clientul continua sa functioneze, adresa a fost buna; altfel conexiunea se va inchide pentru ca client a fost terminat de SO; atacatorul incearca apoi urmatoarea valoare (adresa de retur este aceeasi pentru ca serverul face fork la fiecare client), si asa mai departe pana gaseste un octet bun. Se continua la fel cu urmatorii octeti pana cand se gaseste o adresa completa valida in segementul de cod de la server.

3CB/CC, varianta 1

1. Un program are o vulnerabilitate de tipul buffer overflow. Sistemul pe care rulează are suport DEP (Data Execution Prevention). Cum puteți obține un shell din exploatarea programului?

- **Răspuns**: Dacă sistemul are suport DEP, nu putem să injectăm cod pe care să-l executăm (în forma unui shellcode). Va trebui să apelăm cod deja existent. Pentru a obține un shell cel mai bine este să apelăm system("/bin/bash") adică să generăm un payload care să apeleze cod din biblioteca standard C (return-to-libc attack); facem acest lucru prin suprascrierea adresei de retur cu adresa funcției system() și a transmiterii corespunzătoare a adresei șirului "/bin/bash".

2. Numiți un avantaj și un dezavantaj al folosirii thread-urilor în locul proceselor pentru o aplicație care folosește paralelism în execuție.
    - **Răspuns**: În momentul în care avem o aplicației cu paralelism în execuție, folosirea thread-urilor are avantajul productivității: fiecare thread va rula pe un core și vom paraleliza astfel programul și vom obține speedup. Este de asemenea, foarte facil să partajăm informațiile între thread-uri. Dezavantajul este că thread-urile pot să acceseze în mod incoerent datele partajate (întreg spațiul de adrese al procesului) rezultând în erori în execuție sau rezultate nedeterministe. De asemenea, dacă un thread execută o operație nevalidă, întreg procesul își va încheia execuția.
3. Mai multe thread-uri folosesc o listă simplu înlănțuită ca structură comună. Parcurg, adaugă și șterg elemente din listă. Ce se întâmplă dacă nu asigurăm accesul exclusiv la listă?
    - **Răspuns**: Dacă nu asigurăm accesul exclusiv la listă pot avea loc ștergeri de noduri în vreme ce alte thread-uri accesează acele noduri. În acea situație, se poate ajunge ca un thread să acceseze date care acum sunt nevalide sau eliberate (use-after-free) sau să rezulte în segmentation fault.

3CB/CC, varianta 2

1. Descrieți o situație în care un apel fgets() are o vulnerabilitate de tipul buffer overflow. Semnătura funcției fgets() este char *fgets(char *s, int size, FILE *stream);
    - **Răspuns**: Funcția fgets() are o vulnerabilitate în cazul în care se citesc mai mulți octeți decât este dimensiunea buffer-ului. Astfel dacă avem o definiție de forma char buffer[32]; și apelăm fgets(stdin, 64, buffer), astfel încâte 64 > 32, vom avea buffer overflow și se va suprascrie dincolo de limitele buffer-ului.
2. Pe un sistem Unix modern, crearea unui proces folosind fork() este foarte rapidă. Totuși, durează de câteva ori mai mult decât crearea unui thread. Care e cauza principală pentru această diferență?
    - **Răspuns**: Atunci când un thread se creează în afara creării unei structurii de tipul TCB (Thread Control Block) nu se execută multe operații. În cazul unui proces, pe un sistem Unix modern, se creează un spațiu virtual de adresă nou, dar cu aceleași informații fizice ale procesului vechi. Adică se creează o tabelă de pagini nouă care este populată cu tabela de pagini a vechiului proces, rezultând într-un timp mai lung de creare a unui proces față de un thread.
3. Când este recomandat să folosim spinlock-uri în loc de mutex-uri?
    - **Răspuns**: Folosim spinlock-uri în loc de mutex-uri în cazul în care regiunea critică pe care o vom proteja este de mici dimensiuni. În această situație, overhead-ul de lock pe mutex durează prea mult față de timpul petrecut în regiunea critică și atunci folosim spinlock-uri. Regiunea critică de mici dimensiuni și fără operații blocante este un candidat pentru folosirea de spinlock-uri în loc de mutex-uri.

Greșeli frecvente

Lucrări foarte bune

Lucrare 4

- La începutul cursului 13:
    - luni, 22 mai 2017, 17:00-17:15, EC105, seria CA
    - miercuri, 24 mai 2017, 14:00-14:15, PR001, seriile CB/CC

3CA, varianta 1

1. Cum putem transfera datele de la un dispozitiv de intrare-ieșire utilizând cât mai puțin procesorul?
    - **Răspuns:** Intreruperi + DMA. Motive pentru care intreruperile ar fi nocive (e.g. dispozitiv de I/O prea rapid).
2. Un programator folosește două apeluri send() pentru a transmite cu TCP dimensiunea mesajului și conținutul său, respectiv, după care așteaptă răspunsul apelând recv(). Ce probleme de performanță pot aparea?
    - **Răspuns:** Interactiunea dintre algoritmul lui Nagle si mecanismul Delayed Ack care reduce performanta dramatic (aproximativ 5 mesaje pe secunda). Doua syscalls per mesaj in loc de un singur syscall - overhead mai mare.
3. Explicați diferența dintre un hard link și symlink într-un sistem de fișiere UNIX.
    - **Răspuns:** Un hardlink reprezinta legatura dintre dentry (sau proces in executie cu fisier deschis) si i-node. Un softlink este legatura inversa, de la un inode la un dentry.

3CA, varianta 2

1. Dați un exemplu în care este mai eficient să folosim polling decât întreruperi pentru a aștepta date de la un dispozitiv de intrare-ieșire.
    - **Răspuns:** Atunci cand dispozitivul de I/O genereaza intreuperi mai repede decat poate procesa sistemul gazda. Exemplu: placa de retea de 10Gbps sau mai rapida atunci cand pachetele sunt mici.
2. Ce ni se garantează atunci când apelul send(socket, …) întoarce N > 0?
    - **Răspuns:** Datele au fost copiate in buffer-ul nucleului de operare si vor fi transmise daca acest lucru e posibil (i.e. reteaua functioneaza corect).
3. De ce nu este corect să măsurăm performanța unui dispozitiv de stocare folosind doar apelul write()?
    - **Răspuns:** Apelul write se intoarce cand datele sunt scrise in memorie (in buffer cache) - trebuie sa fortam si sincronizarea (fsync) daca vrem sa masuram performanta dispozitivului.

3CC, varianta 1

1. Precizați un avantaj și un dezavantaj în folosirea operațiilor I/O asincrone.
    - **Răspuns:** Avantaj este că nu se blochează programul și putem rula mai multe acțiuni (asincrone) în paralel. Dezavantajul este că trebuie să avem mecanisme de așteptare a operațiilor asincrone care fac programarea mai dificilă; în momentul în care o operație se încheie ne va notifica de acest lucru (care va întrerupe fluxul normal de execuție) sau va trebui să apelăm o funcție dedicată (de obicei blocantă) de așteptare. Uzual este nevoie de un automat de stări care să mențină starea operațiilor asincrone, lucru de asemenea dezavantajos.

2. De ce este necesar suportul de TCP Offload Engine pentru legături de rețea de mare viteză (precum 10Gbit)?
    - **Răspuns:** Fără suport de TCP Offload Engine, prelucrarea pachetelor (la nivelul stivei TCP) în codul driverului/sistemului de operare durează suficient de mult încât să nu ajungă la viteza maximă de 10Gbit.
3. Un director conține 100 de intrări identificate ca fișiere obișnuite (regular files). Care este numărul minim de inode-uri referite de acele intrări?
    - **Răspuns:** Fiecare intrare de tip regular file pointează către un inode. Dacă toate intrările sunt link-uri hard ale aceluiași inode, atunci numărul minim de inode-uri este 1.

3CC, varianta 2

1. De ce este avantajoasă folosirea DMA (Direct Memory Access) în cazul operațiilor de intrare/ieșire?
    - **Răspuns:** Folosirea DMA duce la eliminarea procesorului din fluxul de prelucrare a unor date. Datele ajung de la dispozitiv la memoria RAM și invers fără intervenția procesorului. În felul acesta procesorul poate fi folosit pentru activități precum rularea codului proceselor, mărind productivitatea sistemului.
2. Un apel send se blochează pentru că buffer-ul de send din kernel al socket-ului este plin. Care este o cauză posibilă pentru acest lucru?
    - **Răspuns:** Buffer-ul de send al unui socket se poate bloca din cauză că a apărut o congestie la un middlebox pe traseul pachetului. Până la eliberarea congestiei bufferul de send rămâne plin și blochează apelul send(). Pe lângă aceasta, buffer-ul de receive al destinatarului/receiver-ului se poate să fie umplut și nu poate primi noi date, blocând și buffer-ul de send. Deblocarea se va face în momentul în care se realizează un apel de tip recv() la receiver care va goli parte din buffer-ul de receive și astfel și parte din buffer-ul de send.
3. Ce conțin blocurile de date indicate de inode-ul unui director?
    - **Răspuns:** Blocurile de date indicate de inode-ul unui director conțin un vector de dentry-uri (intrări de tip directory) pentru intrările conține de director. Acestea conțin, în general, numele intrărilor și indexul inode-ului.