Tema 2: Procese și Threaduri
Student: Dina Pop

1 Descrierea temei

1.1 Procese și Threaduri

În această temă trebuie create procese și threaduri în Linux, a căror execuție trebuie sincronizată. În afară de sincronizare, procesele și thread-urile voastre vor afișa doar câteva mesaje și se vor încheia. Sincronizările cerute vor influența ordinea în care mesajele sunt afișate.

Pentru ca soluția voastră să poată fi testată automat, procesele și thread-urile voastre trebuie să apeleze în locul potrivit și în ordinea corectă câteva funcții, furnizate în fișierele "*a2_helper.h*" și "*a2_helper.c*". Pentru aceasta, următoarele cerințe trebuie satisfăcute:

1. Procesul principal (cel care se creează la lansarea programului și este ilustrat în Figura 1 ca P_1) trebuie să înceapă cu apelul funcției "`init()`", a cărei rol este pregătirea programului pentru interacțiunea cu tester-ul.

2. Toate procesele (adică atât cel principal cât și cele create explicit de programul vostru), trebuie să apeleze funcția "`info()`" la început și înainte de final, cu anumite argumente predefinite, după cum se ilustrează în caseta de mai jos, unde se poate observa cum ar trebui să arate codul programului principal și al unuia dintre procesele create.

Sample Process Code

```
int main(int argc, char **argv)
{
    // tester initialization
    // only one time in the main process
    init();

    // inform the tester about (main) process' start
    info(BEGIN, process_no, thread_no);

    // other process' actions
    ...

    // create a new process
    if (fork() == 0) {
        // inform the tester about process' start
        info(BEGIN, process_no, thread_no);
```

```
        // other process' actions
        ...

        // inform the tester about process' termination
        info(END, process_no, thread_no);
    }

    // inform the tester about (main) process' termination
    info(END, process_no, thread_no);
}
```

3. Fiecare thread trebuie să apeleze funcția "`info()`" la început, respectiv înainte de final, cu anumite argumente predefinite, după cum se ilustrează în caseta de mai jos.

Sample Thread Code

```
void thread_function(void* arg)
{
    // possible thread' actions
    ...

    // inform the tester about thread start
    info(BEGIN, process_no, thread_no);

    // other possible thread' actions
    ...

    // inform the tester about thread termination
    info(END, process_no, thread_no);
}
```

Funcția "`info()`" poate fi considerată (din punctul vostru de vedere) ca o funcție ce afișează un mesaj. Argumentele sale au următoarea semnificație:

1. primul argument este una din constantele definite în fișierul "*a2_helper.h*", respectiv "`BEGIN`" și "`END`", marcând începutul sau finalul unui proces sau a unui thread;
2. al doilea argument este un întreg, ce va identifica în mod unic procesul apelant, după cum se va explica mai jos;
3. al treilea argument este tot un întreg, ce va identifica în mod unic threadul apelant, după cum se va explica mai jos.

1.2 Convenția de nume pentru procese și threaduri

Programul vostru va trebui să creeze anumite procese, după cum va fi descris în Secțiunea 2. De asemenea trebuie să asocieze identificatori unici acelor procese, independenți de de identificatorii unici generați de sistem. Astfel, atunci când

se cere generarea a N procese, identificatorii asociați vor fi 1, 2, ..., N. Modul de asociere pentru acești identificatori va fi descris mai jos. Ne vom referi la procese în acest document folosind etichetele derivate din acești identificatori, respectiv P_1, P_2, ..., P_N pentru procesele 1, 2, ..., and N.

Fiecare proces, la rândul lui, poate să creeze mai multe thread-uri. Când se discută despre thread-urile create de un anumit proces, trebuie să fiți atenți la faptul că thread-ul implicit al procesului (thread-ul principal) nu se numără. Astfel, dacă pentru un proces se cere crearea a N thread-uri, acesta trebuie să apeleze funcția de creare de thread-uri de N ori.

În mod similar cu procesele, fiecăruia din thread-urile unui proces i se asociază de asemenea un identificator unic (din nou, diferite de cele asociate implicit de către sistemul de operare). Astfel, pentru N thread-uri create, identificatorii asociați vor fi 1, 2, ..., N. Identificatorii de thread-uri vor fi asociați în ordine crescătoare, în funcție de ordinea creației: primul thread va avea identificatorul 1, al doilea thread identificatorul 2, ș.a.m.d. Se consideră că thread-ul principal are identificatorul 0. Observați că identificatorii thread-urilor sunt unice doar pentru thread-urile din același proces, putând lua aceleași valori în procese diferite.

În cerințele noastre ne vom referi la threadurile proceselor folosind etichete similare cu cele pentru procese. Deoarece seturile de identificatori de thread-uri pentru fiecare proces nu sunt disjuncte, un thread va fi identificat în mod unic prin perechea formată din identificatorul procesului, respectiv identificatorul threadului. Acest lucru se poate observa în modul în care apelăm funcția "`info()`". Vom folosi următoarea convenție de nume: threadul $T_{N.M}$ este threadul cu numărul M din procesul N. Gama de valori pentru M este între 1 și numărul de threaduri create de proces.

Bazându-ne pe convențiile de denumire de mai sus, scheletul de cod pentru procesul P_1 (cel principal) și P_2 (cel creat de P_1), considerând că rulează pe threadurile principale ($T_{1.0}$ și $T_{2.0}$) va arăta precum mai jos.

Sample Code for Processes P_1 and P_2

```
int main(int argc, char **argv)
{
    // tester initialization
    // only one time in the main process
    init();

    // inform the tester about (main) process' start
    info(BEGIN, 1, 0);

    // other process' actions
    ...

    // create a new process
    if (fork() == 0) {
        // inform the tester about process' start
        info(BEGIN, 2, 0);

        // other process' actions
        ...
```

```
        // inform the tester about process' termination
        info(END, 2, 0);
    }

    // inform the tester about (main) process' termination
    info(END, 1, 0);
}
```

În mod similar, scheletul de cod pentru unul din threadurile lui P_2 (de exemplu $T_{2.5}$) va arăta în felul următor:

Sample Code for Thread $T_{2.5}$

```
void thread_function(void* arg)
{
    // possible thread' actions
    ...

    // inform the tester about thread start
    info(BEGIN, 2, 5);

    // other possible thread' actions
    ...

    // inform the tester about thread termination
    info(END, 2, 5);
}
```

1.3 Mecanisme și strategii de sincronizare

Puteți utiliza orice mecanism de sincronizare doriți, cum ar fi:

- semafoare System V
- semafoare POSIX
- lacăte și variabile condiționale POSIX

Puteți deasemenea utiliza o combinație de mecanisme de sincronizare.

Cerințele de sincronizare a threadurilor descrise mai jos se referă de obicei la ordinea în care mesajele de început și sfârșit trebuie să apară. Această ordine se poate controla utilizând mecanisme de sincronizare înainte să apelați funcția "`info()`".

La alte cerințe mecanismele de sincronizare se pot referi la numărul maxim de threaduri ce pot rula simultan într-un anumit proces. În mod normal, acest lucru înseamnă că threadul principal trebuie să sincronizeze crearea altor threaduri. Totuși, deoarece testerul ia în considerare doar apelurile funcției "`info()`", această cerință poate fi înțeleasă și ca numărul de maxim de thread-uri ce se află simultan între mesajele BEGIN și END. În acest caz, threadurile create pot controla ele însele câte pot ajunge simultan în regiunea dintre cele două mesaje.

Un astfel de mecanism de sincronizare ar trebui realizat înainte de a se apela "`info(BEGIN, ...)`".

Tipurile de sincronizare menționate mai sus vor duce la scheletul de funcții din caseta de mai jos.

Sample Code for Processes P_1 and P_2 With Synchronization

```
int main(int argc, char **argv)
{
    // tester initialization
    // only one time in the main process
    init();

    // inform the tester about (main) process' start
    info(BEGIN, 1, 0);

    // other process' actions
    ...

    // create a new process
    if (fork() == 0) {
        // inform the tester about process' start
        info(BEGIN, 2, 0);

        // other process' actions
        ...

        // calls to synchronization mechanisms
        ...

        // new thread creation
        pthread_create(...);

        // inform the tester about process' termination
        info(END, 2, 0);
    }

    // calls to synchronization mechanisms
    ...

    // new thread creation
    pthread_create(...);

    // inform the tester about (main) process' termination
    info(END, 1, 0);
}
```

Sample Code for Thread $T_{2.5}$ With Synchronization

```
void thread_function(void* arg)
{
    // possible thread' actions
    ...

    // calls to synchronization mechanisms
    ...

    // inform the tester about thread start
    info(BEGIN, 2, 5);

    // other possible thread' actions
    ...

    // calls to synchronization mechanisms
    ...

    // inform the tester about thread termination
    info(END, 2, 5);
}
```

Se poate observa că același proces poate crea deopotrivă procese și threaduri noi, cum e cazul lui P_1 în codul de mai sus. În timpul creări unui proces, tot ce este în părinte se copiază în copil, inclusiv posibilele threaduri. care s-au creat anterior. Totuși, în procesul copil doar threadul care a apelat "`fork()`" va fi activ, celelalte nu vor mai rula. Din cauza complicațiilor ce pot apărea, recomandăm ca pentru un proces ce crează deopotrivă procese și threaduri, să se creeze întâi procesele, apoi threadurile. Acest lucru este sugerat și în exemplul de cod de mai sus.

2 Cerințele temei

Trebuie să scrieți un program C numit "*a2.c*", extinzând scheletul existent, în care cerințele de mai jos vor fi implementate.

2.1 Compilarea și rularea

Programul vostru trebuie să poată fi **compilat fără erori** pentru a fi acceptat. Comanda de compilare va fi cea de mai jos:

Compilation Command

```
gcc -Wall a2.c a2_helper.c -o a2 -lpthread
```

Warning-uri la compilare vor aduce o penalizare de 10% la scorul final.

Atunci când rulează, executabilul vostru (îl vom numi "*a2*") **trebuie să ofere funcționalitatea minimală și rezultatele așteptate**, pentru a fi acceptat. Vom defini mai jos ce înseamnă funcționalitatea minimală. Următoarea casetă ilustrează modul în care programul va fi rulat.

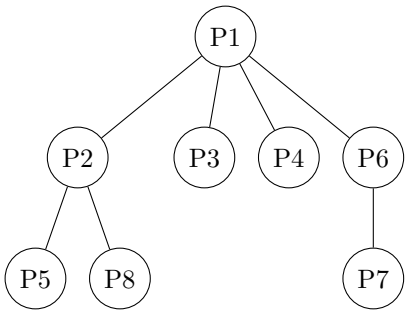

Figure 1: Ierarhia de procese cerută

Running Command

```
./a2
```

2.2 Ierarhia de procese

Soluția voastră trebuie să genereze ierarhia de procese din Figura 1.

Nici un proces care are copii nu trebuie să se termine înaintea copiilor săi. Raportat la mesajele afișate prin funcția "`info()`", mesajul END al unui părinte nu trebuie să apară înaintea mesajului END a copiilor.

2.3 Sincronizarea threadurilor din același proces

Procesul P7 trebuie să creeze 4 threaduri: T7.1, T7.2, ..., T7.4. Următoarele condiții trebuie impuse asupra threadurilor din P7:

- Threadul principal al procesului (T7.0) nu trebuie să se termine înaintea celorlalte 4 threaduri. Asta înseamnă că nu trebuie să se apeleze "`info(END, ...)`" înaintea apelului similar din celelalte threaduri. Practic mesajul END al threadului prinicipal trebuie să apară ultimul.
- Threadul T7.1 trebuie să înceapă înainte ca T7.3 să înceapă și trebuie să se încheie după terminarea acestuia.

2.4 Barieră pentru threaduri

Procesul P2 trebuie să creeze 44 threaduri: T2.1, T2.2, ..., T2.44. Următoarele condiții trebuie impuse asupra threadurilor din P2:

- Threadul principal al procesului (T2.0) nu trebuie să se termine înainte celorlalte 44 threaduri.
- În orice moment, cel mult 6 threaduri din procesul P2 pot rula simultan, fără a se număra threadul principal.
- Threadul T2.12 nu are voie să se încheie decât în timp ce 6 threaduri (cu tot cu el însuși) rulează.

2.5 Sincronizarea threadurilor din procese diferite

Procesul P6 trebuie să creeze 4 threaduri: T6.1, T6.2, ..., T6.4. Următoarele condiții de sincronizare trebuie impuse asupra threadurilor din P2 și P7:

- Threadul principal al procesului (T6.0) nu trebuie să se termine înainte celorlalte 4 threaduri.
- Threadul T6.1 trebuie să se încheie înainte ca threadul T7.4 să pornească, dar threadul T6.4 nu poate începe decât după ce T7.4 s-a terminat.

3 User Manual

3.1 Auto-evaluare

Pentru a genera și rula testele pentru soluția voastră, trebuie rulat script-ul "tester.py", furnizat împreună cu cerințele. Script-ul are nevoie de Python 3.x, nu de Python 2.x.

Sample Command for Tests Generation

```
python3 tester.py
```

Chiar dacă script-ul funcționează și pe Windows (și pe alte sisteme de operare), recomandăm rularea pe Linux, deoarece așa se va evalua tema voastră.

Script-ul va compila și va rula programul vostru de mai multe ori și va afișa scorul obținut pe baza mesajelor primite de la program.

Restricții:

- La această temă, este interzisă orice interacțiune cu sistemul de fișiere.
- Nu aveți voie să efectuați următoarele apeluri de sistem (nici să le apelați în mod indirect, prin funcții de nivel înalt):
  - `sleep()`
  - `usleep()`
  - `pthread_atfork()`
- Nu aveți voie să comunicați cu script-ul de testare în alte moduri decât prin funcțiile furnizate "`init()`" și "`info()`".

Atunci când tema voastră este evaluată pentru notă, aceasta se rulează într-un container de *docker*. Chiar dacă o soluție corectă și deterministă se va comporta la fel, un bug în soluție poate trece neobservat pe sistemul vostru dar poate produce un crash / comportament nedefinit la evaluarea "oficială". Din acest motiv vă încurajăm să vă testați soluția și cu docker înainte să o trimiteți. Pentru asta trebuie să:

- instalați *docker* pe sistem (`sudo apt install docker.io`)
- creați un cont pe Docker Hub și să vă logați din linia de comandă (e necesar pentru a descărca imaginea de evaluare)
- instalați modulul *docker* pentru Python (`python3 -m pip install docker`)

Ca să vă testați soluția folosind un container de *docker* trebuie să dați argumentul `--docker` script-ului de testare.

Sample Command for testing with Docker

```
python3 tester.py --docker
```

Dacă doriți să păstrați container-ul de docker ulterior rulării testelor (de exemplu pentru a putea face debug în interiorul acestuia), folosiți opțiunea `--docker-persist`.

3.2 Evaluation and Minimum Requirements

- Dacă apar warning-uri la compilare, se va aplica o penalizare de 10%.
- Dacă programul nu respectă stilul de cod, se poate aplica o penalizare de până la 10% (decisă de cadrul didactic de la laborator).

Nu încercați să trișați, deoarece se vor rula tool-uri de detecție a temelor plagiate.

Notă: Testele furnizate nu sunt neapărat aceleași cu cele pe care se va testa soluția voastră. Deasemenea, verificați-vă codul în privința implementării corecte și complete a cerințelor, deoarece prin simpla rulare a unui set de teste nu se garantează faptul că soluția este perfectă. Este posibil ca unele cazuri mai rare să nu apară în testele voastre, dar să fie testate la evaluare.