Enunt

Sa se implementeze un monitor generic care sa poata fi folosit de catre mai multe fire de executie pentru rezolvarea unor probleme de sincronizare. Monitorul va fi generic in sensul ca va contine un numar oarecare (precizat la crearea monitorului) de variabile conditie pe care pot astepta firele de executie, facand astfel posibila rezolvarea oricarei probleme de sincronizare.

Implementarea monitorului se va face intr-o biblioteca partajata dinamica pe care o vor incarca firele de executie care doresc sa foloseasca monitorul in scopul sincronizarii.

Trebuie sa construiti o structura de date `Monitor` care sa contina variabilele conditie ale monitorului, cozi de asteptare, cine este posesorul monitorului, politica de functionare a monitorului, contoare, flaguri, obiecte native de sincronizare sau orice alte elemente pe care le considerati necesare pentru asigurarea functionalitatii monitorului.

Biblioteca Monitor trebuie sa exporte urmatoarele metode de lucru cu monitorul:

- `Monitor* Create(int conditions, char policy)` - va crea un monitor cu numarul de variabile conditie si cu politica de functionare specificate ca parametri;
- `int Enter(Monitor *m)` - firul de executie apelant va incerca sa intre in monitor si daca monitorul este deja ocupat se va bloca la coada Entry.
- `int Leave(Monitor *m)` - va fi apelat de catre un fir de executie care doreste sa paraseasca monitorul.
- `int Wait(Monitor *m, int cond)` - firul de executie apelant se va bloca la variabila conditie `cond`.
- `int Signal(Monitor *m, int cond)` - se va semnaliza variabila conditie `cond` deblocandu-se astfel un fir de executie care astepta la variabila conditie respectiva. Daca nici un alt fir de executie nu asteapta la variabila conditie `cond`, apelul `signal` nu va avea nici un efect.
- `int Broadcast(Monitor *m, int cond)` - se va semnaliza variabila conditie `cond` deblocandu-se astfel toate firele de executie care asteptau la variabila conditie respectiva. Daca nici un alt fir de executie nu asteapta la variabila conditie `cond`, apelul `broadcast` nu va avea nici un efect.
- `int Destroy(Monitor *m)` - va distruge monitorul eliberand toate resursele alocate la crearea acestuia.

Toate functiile intorc valoarea -1 in caz de eroare. Functia Create intoarce NULL in caz de eroare. Functiile intorc 0 in caz de succes, mai putin Create care intoarce un pointer la monitor

Trebuie verificate cazurile de folosire incorecta a monitorului precum:

- `Enter` apelat de catre un fir de executie care este deja in monitor;
- `Leave, Wait, Signal, Broadcast` - apelat de catre un fir de executie care nu este in monitor;
- `Destroy` - apelat cand monitorul nu este liber, mai exista un fir de executie activ in monitor.

Monitorul va functiona dupa una din politicile SIGNAL AND CONTINUE sau SIGNAL AND WAIT. Daca monitorul functioneaza dupa politica SIGNAL AND CONTINUE, un fir de executie care executa `Signal` isi va continua executia fara sa elibereze monitorul. Daca monitorul functioneaza dupa politica SIGNAL AND WAIT, un fir de executie care executa `Signal` cedeaza monitorul firelor de executie trezite. Politica de functionare a monitorului se va specifica la crearea acestuia. Implementarea ambelor politici de functionare este obligatorie.

La orice moment de timp un singur fir de executie dintre cele intrate in monitor se poate executa. Cu alte cuvinte, dintre toate firele de executie intrate in monitor (firele care au executat `Enter` si inca nu au executat `Leave`) doar unul va executa cod efectiv, celelalte fiind blocate in cozile de asteptare ale monitorului. Acesta reprezinta un invariant pentru orice tip de monitor.

Intr-un monitor exista, dupa cum veti afla din documentatie, 3 cozi de fire de executie in asteptare care concureaza la intrarea in monitor:

- Coada firelor de executie ce au executat `enter` - Entry Queue;
- Coada firelor de executie ce au fost trezite de un `Signal` si vor monitorul pentru a continua - Waiting Queue;
- Coada firelor de executie care au facut `Signal` si asteapta sa intre din nou in monitor pentru ca politica era SIGNAL AND WAIT- Signaller Queue;

Cand aveti fire de executie in toate cozile, coada firelor de executie blocate la intrare - Entry Queue - este cea mai putin prioritara.

Folosind monitorul astfel implementat veti rezolva problema clasica de sincronizare Readers Writters cu prioritate pe writers. Aveti un buffer partajat intre N cititori si M scriitori. Din buffer pot citi simultan mai multi cititori daca nu exista nici un scriitor in buffer. In buffer nu se pot afla simultan mai multi scriitori. Daca un scriitor gaseste un alt scriitor asteapta pana cand acesta paraseste bufferul. Daca un scriitor gaseste cititori asteapta pana cand parasesc acestia bufferul. Daca un cititor gaseste un scriitor in buffer sau descopera ca un scriitor asteapta sa intre in buffer nu intra si ramane in asteptare acordand astfel prioritate scriitorilor.

Pentru aceasta veti implementa o a doua biblioteca care ofera functiile:

- `void StartCit(Monitor* m);` - va fi apelata in test atunci cand cititorul vrea sa inceapa sa citeasca
- `void StopCit(Monitor *m);` - va fi apelata in test atunci cand cititorul vrea sa se opreasca din citit
- `void StartScrit(Monitor* m);` - va fi apelata in test atunci cand scriitorul vrea sa inceapa sa scrie
- `void StopScrit(Monitor *m);` - va fi apelata in test atunci cand scriitorul vrea sa se opreasca din scris
- `int GetNrConds();` - returneaza numarul de variabile de conditie cu care a fost creat monitorul utilizat la implementarea acestor functii
- `Monitor* CreateRWMonitor();` - Creaza monitorul folosit la implementarea functiilor. (Avand numarul de variabile de conditie si politica de functionare pe care le considerati necesare)

Precizari generale

Pentru documentare aveti la dispozitie urmatoarele doua documente: Buhr, Fortier: Monitor classification si Hoare,Monitors: An Operating System Structuring Concept. Primul dintre ele este foarte important. Atentie ca operatiile pe care le implementati (`Enter, Leave, Wait, Signal`) trebuie sa respecte specificatiile formale, preconditiile si postconditiile specificate prin axiomele prezentate in primul document pentru fiecare politica in parte. Axiomele respective vor face parte din baremul de corectare. Politica SIGNAL AND WAIT la care se face referire in enuntul temei se refera la politica SIGNAL AND URGENT WAIT in terminologie Buhr & Fortier.

Deoarece specificatiile formale prezentate in documentatie prin axiome pot fi dificil de inteles aveti mai jos descrierea in limbaj natural a specificatiilor monitorului. Aceasta descriere contine si specificatiile pentru operatia broadcast care lipseste din documentatie. Monitorul va functiona dupa una din politicile SIGNAL AND WAIT sau SIGNAL AND CONTINUE. Aceste doua politici sunt descrise la paginile 13 si 14 din documentatie sub forma 4.2.2PB si 4.2.4 PNB.

Politica SIGNAL AND WAIT

Prioritatile sunt: thread din Entry mai putin prioritar decat thread din Signaller Queue. Thread din Signaller Queue mai putin prioritar decat thread din Waiting Queue. Exista 5 posibilitati:

- **un thread executa Enter:** Daca monitorul nu este liber, threadul se va bloca in coada Entry. Functia Enter nu se va intoarce decat in momentul in care threadul va fi planificat sa intre in monitor, de catre un alt thread. Acest lucru se va intampla la o planificare la care vom avea in mod cert Signaller Queue si Waiting Queue vide, deoarece, datorita prioritatilor acestora, daca ar fi existat un thread in oricare din aceste cozi ar fi fost planificat inaintea celui din coada Entry.
- **un thread executa Signal pe conditia q:** Signaller Queue va creste cu unu, deoarece politica este signal and wait; coada de asteptare la conditia q, daca e nevida se va micsora cu unu; coada Waiting Queue se va mari cu unu in cazul in care coada de asteptare la conditia q este nevida. Asadar, daca exista threaduri care astepta la conditia q, unul dintre acestea va fi scos din coada de asteptare la conditia q si va fi pus in coada de asteptare Waiting. Acum se va planifica un alt thread care sa devina activ, iar threadul curent se va bloca in coada Signaller. Functia Signal se va intoarce abia in momentul cand threadul curent va fi trezit din coada Signaller de catre alt thread si planificat. Inainte de blocarea la Signaller Queue planificarea se va face astfel: Daca Waiting Queue nu e vida se va lua un thread din Waiting Queue si se va trezi; daca Waiting Queue e vida (ceea ce inseamna ca si coada de asteptare la conditia q era vida, altfel Waiting Queue ar fi avut cel putin un element) se va alege un thread din Signaller Queue (aceasta in mod sigur nu e vida deoarece contine cel putin threadul care tocmai a facut signal) si se va planifica acesta.

- **un thread executa Broadcast pe conditia q:** Signaller Queue va creste cu unu, deoarece politica este **signal and wait**; coada de asteptare la conditia q, daca e nevida, va deveni vida; coada Waiting Queue se va mari cu dimensiunea cozii de asteptare la conditia q. Asadar, daca exista threaduri blocate la conditia q, toate vor fi scoase din coada de asteptare la conditia q si vor fi puse in coada de asteptare Waiting. Acum se va planifica un alt thread care sa devina activ, iar threadul curent se va bloca in coada Signaller. Functia Broadcast se va intoarce abia in momentul cand threadul curent va fi trezit din coada Signaller de catre alt thread si planificat. Inainte de blocarea la Signaller Queue planificarea se va face astfel: Daca Waiting Queue nu e vida se va lua un thread din Waiting Queue si se va planifica; daca Waiting Queue e vida (ceea ce implica ca si coada de asteptare la conditia q era vida, altfel Waiting Queue ar fi avut cel putin cate elemente a avut coada de asteptare la conditia q) se va alege un thread din Signaller Queue (aceasta in mod sigur nu e vida deoarece contine cel putin threadul care tocmai a facut broadcast) si se va planifica acesta.
- **un thread executa Wait pe conditia q:** Se va planifica un alt thread pentru executie dupa care threadul curent se va bloca la coada de asteptare a conditiei q. FunctiaWait se va intoarce si threadul curent (blocat la q) isi va continua executia in monitor abia dupa ce conditia q va fi fost semnalizata, threadul curent va fi trecut in coada Waiting si un alt thread va planifica threadul curent din Waiting. Inainte de blocarea la conditia q planificarea se va face astfel: Daca exista cel putin un thread in Waiting Queue, acesta va fi planificat (primul intalnit, unul arbitrar ales, oricum un thread din Waiting Queue, deoarece aceasta are prioritatea cea mai mare). Daca nu exista nici un thread in Waiting Queue, se va incerca sa se gaseasca unul in Signaller Queue. Daca s-a gasit unul in Signaller Queue, se va planifica acesta (primul gasit, unul arbitrar - depinde de implementarea pe care o alegeti). Daca si aceasta coada(Signaller Queue) e vida se va alege unul din Entry Queue. Daca s-a gasit unul in Entry Queue, se va planifica acesta (la fel, alegerea va apartine, voi decideti care va fi cel ales din Entry Queue). Daca si aceasta(Entry Queue) va fi goala, threadul curent nu are pe cine sa planifice si se va bloca (eliberand monitorul) la conditia q, asteptand ca un alt thread sa intre in monitor si sa execute un Signal q.
- **un thread executa Leave:** Inainte de a parasi monitorul, un thread va planifica pe altcineva. Planificarea se va face conform priorittilor de la SIGNAL AND WAIT: Daca exista cel putin un thread in Waiting Queue, acesta va fi planificat (primul intalnit, unul arbitrar ales, oricum un thread din Waiting Queue, deoarece aceasta are prioritatea cea mai mare). Daca nu exista nici un thread in Waiting Queue, se va incerca sa se gaseasca unul in Signaller Queue. Daca s-a gasit unul in Signaller Queue, se va planifica acesta (primul gasit, unul arbitrar - depinde de implementarea pe care o alegeti). Daca si aceasta coada(Signaller Queue) e vida se va alege unul din Entry Queue. Daca s-a gasit unul in Entry Queue, se va planifica acesta (la fel alegerea va apartine). Daca si aceasta (Entry Queue) va fi goala, threadul curent nu are pe cine sa planifice si va parasi monitorul.

POLITICA SIGNAL AND CONTINUE

Prioritatile sunt thread din Entry mai putin prioritar decat thread din Waiting Queue. Thread din Waiting Queue mai putin prioritar decat thread din Signaller Queue. Exista 5 posibilitati:

- **un thread executa Enter:** Daca monitorul nu este liber, threadul se va bloca in *coada Entry*. Functia Enter nu se va intoarce decat in momentul in care threadul va fi planificat sa intre in monitor, de catre un alt thread. Acest lucru se va intampla la o planificare la care vom avea in mod cert *Waiting Queue* si *Signaller Queue* vide, deoarece, datorita prioritatilor acestora, daca ar fi existat un thread in oricare din aceste cozi ar fi fost planificat inaintea celui din *coada Entry*.
- **un thread executa Signal pe conditia q:** Daca coada de asteptare la conditia q nu este vida , *Waiting Queue* va creste cu 1, iar coada de asteptare la conditia q se va micsora cu 1 (adica va fi scos un thread din coada de asteptare la conditia q si va fi trecut in coada de asteptare *Waiting Queue*). Cum politica este SIGNAL AND CONTINUE, apelul Signal se va intoarce imediat iar threadul apelant isi va continua executia.
- **un thread executa Broadcast pe conditia q:** Daca coada de asteptare la conditia q nu este vida , *Waiting Queue* va creste cu marimea lui q (|q|), iar coada de asteptare la conditia q va deveni vida (|q|=0). Asadar threadurile care asteptau in coada de asteptare la conditia q , vor trece in coada de asteptare *Waiting Queue*. Cum politica este SIGNAL AND CONTINUE, apelul broadcast se va intoarce imediat, iar threadul apelant isi va continua executia.
- **un thread executa Wait pe conditia q:** Se va planifica un alt thread pentru executie dupa care threadul curent se va bloca la coada de asteptare a conditiei q. FunctiaWait se va intoarce si threadul curent (blocat la q) isi va continua executia in monitor abia dupa ce conditia q va fi fost semnalizata, threadul curent trecut in *coada Waiting*si un alt thread va planifica threadul curent din *Waiting*. Inainte de blocarea la conditia q planificarea se va face astfel: *Signaller Queue* e sigur vida, deoarece orice thread care face Broadcast sau Signal se executa in continuare fara sa se blocheze la *Signaller Queue*. Daca exista cel putin un thread in *Waiting Queue*, acesta va fi planificat (primul intalnit, unul arbitrar ales, oricum un thread din *Waiting Queue*). Daca nu exista nici un thread in *Waiting Queue*, se va alege unul din *Entry Queue*. Daca s-a gasit unul in *Entry Queue*, se va planifica acesta (la fel alegerea va apartine). Daca si aceasta(*Entry Queue*) va fi goala, threadul curent nu are pe cine sa planifice si se va bloca (eliberand monitorul) la conditia q, asteptand ca un alt thread sa intre in monitor si sa execute un Signal q.
- **un thread executa Leave:** Inainte de a parasi monitorul, un thread va planifica pe altcineva. Planificarea se va face conform prioritatilor de la SIGNAL AND CONTINUE:*Signaller Queue* e sigur vida, deoarece orice thread care face Broadcast sau Signal se executa in continuare fara sa se blocheze la *Signaller Queue*. Daca exista cel putin un thread in *Waiting Queue*, acesta va fi planificat (primul intalnit, unul arbitrar ales, oricum un thread din Waiting Queue). Daca nu exista nici un thread in *Waiting Queue*, se va alege unul din *Entry Queue*. Daca s-a gasit unul in *Entry Queue*, se va planifica acesta (la fel alegerea va apartine). Daca si aceasta(Entry Queue) va fi goala, threadul curent nu are pe cine sa planifice si va parasi monitorul.

Observati similitudinea dintre monitor si nucleul unui sistem de operare. Se poate face o analogie intre functiile exportate de monitor si apelurile de sistem exportate de un nucleu. Precum unele apeluri de sistem si aceste apeluri pot bloca procesul (in cazul de fata firul de executie) apelant, alt proces din monitor fiind planificat pentru executie.

Precum pe o masina uniprocesor un singur proces poate rula la un moment dat pe procesor la fel in monitor un singur fir de executie poate fi activ la un moment dat.

Precizari Windows

Tema de Windows trebuie sa implementeze cele doua biblioteci partajate ca DLL-uri, sub numele de `LibMonitor.DLL` si `LibRW.DLL`. Pentru instructiuni de construire a unui DLL consultati Platform SDK.

Datorita faptului ca in Windows DLL-urile nu pot contine simboluri nedefinite, in momentul in care construiti DLL-ul va trebuie sa o link-ati cu `ControlMonitor.obj` si`ControlRW.obj`, fisiere obtinute din pasul 1 de compilare al testelor. Exemplu:

```
build: LibMonitor.obj LibRW.obj
        link /release /dll /out:LibMonitor.dll  LibMonitor.obj
ControlMonitor.obj
        link /release /dll /out:LibRW.dll LibRW.obj  ControlRW.obj
LibMonitor.lib
```

Pentru a testa utilizarea corecta a monitorului mentineti un index in Thread Local Storage care indica pentru fiecare fir de executie daca se afla sau nu in interiorul monitorului. Nu e necesar sa alocati memorie, puteti folosi doar valoarea pointerului din indexul TLS. Pentru indicatii privind Thread local Storage pe Windows consultati urmatoarele linkuri din Platform SDK:
[ms-help://MS.PSDKXPSP2.1033/dllproc/base/thread_local_storage.htm ms-help://MS.PSDKXPSP2.1033/dllproc/base/thread_local_storage.htm]
[ms-help://MS.PSDKXPSP2.1033/dllproc/base/using_thread_local_storage.htm ms-help://MS.PSDKXPSP2.1033/dllproc/base/using_thread_local_storage.htm]
[ms-help://MS.PSDKXPSP2.1033/dllproc/base/using_thread_local_storage_in_a_dynamic_link_library.htm ms-help://MS.PSDKXPSP2.1033/dllproc/base/using_thread_local_storage_in_a_dynamic_link_library.htm]

Implementarea temei de Windows trebuie sa nu foloseasca functia PulseEvent.

Tema se va rezolva folosind doar functii Win32. Se pot folosi de asemenea si functiile de formatare `printf`, `scanf`, functiile de alocare de memorie `malloc`, `free` si functiile de manipulare a sirurilor de caractere (`strcat`, `strdup`, etc.)

Pentru partea de I/O si procese se vor folosi doar functii Win32. De exemplu, functiile `open`, `read`, `write`, `close` nu trebuie folosite, in locul acestor trebuind sa folositi `CreateFile`,`ReadFile`, `WriteFile`, `CloseHandle`.

Precizari Linux

Tema de Linux trebuie sa compileze cele doua biblioteci partajate cu numele de `LibMonitor.so` si `LibRW.so`. O biblioteca partajata se creaza folosind optiunea `-fPIC` la compilare pentru generarea de cod independent de pozitie si optiunea `-shared` la linkare. Pentru infomatii mai detaliate despre biblioteci consultati Program Library HOWTO.

Pentru a testa utilizarea corecta a monitorului mentineti un index in Thread Specific Data care indica pentru fiecare fir de executie daca se afla sau nu in interiorul monitorului. Nu e necesar sa alocati memorie, puteti folosi doar valoarea pointerului din indexul TSD.

Tema se va rezolva folosind doar functii POSIX. Se pot folosi de asemenea si functiile de formatare `printf, scanf`, functiile de alocare de memorie `malloc, free` si functiile de manipulare a sirurilor de caractere (`strcat`, `strdup`, etc.)

Tema se va rezolva folosind fire de executie POSIX si exclusiv mecanisme de sincronizare a firelor de executie POSIX.

Pentru partea de I/O si procese se vor folosi doar functii POSIX. De exemplu, functiile `fopen`, `fread`, `fwrite`, `fclose` nu trebuie folosite, in locul acestor trebuind sa folositi `open`,`read`, `write`, `close`.

Testare

Programul de test (linux, windows) foloseste functiile oferite de cele doua biblioteci pentru a implementa 3 teste:

- TestMonitor verifica functionarea corecta a monitorului
- TestRW verifica functionarea corecta a functiilor scriitorilor si cititorilor
- TestStres verifica functionarea scriitorilor si cititorilor in conditii de incarcare puternica

Pentru testarea celor doua biblioteci va trebui sa faceti voi apeluri catre niste functii care vor verifica corectitudinea starii monitorului si a bufferului de citire/scriere . Voi prezenta mai jos scopul acestor functii. Voi trebuie sa decideti, in functie de implementarea voastra, cand trebuie apelate.

- `void IncEnter();` - cand un thread intra in coada Entry Queue
- `void DecEnter();` - cand un thread paraseste coada Entry Queue
- `void IncSignal();` - cand un thread intra in coada Signaller Queue
- `void DecSignal();` - cand un thread paraseste coada Signaller Queue
- `void IncWait();` - cand un thread intra in coada Waiting Queue
- `void DecWait();` - cand un thread paraseste coada Waiting Queue
- `void IncCond(int nrCond);` - cand un thread incepe sa astepte la coada variabilei de conditie nrCond

- `void DecCond(int nrCond);` - cand un thread paraseste coada de asteptare la variabila de conditie nrCond
- `void AnnounceStartCit();` - cand un cititor incepe efectiv sa citeasca (deci NU cand intentioneaza sa citeasca, ci ulterior, cand incepe efectiv sa citeasca)
- `void AnnounceStopCit();` - cand un cititor se opreste efectiv din citit
- `void AnnounceStartScrit();` - cand un scriitor incepe efectiv sa scrie
- `void AnnounceStopScrit();` - cand un scriitor se opreste efectiv din scris

Antelele acestor functii se gasesc in CallbackMonitor.h, respectiv CallbackRW.h. In Constante.h gasiti constantele pentru politicile de functionare ale monitorului. Pentru a rula tema executati urmatoare secventa de comenzi, dupa ce ati dezarhivat testele in directorul cu sursele temei:

```
#compileaza testele, stage 1
make -f Makefile.checker build-pre &&
#compileaza tema
make build &&
#compileaza testele, stage 2
make -f Makefile.checker build-post &&
#ruleaza testele
make -f Makefile.checker
```

Notare

Primele 11 teste (TestMonitor.c si TestRW.c) vor fi notate cu 8.5 puncte fiecare (din 100). Testul 12 (TestStres.c) are 6.5 puncte.

Un test este considerat trecut numai daca toti pasii din care este format se termina cu succes.

Depunctari suplimentare:

-0.1 pentru makefile incorect

-0.2 pentru README necorespunzator

-0.2 pentru surse prost comentate

-0.4 neverificarea conditiilor de eroare sau/si neeliberarea de resurse

-0.1 diverse alte probleme constatate in implementare