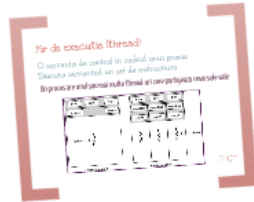


Fire de Executie

Sisteme de Operare, Curs 8

Thread-uri



Gata!

Hai Sa formam APA

Thread-urile reprezinta atomi de hidrogen sau oxigen

- O molecula de apa se formeaza din doi atomi de hidrogen si unul de oxigen
- Daca exista doi atomi de hidrogen, vor astepta un atom de oxigen
- Daca exista un atom de oxigen, va astepta doi atomi de hidrogen



Cum implementam un server de web?

Cerinta
Servirea cererilor clientilor (requests)
Incarcarea paginilor dintr-un server (HTTP)
Servirea cererilor clientilor (requests)
Incarcarea paginilor dintr-un server (HTTP)

Alte cerinte
Incarcarea paginilor dintr-un server (HTTP)
Servirea cererilor clientilor (requests)

Server de web

Operatii cu thread-uri

- Incarcarea in executie
- Incetarea executiei
- Terminare forzata (cancel)
- Asteptare (join)
- Retime

API

Sincronizare

Un thread poate sa execute o operatie doar daca este permis
Acest lucru este posibil prin intermediul
unor functii care controleaza accesul
la resurse comune

Sincronizare

Implementare Thread-uri

Un thread poate sa execute o operatie doar daca este permis
Acest lucru este posibil prin intermediul
unor functii care controleaza accesul
la resurse comune

Implementare

Cum implementam un Server de web?

Cerinte

Servește un număr arbitrar de clienți

Fiecare client poate cere oricate pagini (HTTP 1.1)

Mentine statistici: nr. total de pagini vizionate,
numarul total de clienți, octeti cititi, etc.

Alternative

Implementare Secventiala

```
while(1){  
    int i=accept();  
    handle_request(i);  
    while(true){  
        read_and_send_data(i);  
        update_stats();  
        handle_request(i);  
    }  
}
```

FoloSind Procese

```
while(1){  
    int i=accept();  
    fork();  
    handle_request(i);  
    read_and_send_data(i);  
    update_stats();  
    handle_request(i);  
    _exit(1);  
}
```

Implementare Asincrona

```
while(1){  
    int i=accept();  
    struct pollfd pfd;  
    pfd.fd=i;  
    pfd.events=POLLIN;  
    poll(&pfd, 1, -1);  
    read_and_send_data(i);  
    update_stats();  
    handle_request(i);  
}
```

Folositind Thread-uri

```
while(1){  
    int i=accept();  
    pthread_t t;  
    pthread_create(&t, NULL, handle_request, i);  
    pthread_join(t, NULL);  
    read_and_send_data(i);  
    update_stats();  
    handle_request(i);  
}
```

Implementare Secventiala

```
while (1){  
    int s = accept(ls);  
    fname = read_request(s);  
    while (fname){  
        read_and_send_file(fname);  
        update_stats();  
        fname = read_request(s);  
    }  
}
```

Probleme

Un singur client simultan
Ineficient chiar si cu un singur procesor

Probleme

Un singur client simultan
Ineficient chiar si cu un singur procesor

```
while (1){  
    int s = accept(ls);  
    fname = read_request(s);  
    while (fname){  
        read_and_send_file(fname);  
        update_stats();  
        fname = read_request(s);  
    }  
}
```

Probleme

Un singur client simultan
Ineficient chiar și cu un singur procesor

Alternative

Modelo di Processi

```
(1){  
    int s = accept(ls);  
    if (fork() == 0){  
        read_request(s);  
    }
```

Implementare ASincrona

```
while (1){  
    int s = accept(ls);  
    add_client(s);  
    select(...);  
    for (c:clients){  
        if (FD_ISSET(c.s)){  
            fname = read_request(s);  
        }    }
```

FoLoSiNd ProceSe

```
while (1){  
    int s = accept(ls);  
    if (fork()==0){  
        fname = read_request(s);  
        while (fname){  
            read_and_send_file(fname);  
            update_stats();  
            fname = read_request(s);  
        }  
    } else { ... }  
}
```

Probleme
Cum actualizam statistica?
Cum ream pentru fiecare
proces

Probleme

Cum actualizam statisticile?
Cost mare pentru pornire
proces

Implementare ASincrona

```
while (1){
    int s = accept(ls);
    add_client(s);
    select(...);
    for (c:clients){
        if (FD_ISSET(c.s)){
            fname = read_request(s);
            c.d = open(fname,...);
            c.status = read_file;
            //...
        } else if (FD_ISSET(c.d)){
            read(c.d, buf, 1000);
            send(c.s,buf,1000);
            ...
        }
    }
}
```

Probleme

Trebuie sa tinem stare pentru fiecare client
Greu de implementat

Probleme

Trebuie sa tinem stare pentru fiecare client
Greu de implementat

Am dori o primitiva SO care:

Executa secvential un set de intructiuni

Este usor de pornit / oprit

Partajeaza date cu usurinta

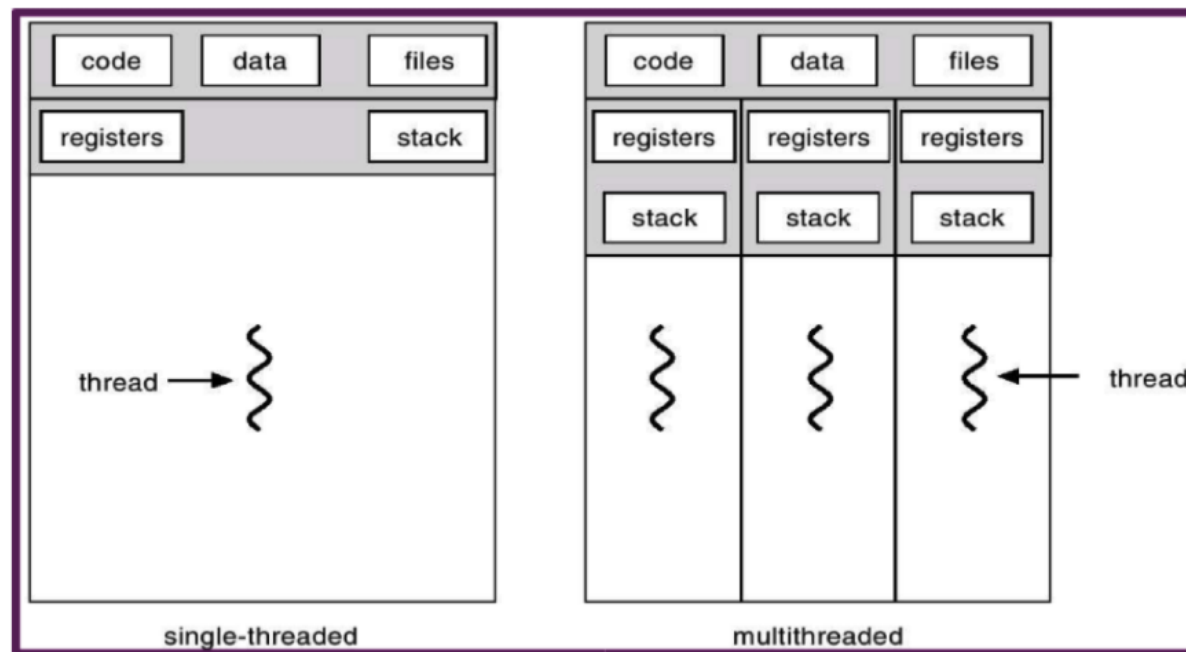
FoLoSind Thread-uri

```
while (1){  
    int s = accept(ls);  
    pthread_create (&t, NULL, (void *) &cnt, (void *) &s);  
}  
  
...  
void* clnt(void* p){  
    int s = *(int*)p;  
    char* fname = read_request(s);  
    while (fname){  
        read_and_send_file(fname);  
        update_stats();  
        fname = read_request(s);  
    }  
}
```

Fir de executie (thread)

○ secventa de control în cadrul unui proces
Executa secvential un set de instructiuni

Un proces are unul sau mai multe thread-uri care partajeaza resursele sale



Ce partajeaza thread-urile?

variabilele globale (.data, .bss)

fisierele deschise

spatiul de adresa

masca de semnale

Ce NU partajeaza thread-urile?

registrele

stiva

program counter/Instruction pointer

stare

TLS (Thread Local Storage)

Procese

vs.

thread-uri

Grupeaza resurse

Fisiere, lucru retea

Spatiu adrese

Fire de executie



Abstractizeaza executia

Stiva

Registri

Program Counter

Source: [https://www.geogebra.org/m/...](#)

Source: [https://www.geogebra.org/m/...](#)

Avantaje thread-uri

Timp de creare mai mic decat al proceselor

Timp mai mic de schimbare context

Partajare facila de informatie

Utile chiar si pe uniprocessor

Dezavantaje thread-uri

Daca moare un thread, moare tot procesul

Nu exista protectie la partajarea datelor

Probleme de sincronizare

Prea multe thread-uri afecteaza performanta!

Operatii cu thread-uri

- Lansarea in executie
- Incetarea executiei
- Terminare fortata (cancel)
- Asteptare (join)
- Planificare

Posix Threads

```
Posix Threads
Start program(s)
All parts covered in previous thread(s)
- pthread_t
- pthread_create()
- pthread_join()
```

```
API Threads
pthread_t
pthread_create()
pthread_join()
```

Linux

```
Threads in Linux
pthread_t
pthread_create()
pthread_join()
```

Windows

```
Threads in Windows
HANDLE
CreateThread()
```

Posix Threads

Folosit pe sistemele Unix

API pentru crearea si sincronizarea thread-urilor

Folosire

- inclus header-ul (`#include <pthread.h>`)
- legarea bibliotecii (`-lpthread`)
- `man 7 pthreads`

API PThreads

```
pthread_t tid;
```

```
pthread_create(&tid, NULL, threadfunc, (void*)arg);
```

```
pthread_exit(void* ret);
```

```
pthread_join(pthread_t tid, void** ret);
```

```
pthread_cancel(pthread_t tid);
```

Thread-uri in Linux

Suport in kernel pentru task-uri (struct task_struct)
Procesele si thread-urile sunt task-uri
planificabile independent

NPTL (New Posix Thread Library)

- implementare pthreads (1:1)
- foloseste apelul de sistem clone
- thread-urile sunt grupate in acelasi grup
- getpid intoarce thread group id

clone

Specific Linux

Folosit de fork si NPTL

Diferite flag-uri specifica resursele partajate

- CLONE_NEWNS
- CLONE_FS, CLONE_VM, CLONE_FILES
- CLONE_SIGHAND, CLONE_THREAD

Thread-uri in Windows

Model hibrid: suport in kernel

Fibre: fire de executie in user-mode

- planificate cooperativ
- blocarea unei fibre blocheaza firul de executie

API Windows

HANDLE CreateThread(...)

ExitThread

WaitForSingleObject / MultipleObjects

GetExitCodeThread

TerminateThread

TlsAlloc

TlsGetValue/TlsSetValue

Implementare thread-uri

User-level

O biblioteca de thread-uri ofera suport pentru crearea, planificarea si terminarea thread-urilor

Mentine o tabela cu fire de executie:

PC, registre, stare pentru fiecare fir

Nucleul "vede" doar procese, nu si thread-uri

Mai multe fire de executie sunt planificate cu un sigur proces

Avantaje

• Nu este nevoie de suport din partea sistemului de operare pentru a utiliza thread-uri

Dezavantaje

• Nu este posibil să se utilizeze thread-uri în aplicații care necesită sincronizare

Kernel

Suport in kernel pentru creare, terminare si planificare
Model unu-la-unu

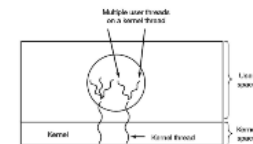
Avantaje

Fara probleme la apeluri blocante sau page faults
Pot fi planificate pe sisteme multiprocesor

Dezavantaje

Crearea si schimbarea de context este mai lenta

Hibrid



User-level

O biblioteca de thread-uri ofera suport pentru crearea, planificarea si terminarea thread-urilor

Mentine o tabela cu fire de executie:

PC, registre, stare pentru fiecare fir

Nucleul "vede" doar procese, nu si thread-uri

Mai multe fire de executie sunt planificate cu un sigur proces



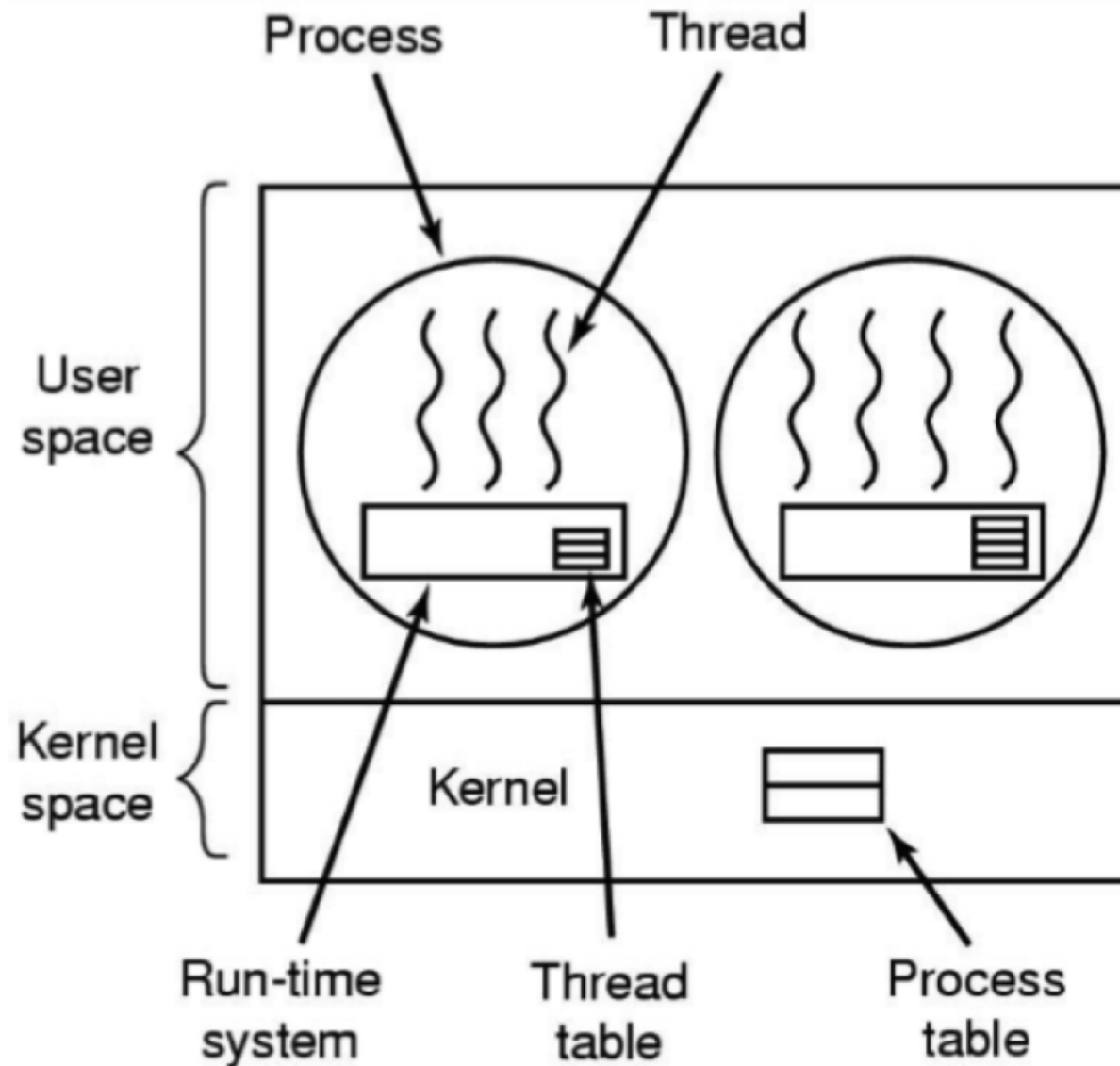
Avantaje

Un apel de sistem nu blocheaza procesul

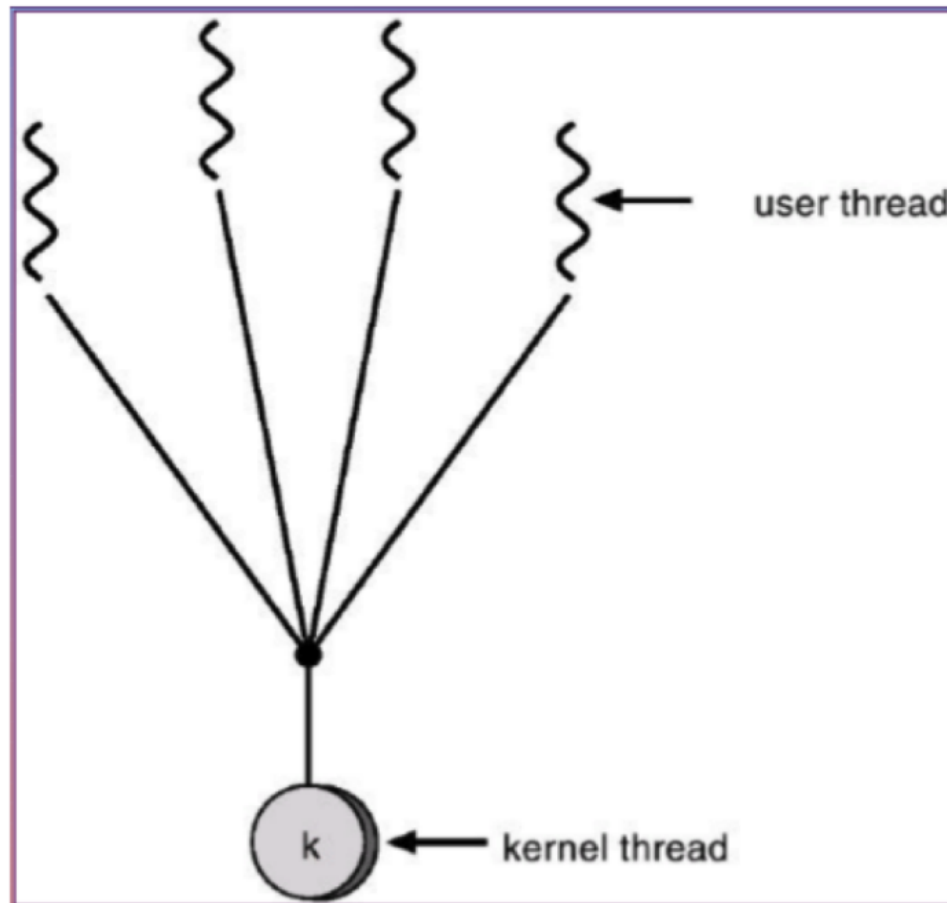
Dezavantaje

Un apel de sistem blocheaza intreg procesul

Thread-variant implementate user-level



Mai multe fire de executie sunt mapate pe acelasi fir de executie din kernel



Avantaje

Usor de integrat în SO: nu sunt necesare modificari

Pot oferi suport multithreaded pe un SO fara suport multithreaded

Schimbare de context rapida: nu se executa apeluri de sistem în nucleu

Aplicatiile pot implementa planificatoare în functie de necesitati

Dezavantaje


Un apel de sistem blocant blocheaza întreg procesul:
cum rezolvam?

Un page-fault blocheaza tot procesul

Planificare cooperativa

Multe aplicatii folosesc apeluri de sistem oricum

Kernel

Suport in kernel pentru creare, terminare si planificare
Model unu-la-unu 

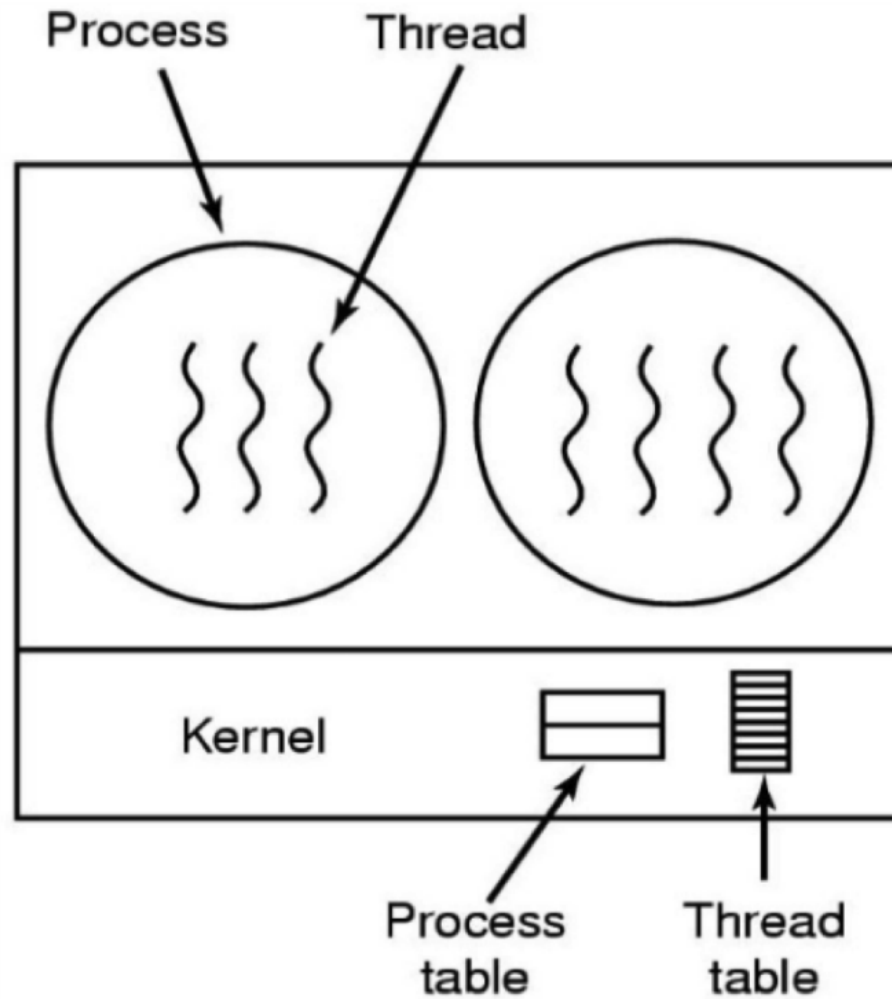
Avantaje

Fara probleme la apeluri blocante sau page faults
Pot fi planificate pe sisteme multiprocesor

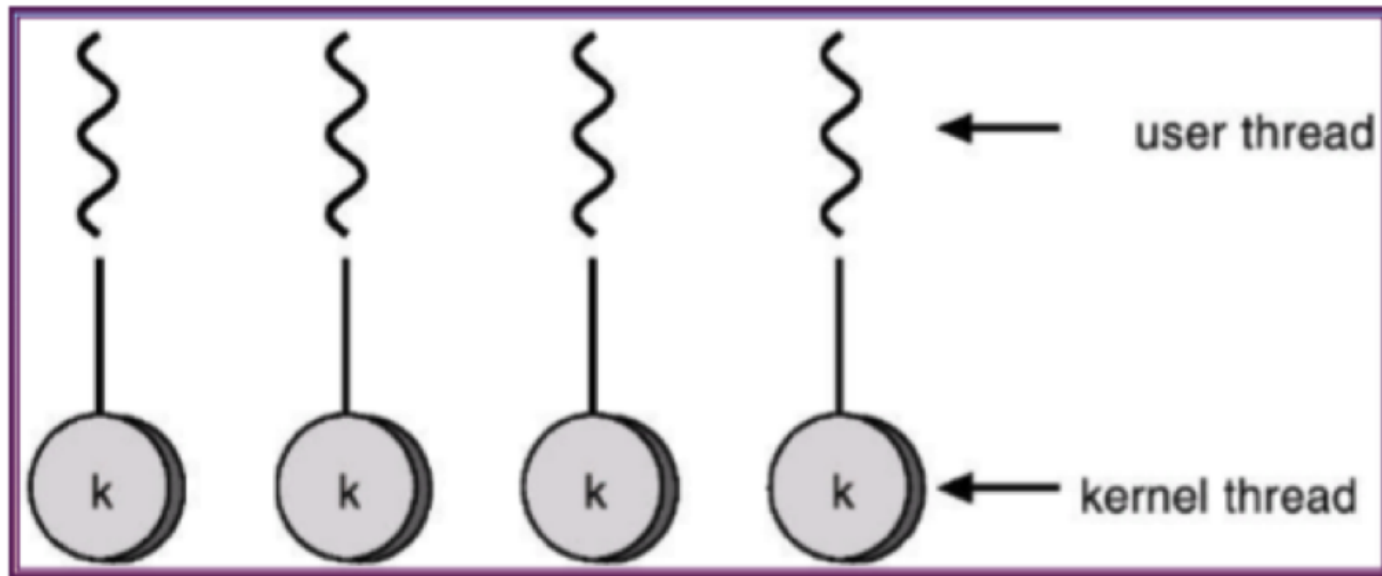
Dezavantaje

Crearea si schimbarea de context este mai lenta

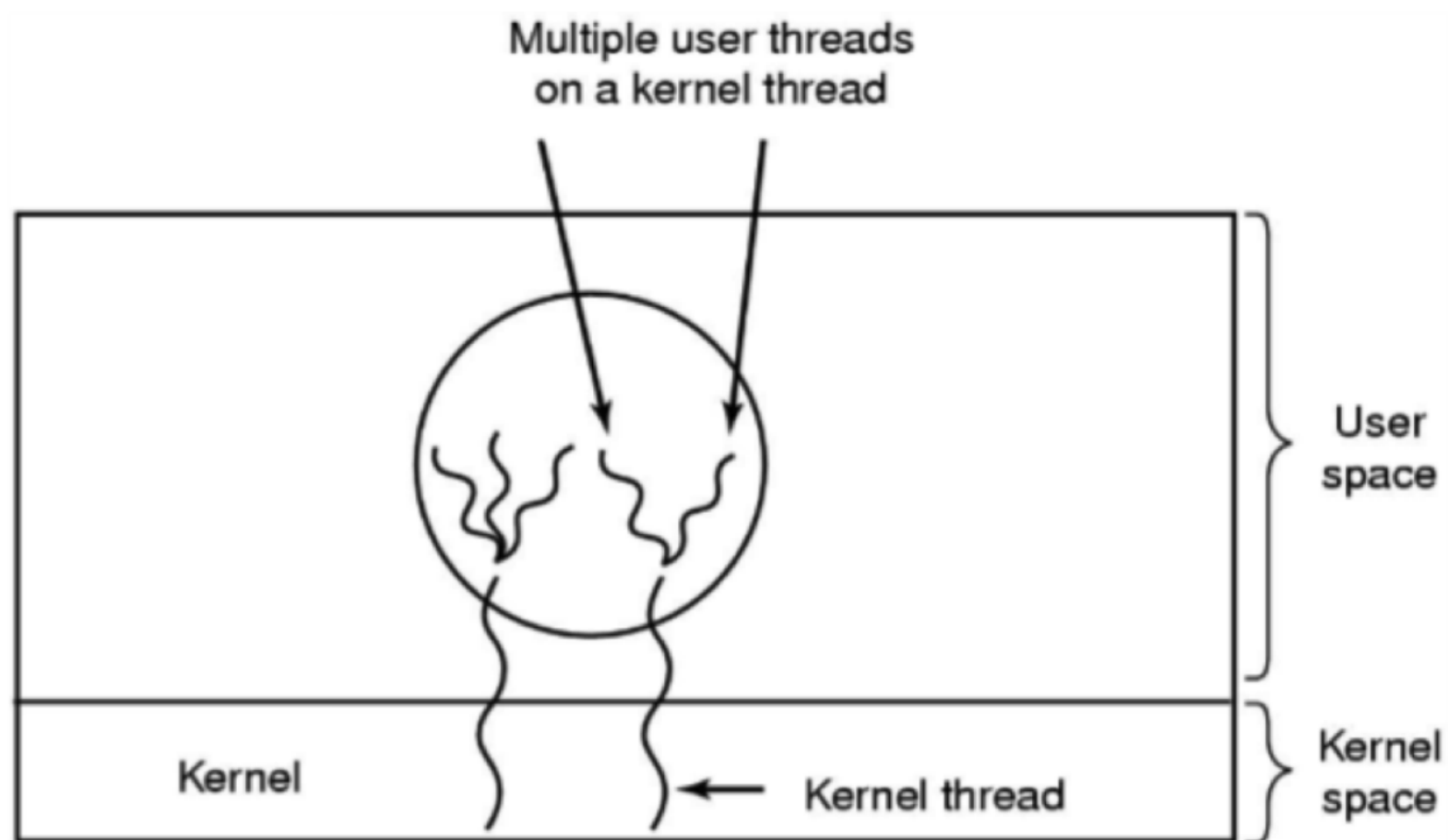
Implementare thread-uri in kernel



Un kernel thread pentru fiecare thread utilizator



Hibrid



Sincronizare

Thread-urile ofera acces la date comune

Accesul trebuie mediat pentru a implementa programe corecte

Chiar si programele cu un singur thread pot crea probleme!

Dar cele cu mai multe thread-uri?

Exemple

```
void foo() {  
    int x = 0;  
    while (x < 10) {  
        x = x + 1;  
    }  
}
```

Reentranta

Funcțiile care pot fi executate din nou și din nou fără a necesita modificări în codul sursă.

Sincronizare în practică

Multe funcții de bibliotecă sunt sincronizabile prin...

Thread safety
O funcție este thread-safe dacă poate fi apelată din mai multe thread-uri în același timp.

Thread synchronization

- mutex
- semaphore
- thread-local storage
- read-write lock
- atomic operations

Exemplu

```
int total_bytes;  
void update_statics(int j){  
    total_bytes += j;  
}
```

```
void signal_handler(){  
    update_statistics(1);  
}
```

Reentrant

O functie este reentranta daca poate fi executata simultan de mai multe ori, fara a afecta rezultatul

Conditii necesare:

- nu lucreaza cu variabile globale/statice
- apeleaza doar functii reentrante

Reentranta este importanta (mai ales) in programe cu un singur thread din cauza semnalelor!

Reentranta in practica

Multe functii de biblioteca seteaza variabila errno
Sunt acestea reentrante?

Depinde de implementare!

Anumite apeluri au versiuni reentrante: gethostbyname_r
Activare cu macroul _REENTRANT

Depinde de implementare!

Anumite apeluri au versiuni reentrante: `gethostbyname_r`
Activare cu macroul `_REENTRANT`

Thread safety

○ functie este thread-safe daca poate fi apelata din mai multe thread-uri in acelasi timp

Strategii implementare

- acces exclusiv
- semafoare
- monitoare
- thread-local storage
- reentranta
- operatii atomice



Access Exclusiv

Posix

pthread_mutex_init
pthread_mutex_destroy
pthread_mutex_lock
pthread_mutex_unlock

Win32 API

Create/OpenMutex
CloseHandle
ReleaseMutex
WaitForSingleObject
Initialize/DeleteCriticalSection
Enter/TryEnter/LeaveCriticalSection

Semafoare

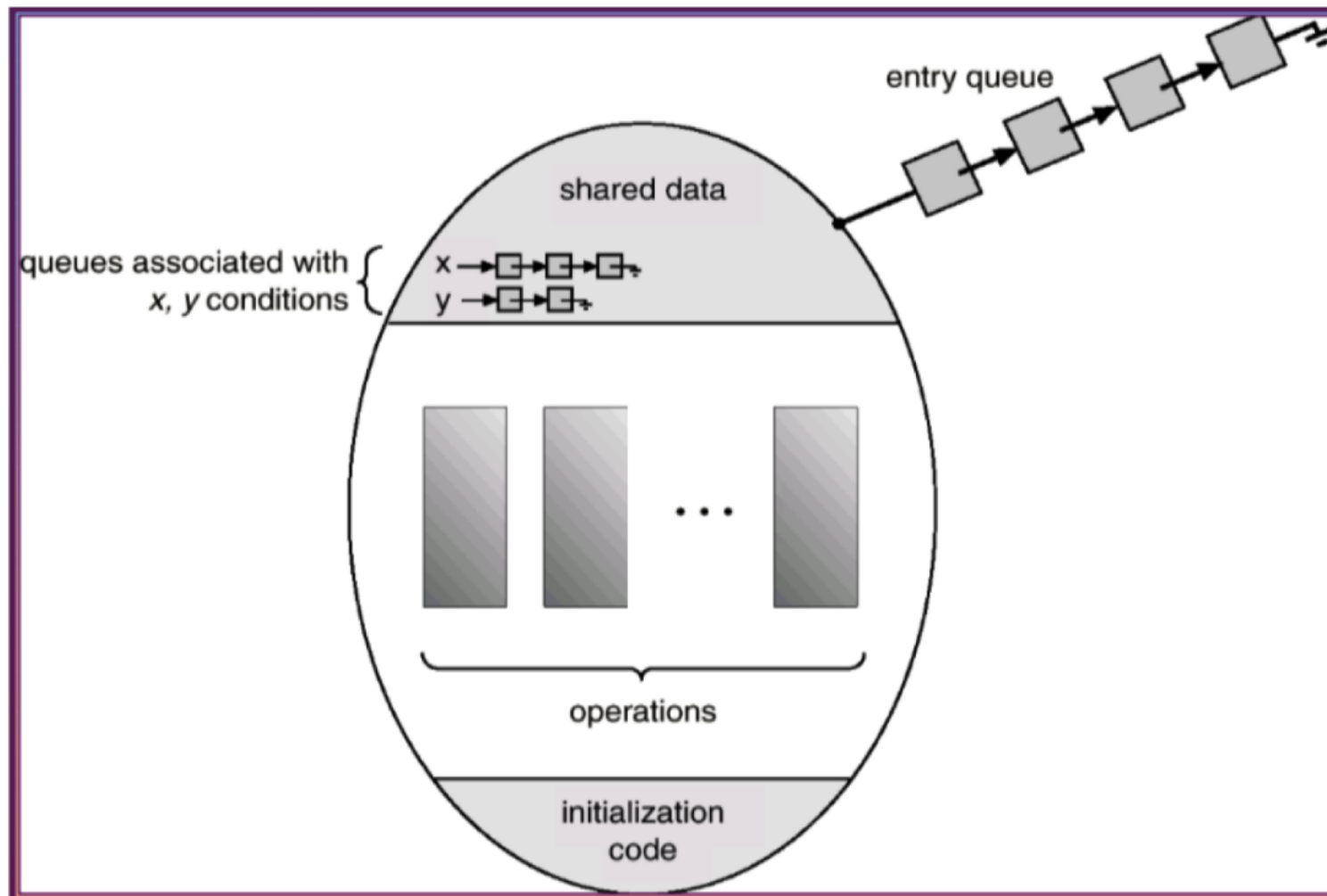
POSIX

sem_t sem
sem_init
sem_destroy
sem_wait
sem_trywait
sem_post

Win32

HANDLE hSem
CreateSemaphore
CloseHandle
WaitForSingleObject
ReleaseSemaphore

Monitor



Operatii Monitor

Intrare: m.entry()

Iesire: m.leave()

Semnalizare: m.signal(cond)

Asteptare: m.wait(cond)

Cozi de asteptare:

- pentru fiecare variabila conditie
- pentru intrare in monitor

Functionare Monitor

Un singur thread ruleaza in monitor la un moment dat

La un apel wait:

- thread-ul se blocheaza
- iese din monitor, trece in coada de asteptare specifica

Politici de planificare:

- signal and wait
- signal and continue

Hai Sa formam APA

Thread-urile reprezinta atomi de hidrogen sau oxigen

- O moleculă de apă se formează din doi atomi de hidrogen și unul de oxigen
- Dacă există doi atomi de hidrogen, vor aștepta un atom de oxigen
- Dacă există un atom de oxigen, va aștepta doi atomi de hidrogen

[illegible]

Națiunile trebuie să se dăruiească mai mult.

[illegible]

Oxide chole

protein- thrombin and rennin participate in fibrinolysis.	factor is activated by thrombin.
plasminogen plasminogen plasminogen plasminogen	plasminogen plasminogen plasminogen plasminogen

Implementare cu Semafoare

```
Semaphore hsem, osem;
```

```
void hydrogen() {  
    up(hsem);  
    down(osem);  
    bond();  
}
```

```
void oxygen(){  
    down(mutex);  
    down(hsem);  
    down(hsem);  
    up(osem);  
    up(osem);  
    up(mutex);  
    bond();  
}
```

Se poate implementa mai ușor?

Observatii:

- excluderea mutuala ne trebuie cam tot timpul
- la fel si nevoie de a comunica intre thread-uri

Monitoarele combina aceste doua primitive in mod elegant

Apa: Implementare cu monitoare

Monitor m;

cond m.oxy_cond, m.hydro_cond;

```
void oxygen(){
    m.enter();
    o_count++;
    if (h_count >= 2){
        o_count--;
        m.hydro_cond.signal();
        m.hydro_cond.signal();
        h_count -= 2;
    } else oxy_cond.wait();
    m.leave();
    bond();
}
```

```
void hydrogen(){
    m.enter();
    h_count++;

    if(h_count == 2 && o_count >= 1){
        m.hydro_cond.signal();
        h_count -= 2;
        m.oxy_cond.signal();
        o_count--;
    } else m.hydro_cond.wait().
    m.leave();
    bond();
}
```

Cuvinte cheie

procese

thread-uri

resurse partajate

POSIX Threads

NPTL

clone

user-level threads

kernel-level threads

hybrid threads

reentranta

thread safety

semafoare

monitoare

problema formarii apei