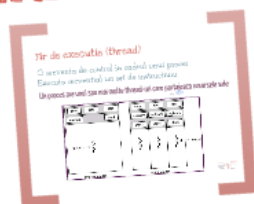


# Fire de Executie

Sisteme de Operare, Curs 8

Gata!

## Thread-uri



Server de Web



API



Implementare



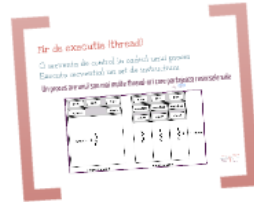
Sincronizare

# Fire de Executie

Sisteme de Operare, Curs 8

Gata!

## Thread-uri



Server de web



API



Implementare



Sincronizare

# Cum implementam un Server de web?

## Cerinte

Serveste un numar arbitrar de clienti

Fiecare client poate cere oricate pagini (HTTP 1.1)

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

## Alternative

### Implementare Sequentiale

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

### Folosiind Procesa

```
while (1) {  
    int = accept();  
    if (fork() == 0) {  
        handle_request();  
        while (handle())  
            read_and_send_data();  
        update_stats();  
        handle_request();  
    }  
}
```

### Implementare Asincrona

```
while (1) {  
    int = accept();  
    if (fork() == 0) {  
        handle_request();  
        while (handle())  
            read_and_send_data();  
        update_stats();  
        handle_request();  
    }  
}
```

### Folosiind Thread-uri

```
while (1) {  
    int = accept();  
    pthread_create(&tid, NULL, handle_request, (void *)tid);  
}
```

# 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

```
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

# Alternative

Processi

```
while(1){  
    int s = accept(ls);  
    if(s < 0){  
        continue;  
    }  
    char name[1024];  
    read(s, name, 1024);  
    // ...  
}
```

Implementare ASincrona

```
while(1){  
    int s = accept(ls);  
    add_client(s);  
    select(...);  
    for(c:clients){  
        if(FD_ISSET(c.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 statisticele?  
Cum scanez pentru procese  
procese



# 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

Am dori o primitiva SO care:

Executa secvential un set de instructiuni

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);
}

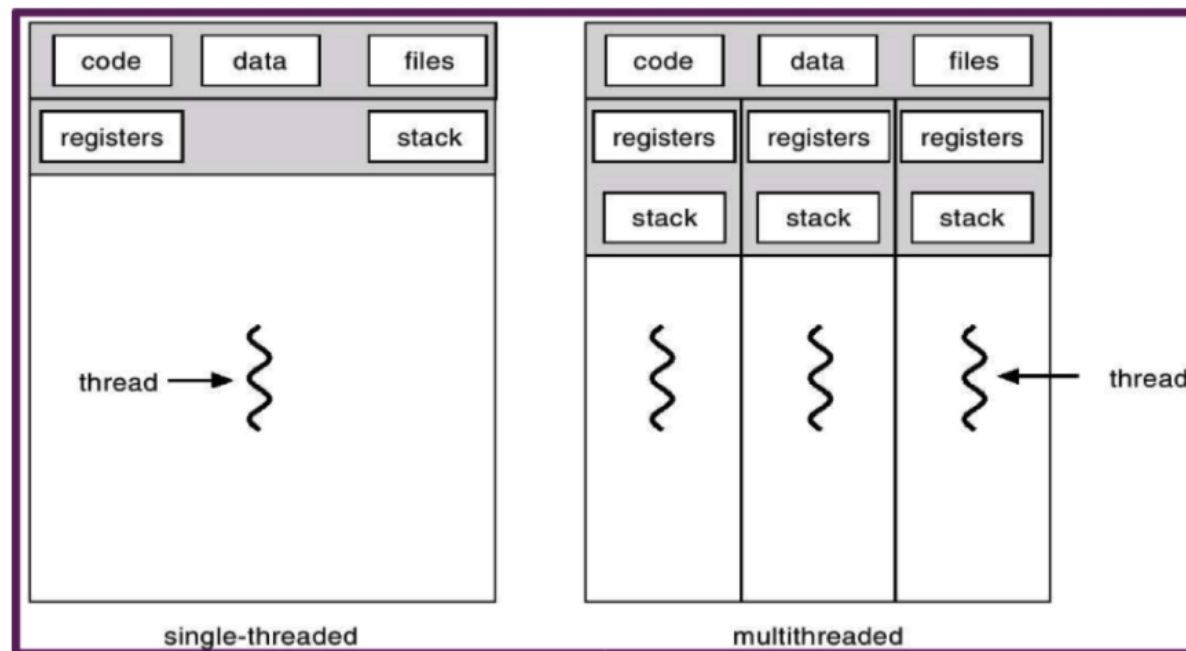
...

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



Proces vs. thread-uri  
Gruparea resurselor  
Fișiere, memorie  
Spațiu adresă  
Fie de execuție

vs.

Unități de execuție independente  
Stare  
Registri  
Program Counter

# Ce partajeaza thread-urile?

variabilele globale (.data, .bss)

fisierele deschise

spatiul de adresa

modul de tratare a semnalelor

# Ce NU partajeaza thread-urile?

registrele

stiva

program counter/Instruction pointer

stare

masca de semnale

TLS (Thread Local Storage)



# Procese

vs.

# thread-uri

Grupeaza resurse

Fisiere, lucru retea

Spatiu adrese

Fire de executie



Abstractizeaza executia

Stiva

Registri

Program Counter

Prezi

Prezi

# 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
#include <pthread.h>
pthread_t thread;
pthread_create(&thread, NULL, func, NULL);
```

```
AT Tutorials
pthread_t
pthread_create(pthread_t*, const pthread_attr_t*, void* (*)(void*), void*)
pthread_join(pthread_t*, void*)
pthread_cancel(pthread_t*)
```

## Linux

```
Threads in Linux
#include <pthread.h>
pthread_t thread;
pthread_create(&thread, NULL, func, NULL);
```

## Windows

```
Threads in Windows
#include <windows.h>
HANDLE thread;
CreateThread(NULL, 0, func, NULL, 0, NULL);
```

# 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 singur proces

### Avantaje

• Nu necesita o schimbare a kernelului  
• Nu necesita o schimbare a sistemului de operare  
• Nu necesita o schimbare a sistemului de operare  
• Nu necesita o schimbare a sistemului de operare

### Dezavantaje

• Nu poate fi folosit pentru a crea thread-uri  
• Nu poate fi folosit pentru a crea thread-uri  
• Nu poate fi folosit pentru a crea thread-uri  
• Nu poate fi folosit pentru a crea thread-uri

## Kernel

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

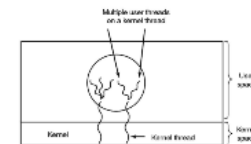
### Avantaje

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:

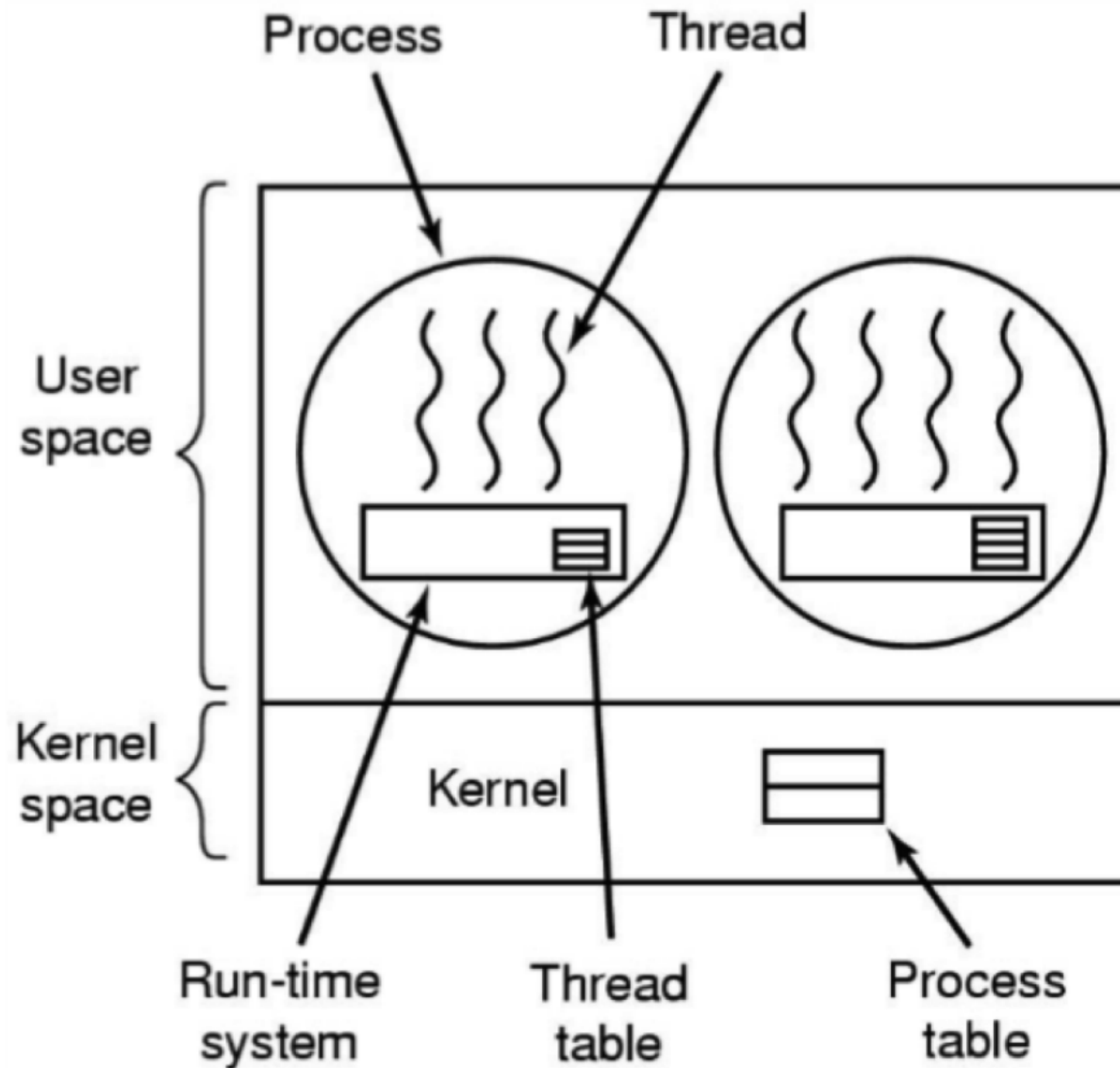
PC, registre, stare pentru fiecare fir

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

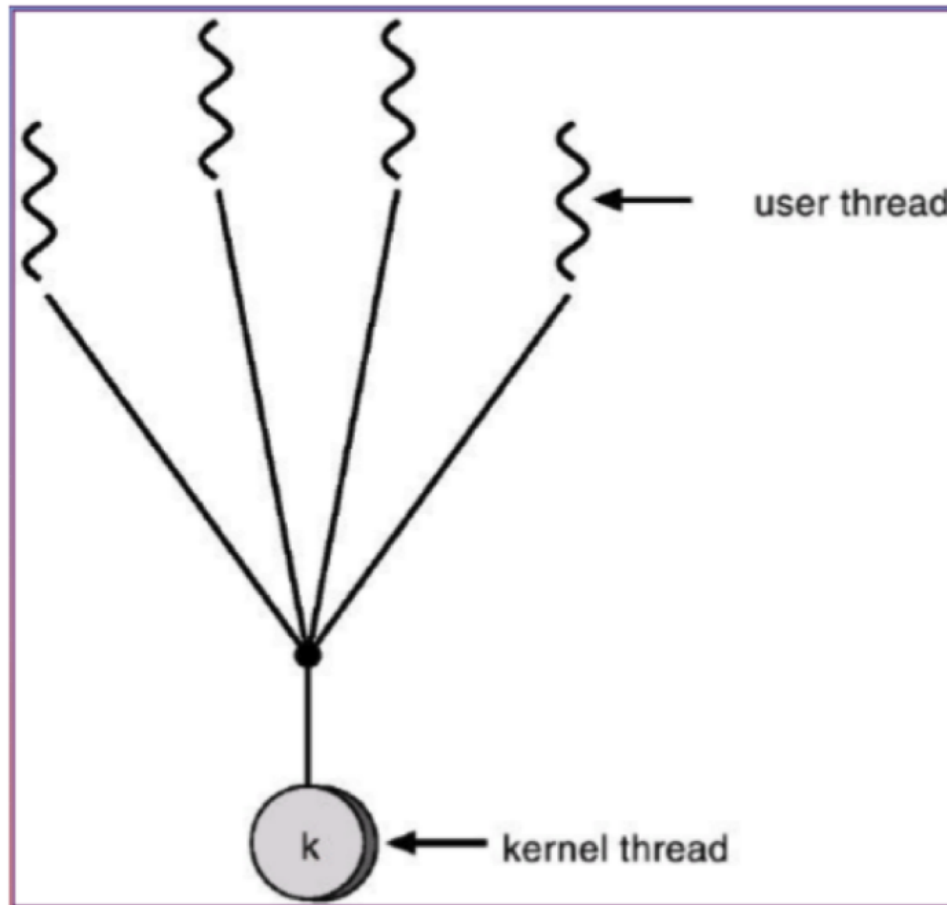
Mai multe fire de executie sunt planificate cu un singur proces



## Thread-uri implementate user-level



Mai multe fire de executie sunt mapate pe același  
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



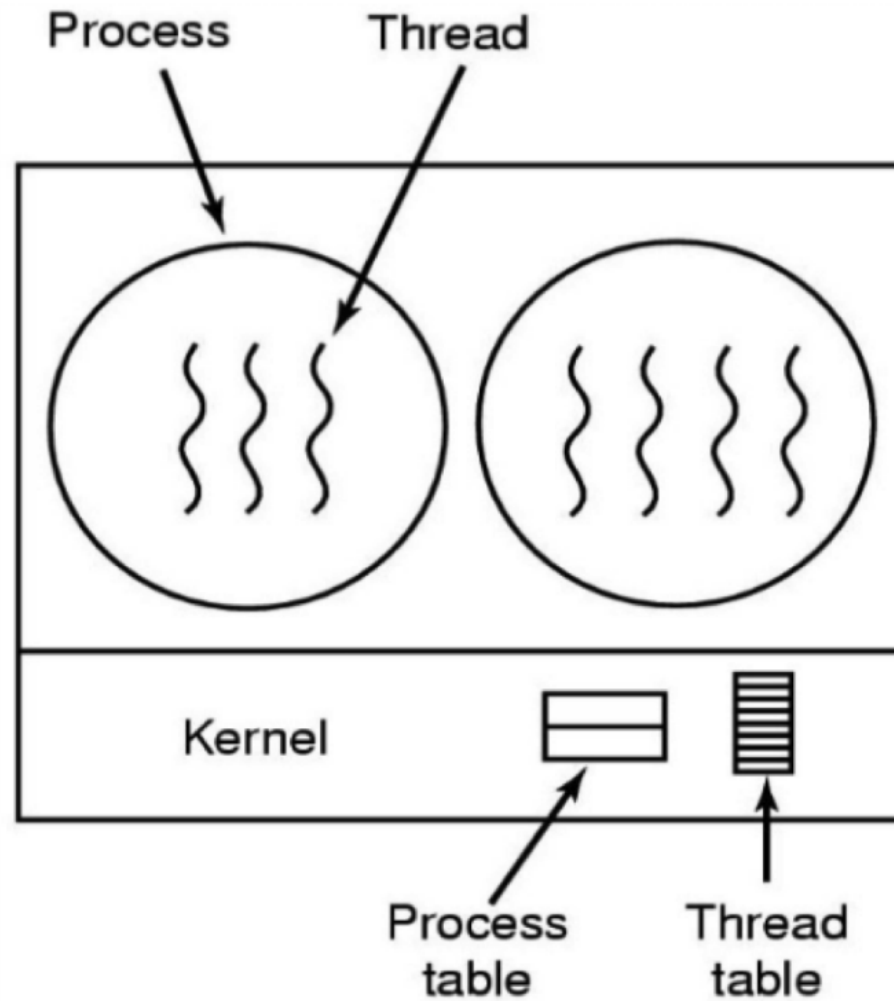
## Avantaje

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

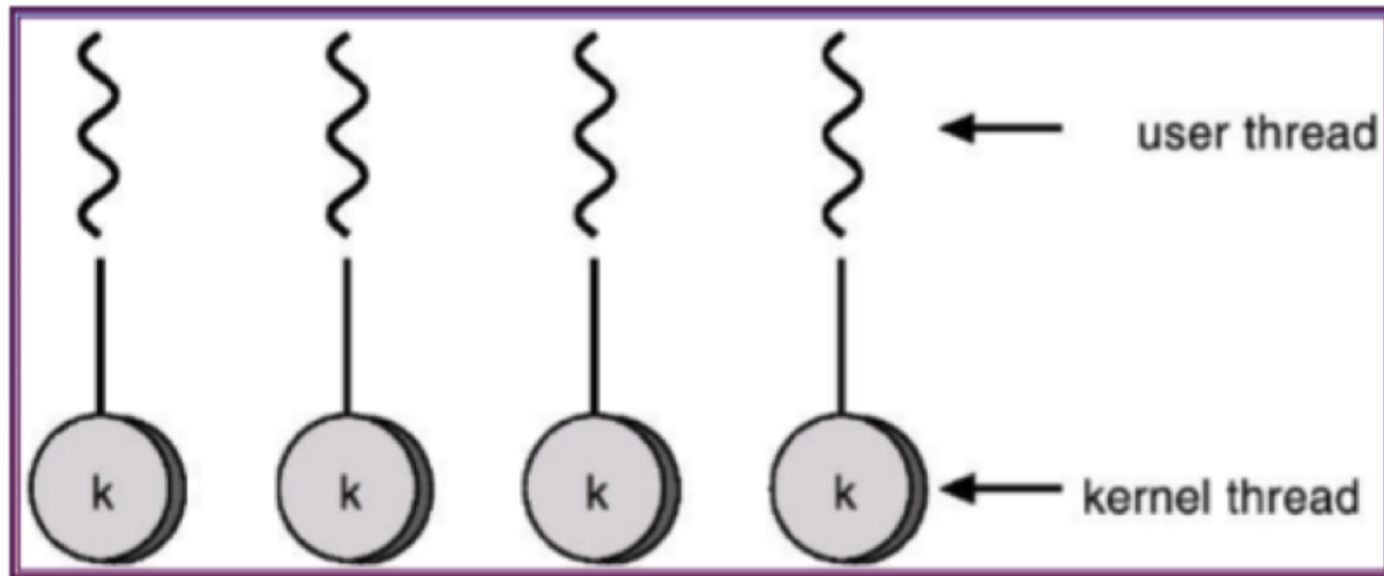
## Dezavantaje

Crearea si schimbarea de context este mai lenta

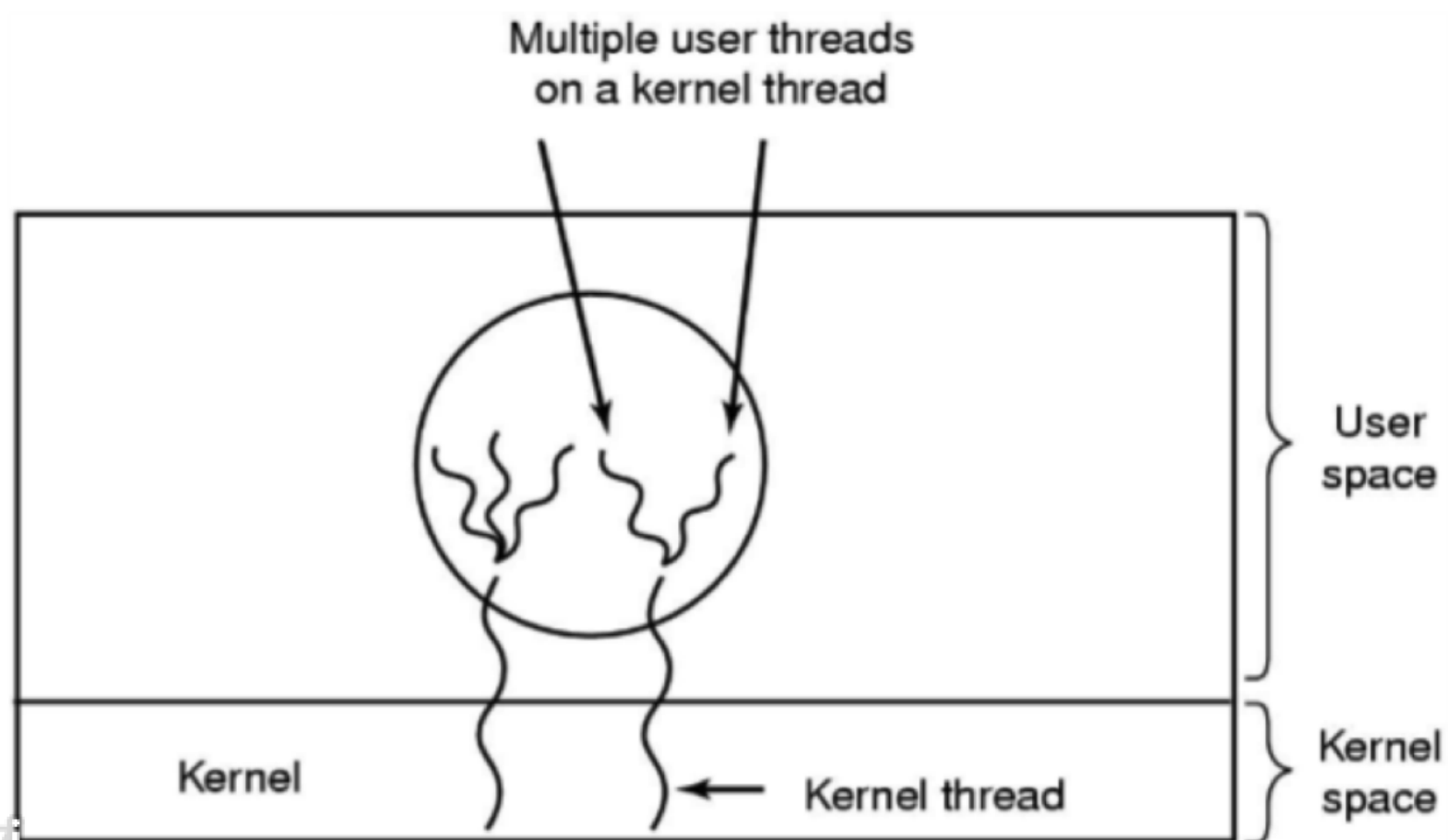
## Implementare thread-uri in kernel



Un kernel thread pentru fiecare thread utilizator



# Hybrid



# 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?

**Exemplu**  
if (x < 0) {  
 x = 0;  
}  
if (x < 0) {  
 x = 0;  
}

**Reentranta**  
O functie este reentranta daca  
poate fi apelata din nou inainte  
sa se termine executia ei.  
O functie este reentranta daca  
nu are variabile locale sau  
variabile statice.

**Reentranta in practica**  
Multe functii de biblioteca sunt reentrante.  
Sunt acestea reentrante? [Exemplu](#)

**Thread safety**  
O functie este thread-safe daca poate  
fi apelata din mai multe thread-uri in  
acelasi timp.  
Strategii de implementare (pentru variabile):  
- mutex  
- semafori  
- read-write lock  
- read-only lock  
- read-only lock

# Exemplu

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

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

# Reentranta

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 macro-ul \_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 (cursul urmator)

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