

Fire de Executie


Sisteme de Operare, Curs 8

Gata!

Thread-uri

Tir de executie (thread)

Un servet de control in codul unei aplicatii
Executie independenta in alii de main() si main()
Un proces (proces) poate avea multe thread-uri care pot executa instructiuni sale



Cum implementam un server de web?

Crearea
Servetilor care vor fi executati
Fiecare client va avea un servet (thread)
Monitorizarea in total de pagina request
Monitorizarea cu thread, unde este ok

Alte optiuni



Server de web


Operatii cu thread-uri

- Lansarea in executie
- Incheierea executiei
- Terminarea forzata (force)
- Asteptare (wait)
- Planificarea



API

Implementare Thread-uri



Implementare

Sincronizare

Un thread poate avea acces la un resursa
Alte thread-uri pot avea acces la acea resursa
Daca un thread are acces la o resursa, alte thread-uri
pot avea acces la acea resursa
De ce? In acest caz, trebuie sa se sincronizeze



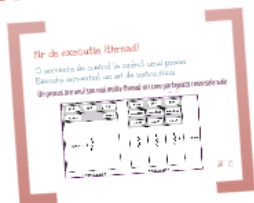
Sincronizare

Fire de Executie

Sisteme de Operare, Curs 8

Gata!

Thread-uri



Server de web



API



Implementare



Sincronizare

Cum implementam un server de web?

Cerinte

Servește un număr arbitrar de clienți
Fiecare client poate cere oricâte pagini (HTTP 1.1)
Mentine statistici: nr. total de pagini vizionate,
numărul total de clienți, octeți citiți, etc.

Alternative

Implementare Secventiala

```
while (1)
  int = accept();
  handle_request();
  while (handle())
    read_and_send_file(handle);
  handle_request();
}
```

Folosind Procese

```
while (1)
  int = accept();
  handle_request();
  while (handle())
    read_and_send_file(handle);
  handle_request();
}
```

Implementare ASincrona

```
while (1)
  int = accept();
  handle_request();
  while (handle())
    read_and_send_file(handle);
  handle_request();
}
```

Folosind Thread-uri

```
while (1)
  int = accept();
  handle_request();
  while (handle())
    read_and_send_file(handle);
  handle_request();
}
```

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 si cu un singur procesor

Alternative

Process

```
while(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)){  
            ...  
        }    }
```

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 sciam 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

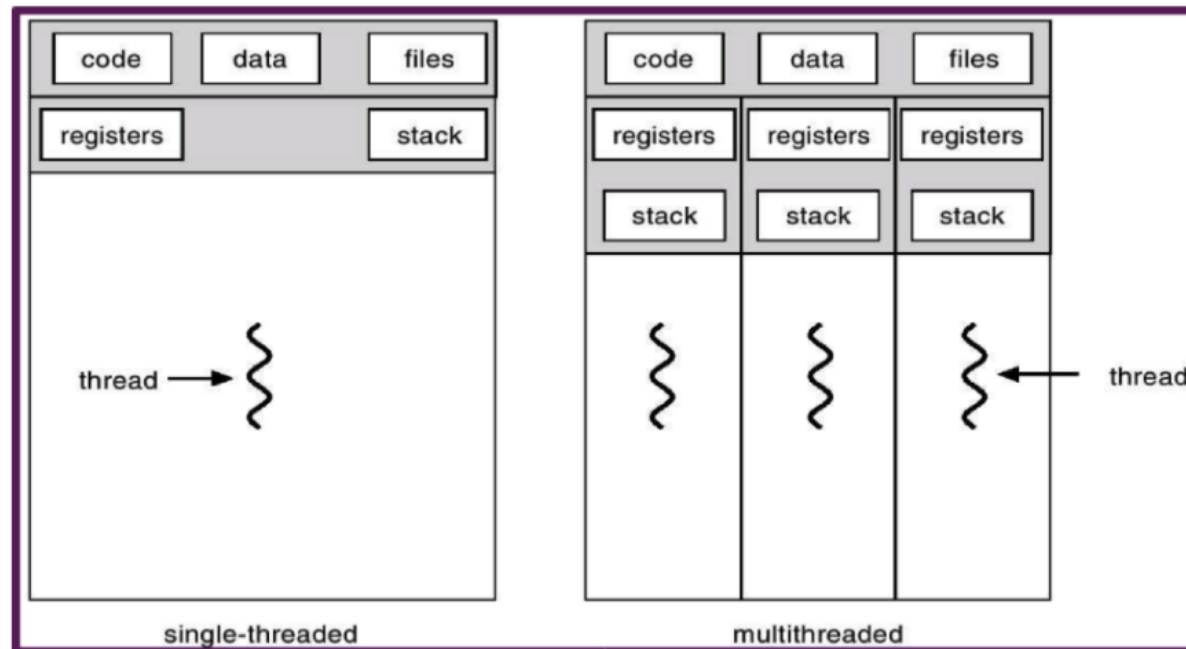
Folosind Thread-uri

```
while (1){
    int s = accept(ls);
    pthread_create (&t, NULL, (void *) &cnt, (void *) &s);
}
...
void* cnt(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



Procese vs thread-uri
Curs de Informatică
Scrieți aici
Prezentare

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

© 2014 Prezi Inc. All rights reserved.

© 2014 Prezi Inc. All rights reserved.

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 uniprosesor

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

```
pthread_t  
pthread_attr_t  
pthread_mutex_t  
pthread_mutexattr_t  
pthread_cond_t  
pthread_condattr_t  
pthread_key_t  
pthread_once_t
```

```
pthread_create  
pthread_join  
pthread_detach  
pthread_cancel  
pthread_kill  
pthread_self  
pthread_equal  
pthread_mutex_lock  
pthread_mutex_unlock  
pthread_mutex_trylock  
pthread_mutexattr_settype  
pthread_mutexattr_setkind  
pthread_mutexattr_setprotocol  
pthread_mutexattr_setpshared  
pthread_mutexattr_setrobust  
pthread_mutexattr_setscope  
pthread_mutexattr_t::e_t  
pthread_mutexattr_t::k_t  
pthread_mutexattr_t::p_t  
pthread_mutexattr_t::ps_t  
pthread_mutexattr_t::r_t  
pthread_mutexattr_t::s_t  
pthread_mutexattr_t::sc_t
```

Linux

```
pthread_t  
pthread_attr_t  
pthread_mutex_t  
pthread_mutexattr_t  
pthread_cond_t  
pthread_condattr_t  
pthread_key_t  
pthread_once_t
```

Windows

```
pthread_t  
pthread_attr_t  
pthread_mutex_t  
pthread_mutexattr_t  
pthread_cond_t  
pthread_condattr_t  
pthread_key_t  
pthread_once_t
```

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
• Este mai usor de implementat

Dezavantaje

• Nu sunt thread-uri care pot fi executate
• Nu sunt thread-uri care pot fi executate
• Nu sunt thread-uri care pot fi executate

Kernel

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

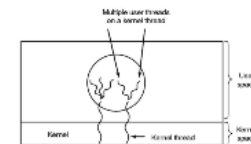
Avantaje

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



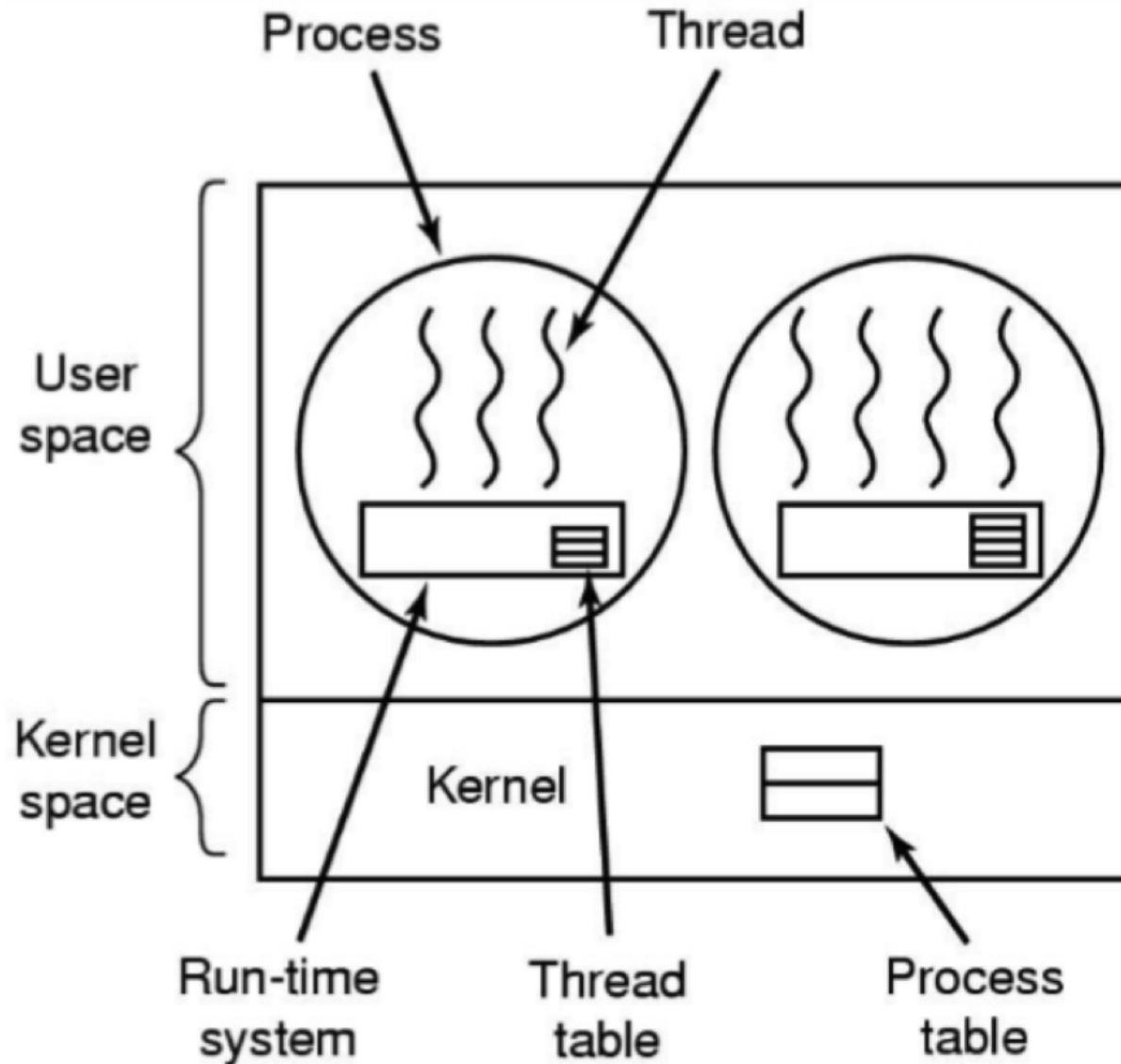
Avantaje

Usor de integrat în SO: nu sunt necesare modificari

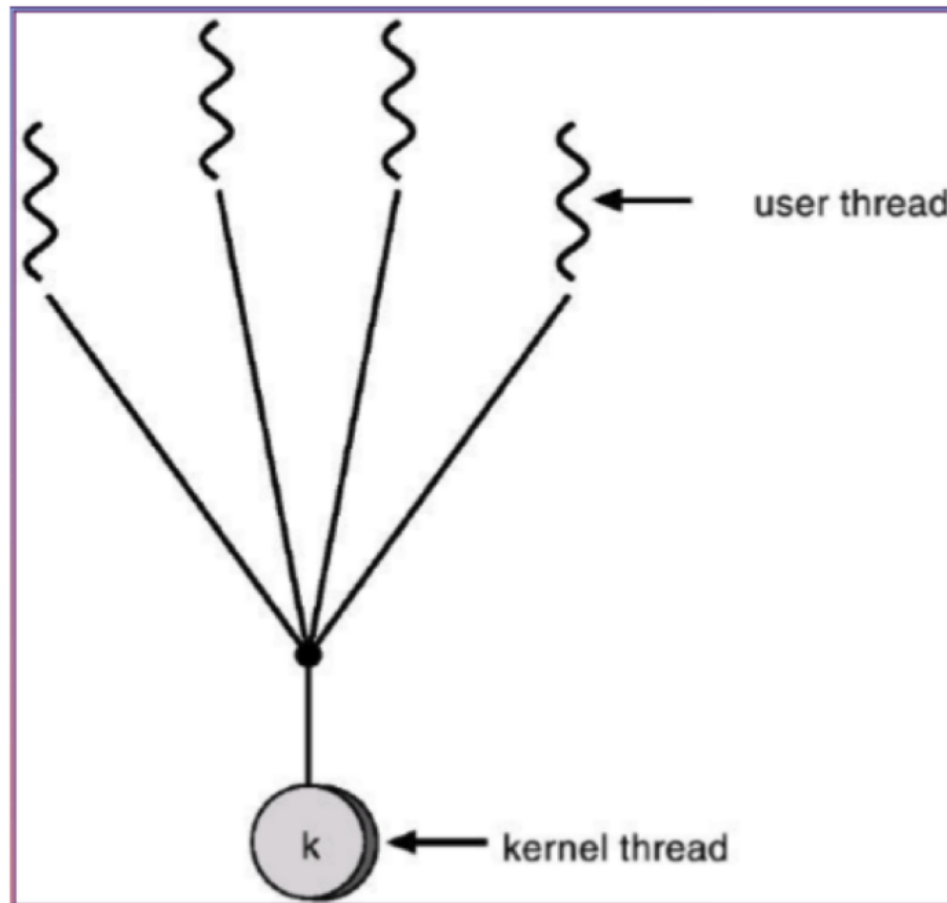
Dezavantaje

Un apel de sistem blocant blocheaza întreg procesul

Thread-ur 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

Un page-fault blocheaza tot procesul

Planificare cooperativa

Multe aplicatii folosesc apeluri de sistem oricand

Kernel

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

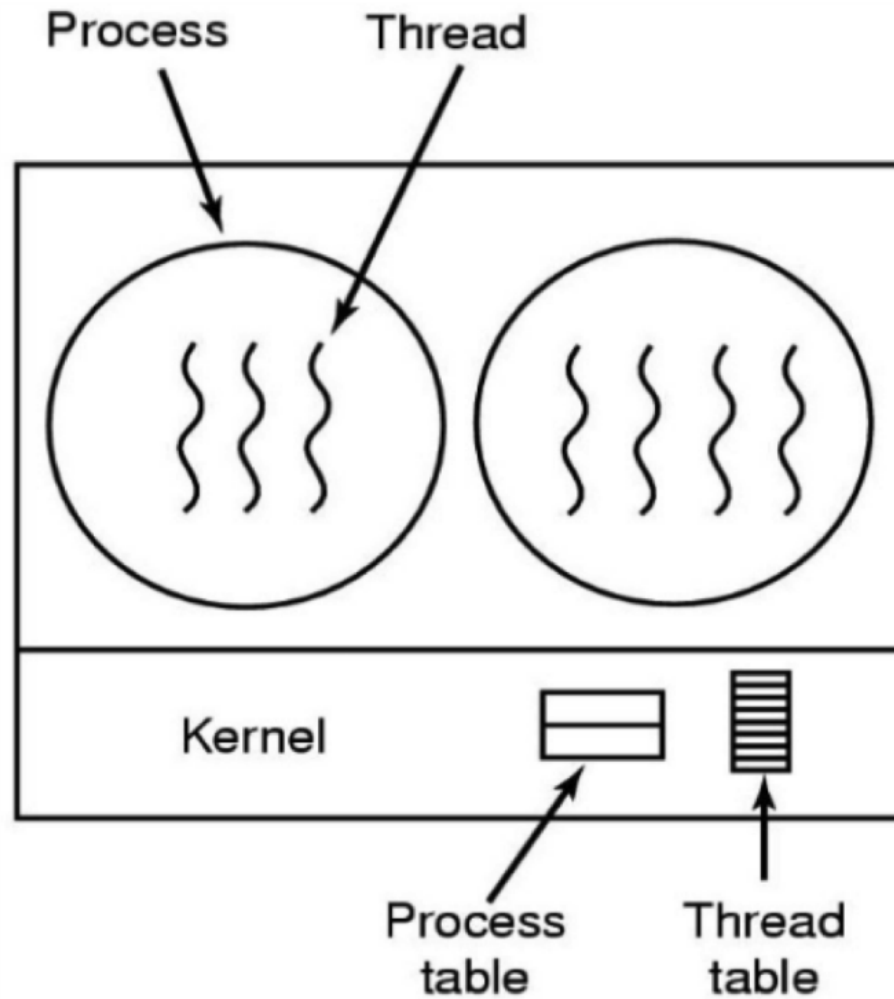
Avantaje

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

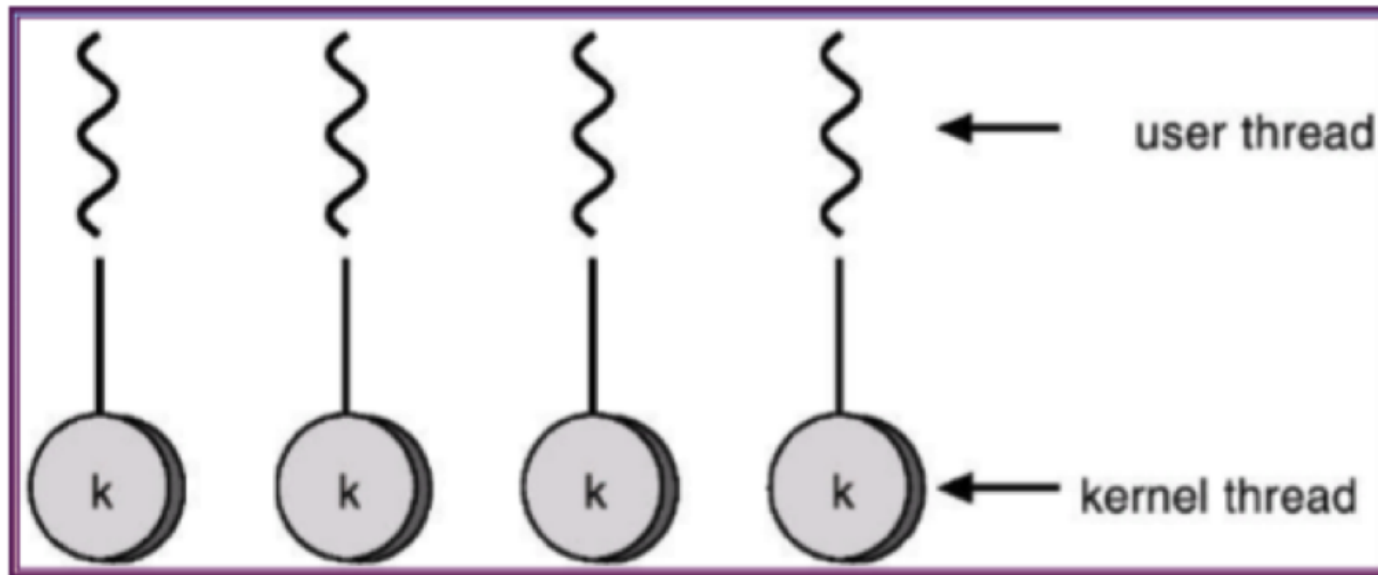
Dezavantaje

Crearea si schimbarea de context este mai lenta

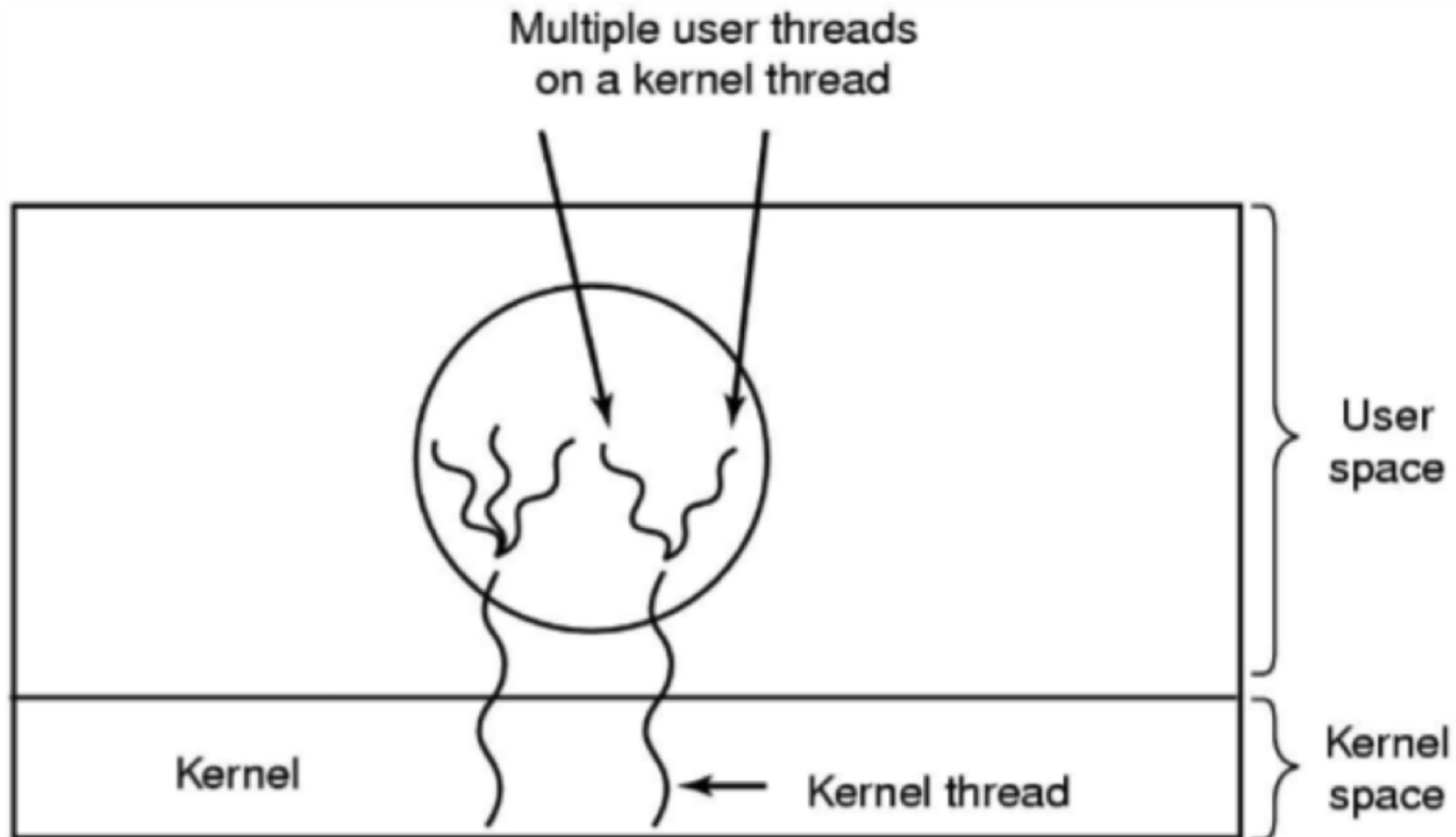
Implementare thread-uri in kernel



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

```
int main() {  
    int data = 1000000;  
    int i = 1;  
    while (i <= 1000000) {  
        cout << i << endl;  
        i++;  
    }  
}
```

Reentranta

```
#include <string>  
using namespace std;  
int main() {  
    string s = "1000000";  
    cout << s << endl;  
}
```

Reentranta in practica

Multi thread-uri accesand aceleasi variabile in memoria
Sunt accesii simultani? [Exemplu](#)

- Thread safety
- Functii care lucreaza cu date comune
- Operatiile de read-write. Se poate ca sa nu se scrie inapoi
- Strategii de implementare (mutex/conditii)
- mutex
- semafore
- monitor
- Read lock de tip
- readwrite
- readwrite

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 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 (cursul urmator)

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