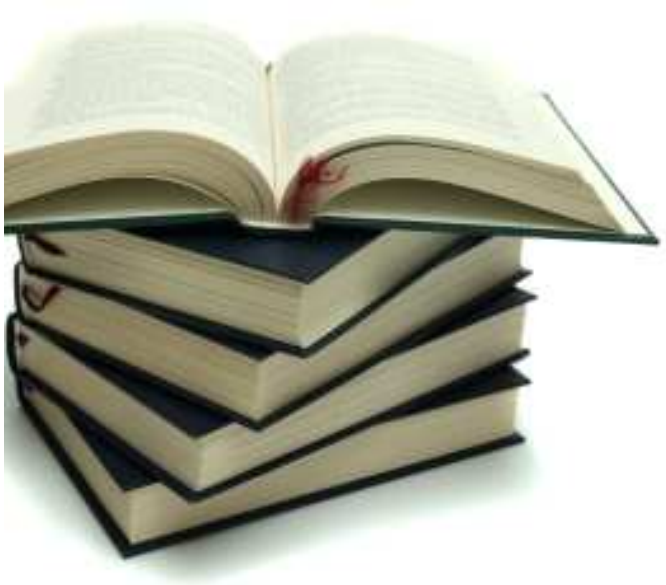
page crunch

Cursul 8



8

Fire de execuție

29 aprilie 2009

Suport curs 8

- OSC
  - Capitolul 4 – Threads
- MOS
  - Capitolul 2 – Processes and Threads
    - Secțiunea 2.2 – Threads
- Beginning Linux Programming
  - Capitolul 12 – POSIX Threads
- Windows System Programming
  - Capitolul 7 – Threads and Scheduling
  - Capitolul 8 – Thread Synchronization

Cuprins

- De ce fire de execuție?
- Modele de fire de execuție
- POSIX Threads
- Thread-uri în Windows
- Mecanisme de sincronizare
- Probleme de sincronizare

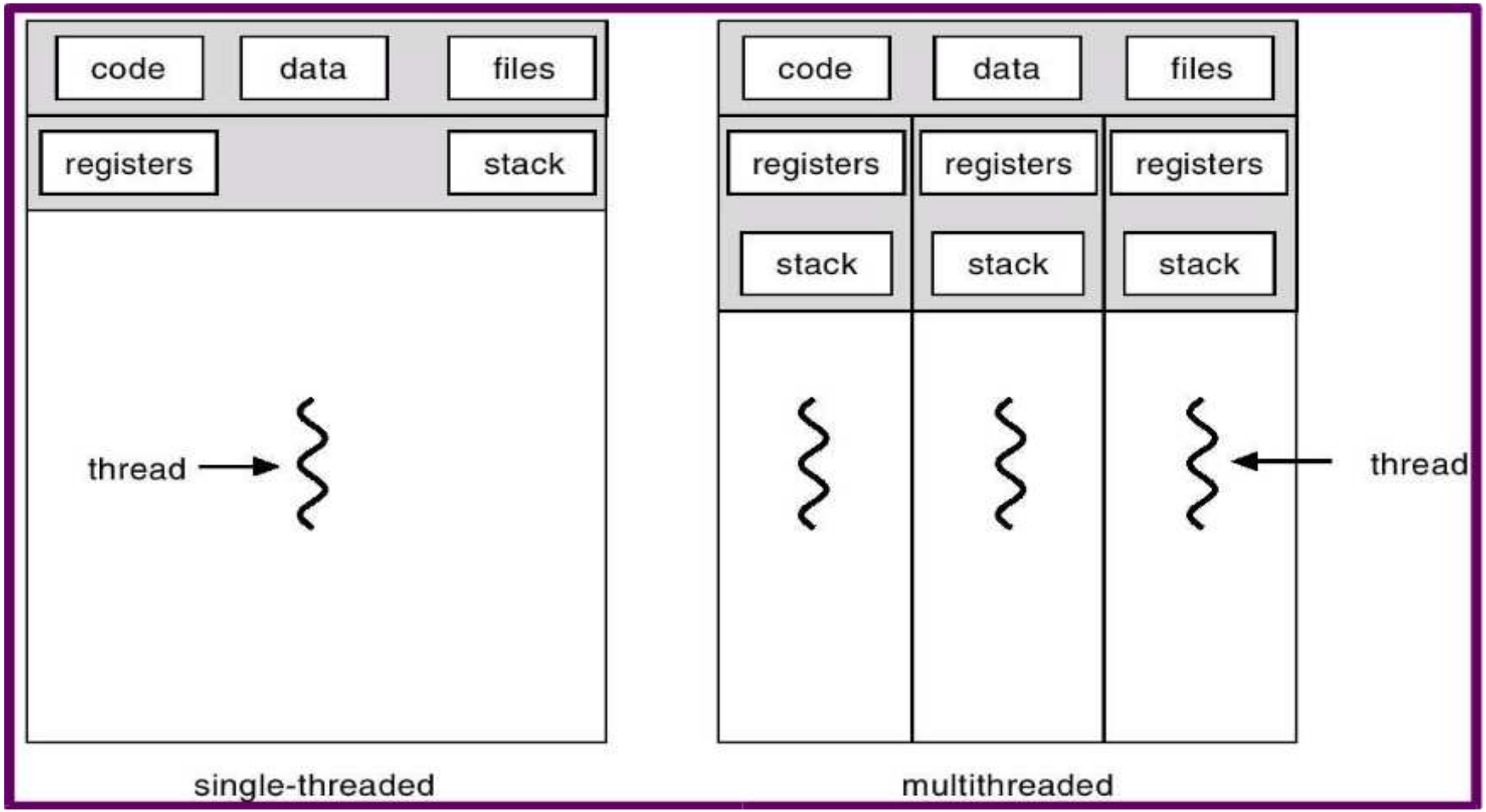
Fire de execuție

- Thread-uri
- O secvență de control în cadrul unui proces
- LWP – lightweight processes
- Un proces are unul sau mai multe thread-uri
- Partajează resursele procesului
- SO multithreaded
  - suportă rularea mai multor thread-uri în cadrul aceluiași proces

Thread-uri vs. procese

- Instanțe de execuție
- SO folosește procesele pentru a grupa resurse
  - proces – abstractizarea execuției, resurselor, spațiului de adresă
- SO folosește thread-uri pentru a grupa informații necesare execuției unui flux de instrucțiuni
  - thread – abstractizează execuția (resursele sunt deținute de proces)

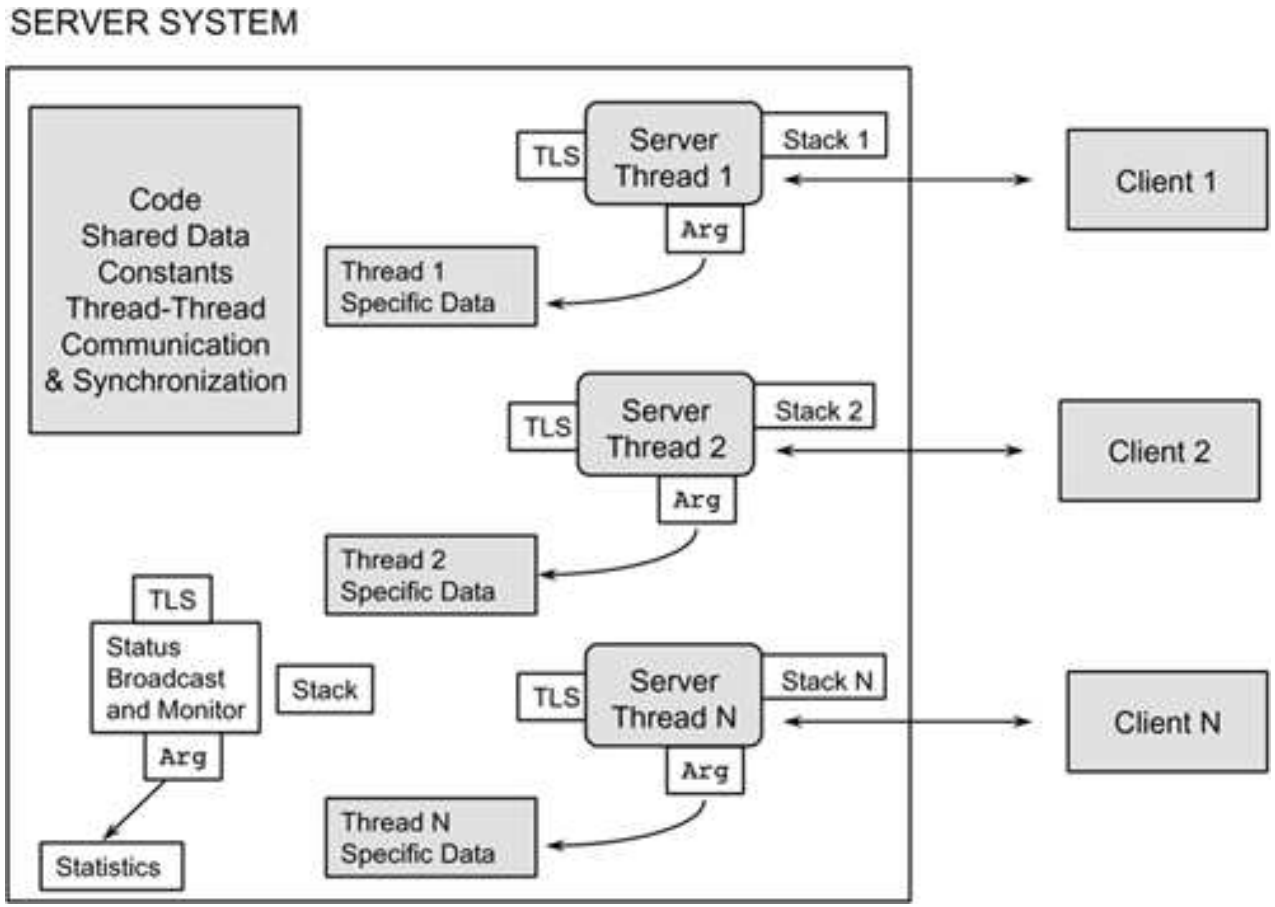
Procese singlethread si multithread



De ce thread-uri?

- Timp de creare mai mic
- Schimbări de context rapide între thread-uri
- Ușor de partajat informații între thread-uri
  - între procese este mai dificil (se poate cu memorie partajată)
- Utile chiar si pe uniprocesor
  - mix între operații de intrare/ieșire și operații de calcul
- Planificare pe sisteme multiprocesor/multicore

De ce thread-uri? (2)



De ce nu thread-uri?

- Daca moare un thread, moare tot procesul
- Partajarea datelor – nu există protecție
  - un thread poate suprascrie datele altui thread
  - TLS (Thread Local Storage)/TSD (Thread Specific Data)
- Concurență exagerată
  - prea multe thread-uri degradează performanța sistemului: overhead context switching
  - implementări event-based, async I/O - alternativă
- Probleme de concurență
  - condiții de cursă / deadlock-uri

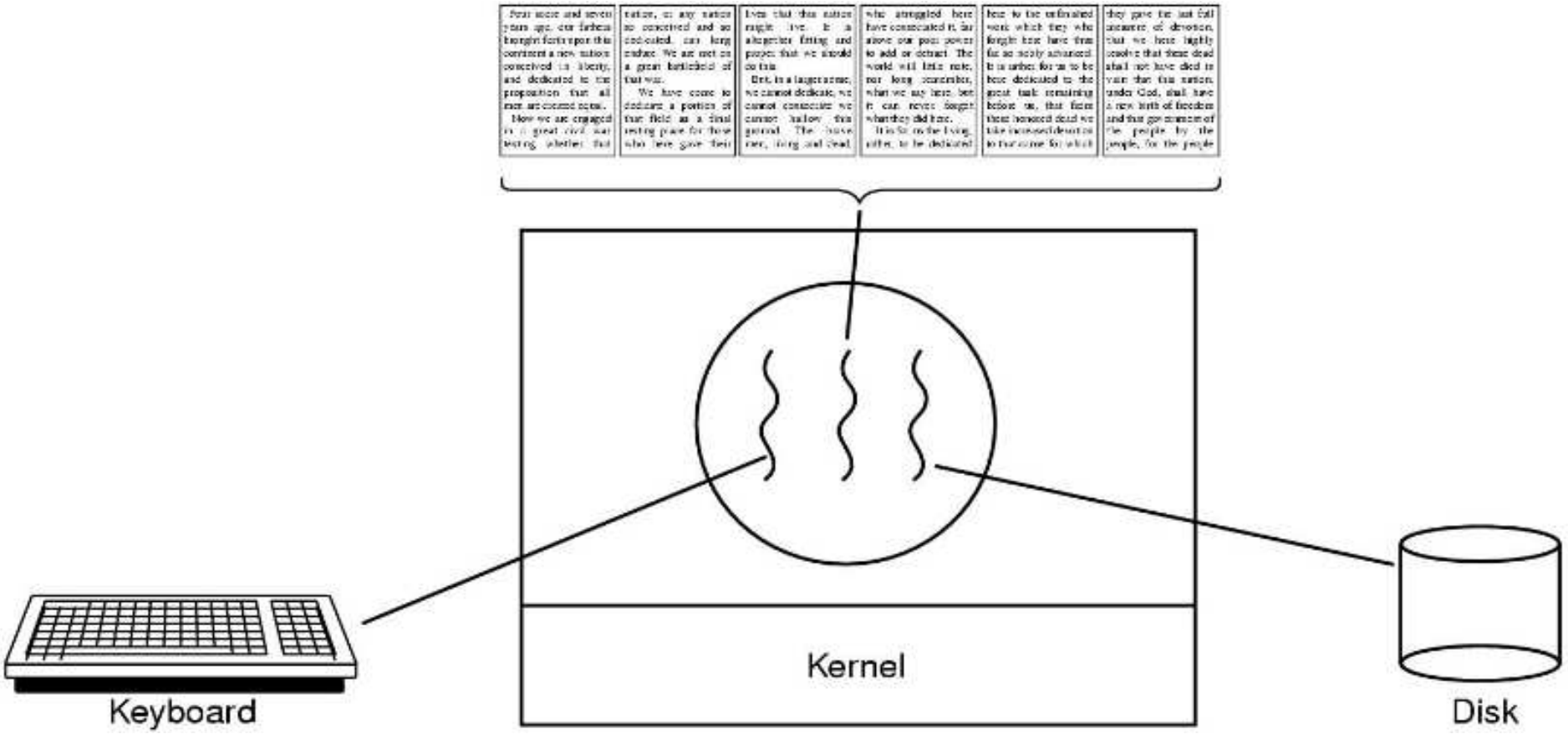
Fire de execuție ca model

- Model de execuție secvențială
- Fiecare thread execută secvențial un set de instrucțiuni
  - mai ușor de urmărit
  - mai ușor de înțeles
  - încapsularea unui flux de execuție secvențial
- Sistemele complexe pot fi privite ca o compunere de fire de execuție

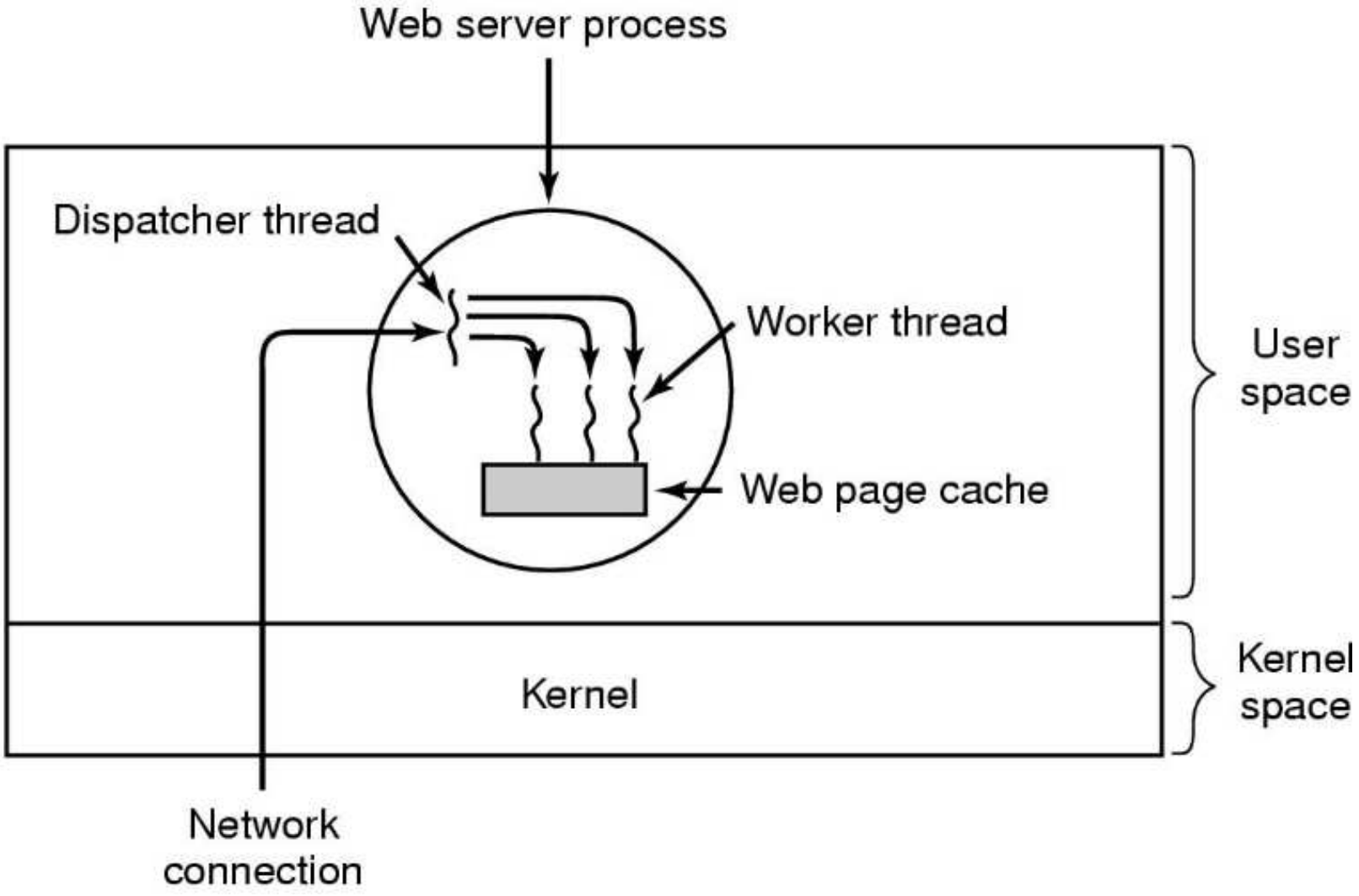
Thread-based vs. event-based

- Event-based / async I/O
  - Nu există noțiunea de concurență
  - Este nevoie de mașină de stări pentru modelul asincron
  - Programul principal (main loop) este notificat de apariția unui eveniment (ex. select)
- Avantaje/dezavantaje
  - Concurența thread-urilor poate cauza probleme
  - Lucrul cu thread-uri nu este scalabil la numărul de thread-uri
  - Thread-urile pot fi utilizate pe sisteme multiprocesor/multicore (true concurency)

Editor de texte multithreaded



Server web multithreaded



Server web multithreaded (2)

- dispatcher

```
while (1) {
     get_next_request(&buf);
     handoff_work(&buf);
}
```

- worker

```
while (1) {
     wait_for_work(&buf);
     look_for_page_in_cache(&buf, &page);
     if (page_not_in_cache(&page))
          read_page_from_disk(&buf,&page);
     return_page(&page);
}
```

Ce (nu) partajează thread-urile?

- Da (per proces)
  - variabilele globale (data, bss)
  - fișierele deschise
  - spațiul de adresă
  - masca de semnale
- Nu (per thread)
  - registrele
  - stiva
  - program counter/Instruction pointer
  - stare
  - TLS (Thread Local Storage)

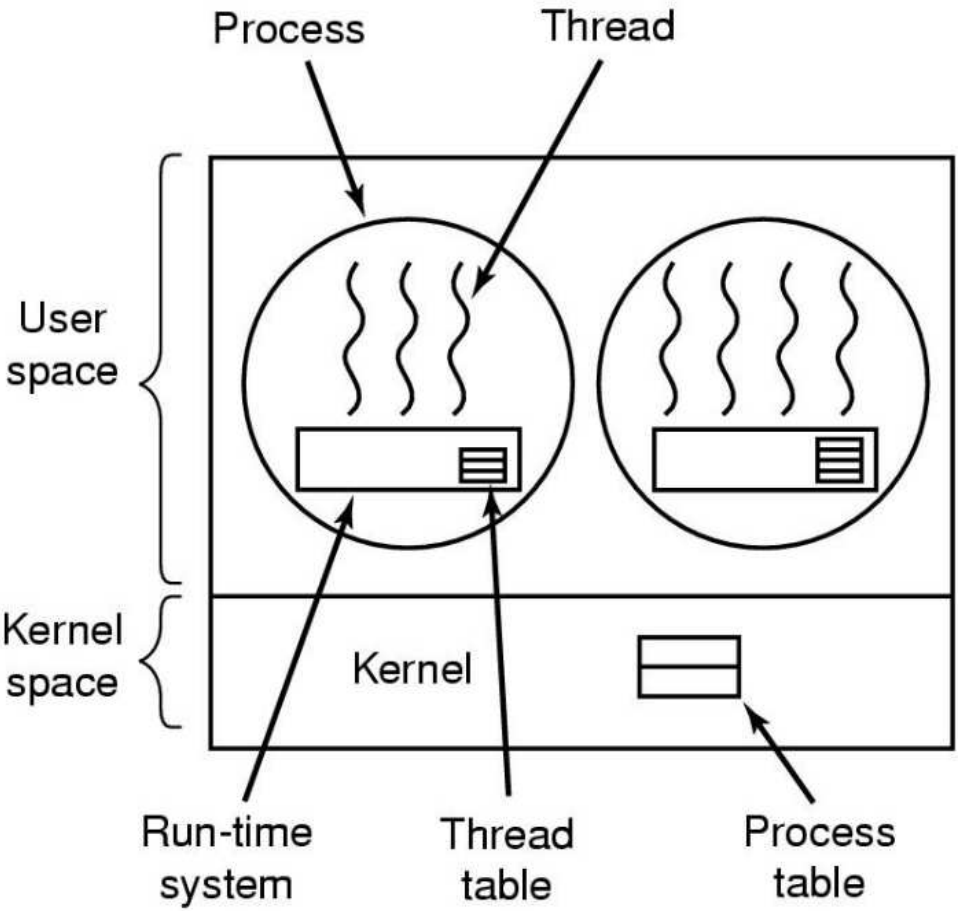
Implementarea firelor de execuție

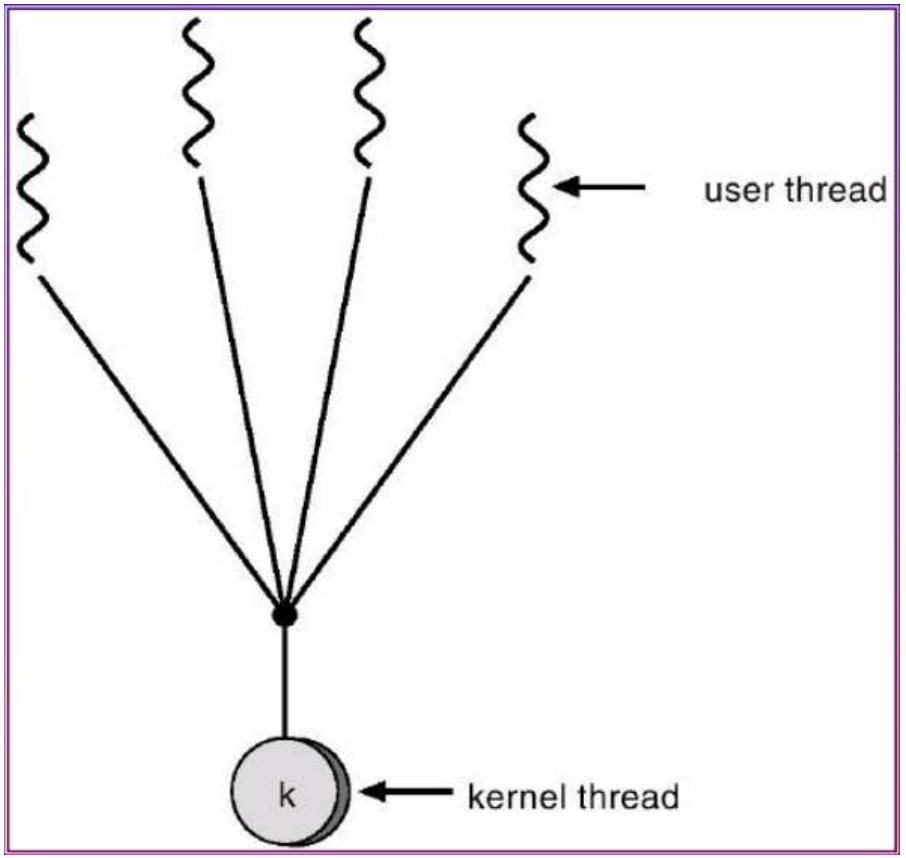
- În biblioteci
  - user-level threads
- În kernel
  - kernel-level threads
- Implementare hibridă

User-level threads

- Bibliotecă dedicată
  - creare
  - terminare
  - planificare
- Singura entitate planificabilă de SO este procesul
  - nucleul “nu vede” thread-urile
- Biblioteca menține o tabelă cu firele de execuție ale procesului
  - PC, registre, stivă, stare
- Model many-to-one

User level threads (2)





Avantaje user threads

- Ușor de integrat în SO
  - nu sunt necesare modificări
  - pot oferi suport multithreaded pe un SO fără suport multithreaded
- Schimbare de context rapidă
  - nu se execută apeluri de sistem în nucleu
- Aplicațiile pot implementa planificatoare în funcție de necesități

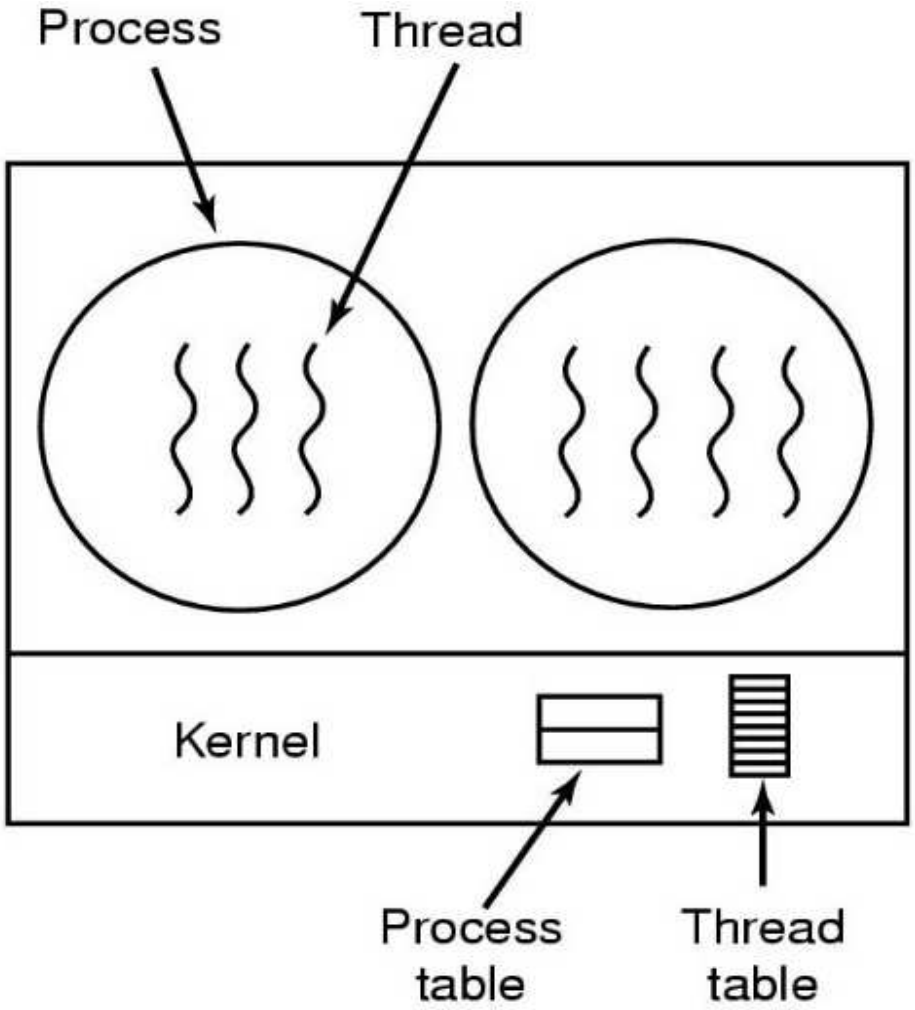
Dezavantaje user threads

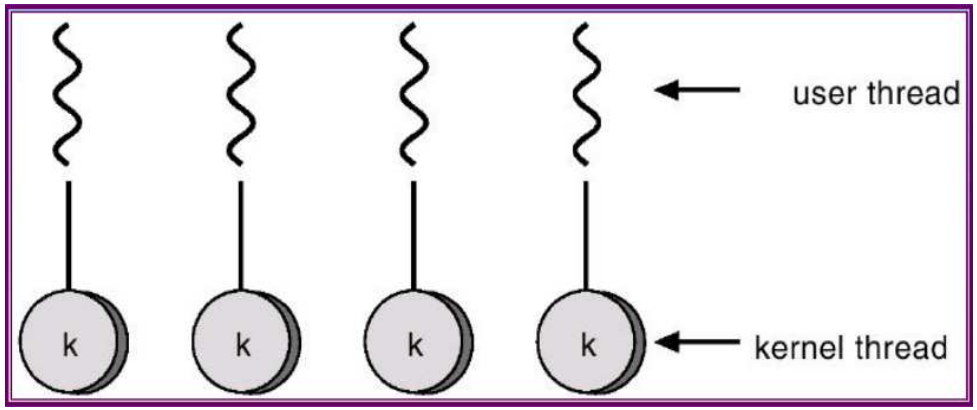
- Un apel de sistem blocante blochează întreg procesul
  - modificare SO
  - modificare bibliotecă pentru a verifica dacă un apel este sau nu blocant
  - rămân probleme la page fault-uri
- Planificare cooperativă
  - nu există întreruperi de ceas precise
  - necanismele existente pot fi folosite și de thread-uri (nu doar de bibliotecă)
- Aplicațiile care folosesc des apeluri de sistem
  - oricum este nevoie de un trap în kernel

Kernel-level threads

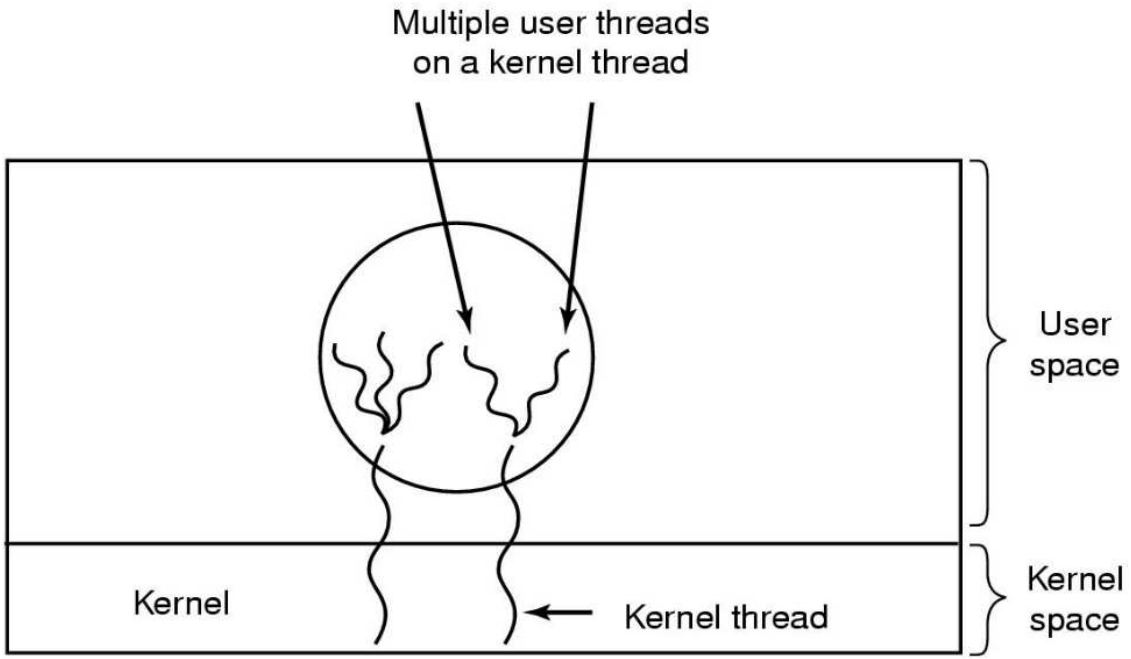
- Suport în kernel
  - creare, terminare, planificare
  - entități planificabile
- Model unu la unu (1:1)
- Dezavantaje
  - creare și schimbare de context mai lentă
- Avantaje
  - fără probleme cu apeluri blocante
  - fără probleme cu page fault-uri
  - pot fi planificate pe sisteme multiprocesor

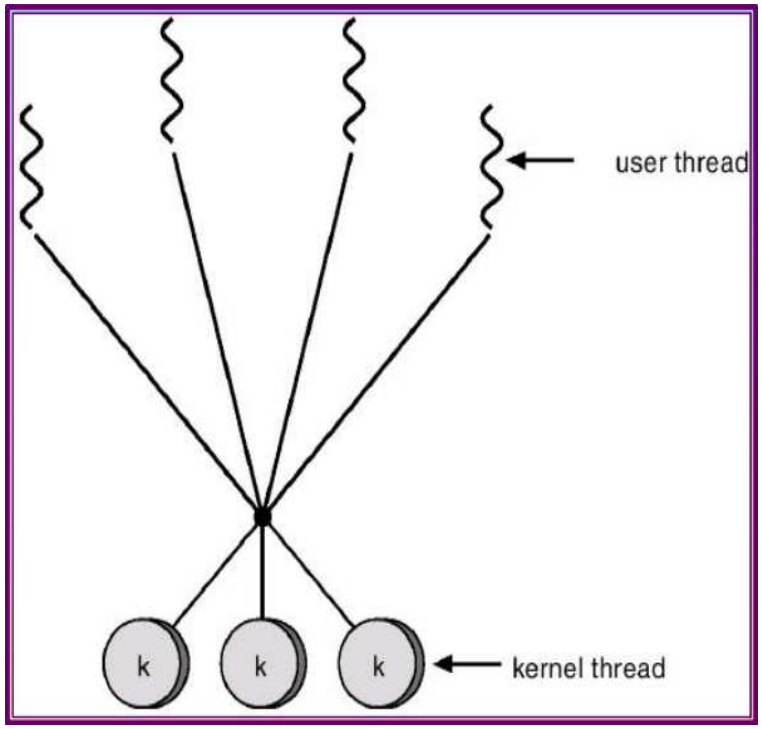
Kernel-level threads (2)





Abordare hibridă





Operații cu thread-uri

- Creare
- Încheierea execuției
- Terminare (cancellation)
- Așteptare (join)
- Planificare
- Sincronizare

POSIX threads

- Standard POSIX (IEEE 1003.1c)
- API pentru crearea și sincronizarea thread-urilor
- API-ul specifică doar comportamentul
  - implementarea se realizează în bibliotecă
- Sisteme UNIX
- Inclus header-ul: `#include <pthread.h>`
- Legarea bibliotecii: `-lpthread`
- `man 7 pthreads`

Operații POSIX threads

- Creare

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

- pthread_exit(void *ret);
- pthread_join(pthread_t tid, void **ret);
- pthread_cancel(pthread_t tid);
- apt-get install manpages-posix manpages-posix-dev
  - man -S 3posix pthread_create

Thread-uri în Linux

- Firele de execuție sunt implementate în kernel
- Nu se face distincția între thread-uri și procese
- Procesele și thread-urile sunt abstractizate în task-uri
  - Fiecare thread/proces este descris de structura `struct task_struct`
- Diferența dintre thread-uri și procese este partajarea anumitor resurse (spațiu de adresă, fișiere)

Apelul de sistem clone

- Specific Linux
- Folosit de fork() și de NPTL pentru crearea proceselor și a firelor de execuție
- Diferite flag-uri specifică ce resurse sunt partajate
  - CLONE_NEWNS
  - CLONE_FS, CLONE_VM, CLONE_FILES
  - CLONE_SIGHAND, CLONE_THREAD
- `man 2 clone`

NPTL

- New POSIX Thread Library
- Implementarea curentă din glibc/Linux
- Necesită kernel 2.6 (futex-uri)
- Implementare 1:1 (kernel-level threads)
- Folosește clone
- Thread-urile sunt grupate în același grup de thread-uri (thread group)
  - getpid(2) întoarce tgid (thread group id)
- http://people.redhat.com/drepper/nptl-design.pdf

Thread-uri în Windows

- Model hibrid
- Firele de execuție implementate în user-mode sunt denumite fibre
  - planificare cooperativă
  - blocarea unei fibre blochează firul de execuție
- Firele de execuție implementate în kernel

Operații cu thread-uri în Windows

- HANDLE CreateThread(...)
- ExitThread
- WaitForSingleObject/MultipleObjects
- GetExitCodeThread
- TerminateThread
- TlsAlloc
- TlsGetValue/TlsSetValue

Reentranță

- Apel reentrant
  - un apel în execuție al unei instanțe nu afectează un apel simultan
  - nu lucrează cu variabile globale/statice
  - nu apelează funcții non-reentrante
  - nu se referă doar la thread-uri: semnale, întreruperi
- Contează interfața oferită de funcție
  - se folosesc doar stiva și argumentele transmise funcției
- Anumite apeluri de bibliotecă au versiuni reentrante (gethostbyname_r)
  - pentru activare, în POSIX threads se definește macroul _REENTRANT

Thread-safety

- Funcții/apeluri thread-safe
- Operații sigure în context multithreaded
- Pot fi apelate simultan de mai multe thread-uri
- Contează implementarea
- Implementare
  - reentranță
  - excludere mutuală
  - TLS
  - operații atomice

Mecanisme de sincronizare

- Apeluri reentrante
- Acces exclusiv (concurență)
  - secțiuni critice
  - mutex-uri
- Sincronizare/coordonare (notificare/așteptare)
  - variabile condiție
  - evenimente
  - semafoare
  - monitoare

Mutex/Critical Section

- Acces exclusiv
- POSIX threads: pthread_mutex_...
  - init/destroy
  - lock/unlock/trylock
- Win32 API
  - Create/OpenMutex
  - Close Handle
  - ReleaseMutex
  - WaitForSingleObject
  - Initialize/DeleteCriticalSection
  - Enter/TryEnter/LeaveCriticalSection

Semafoare

- sem_t sem;
- sem_init
- sem_destroy
- sem_wait
- sem_trywait
- sem_post

- HANDLE hSem;
- CreateSemaphore
- CloseHandle
- WaitForSingleObject
- ReleaseSemaphore

Variabile condiție/evenimente

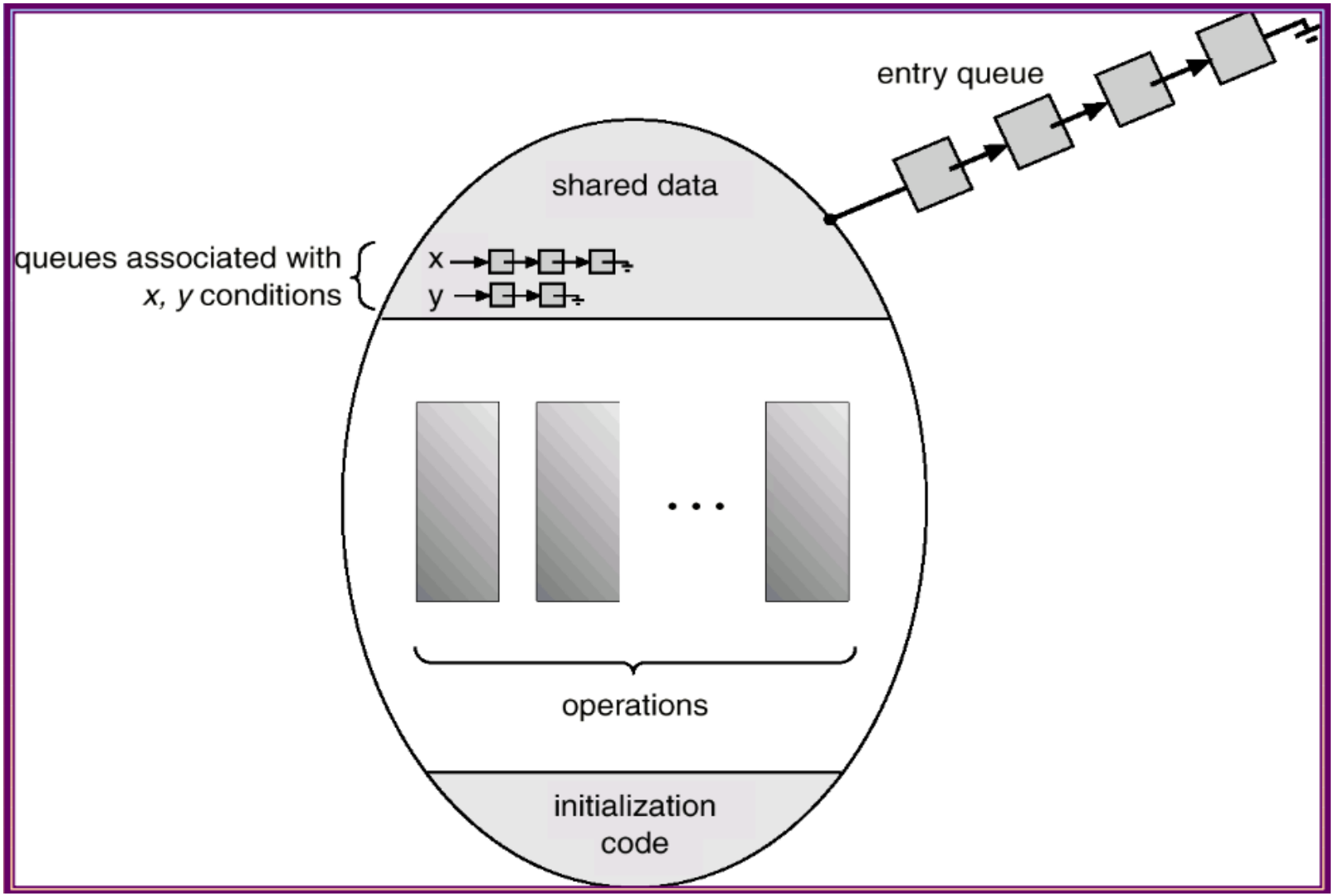
- pthread_cond_t cond
- pthread_cond_init
- pthread_cond_destroy
- pthread_cond_signal
- pthread_cond_broadcast
- pthread_cond_wait

- HANDLE event
- CreateEvent
- OpenEvent
- SetEvent
- ResetEvent
- PulseEvent
- WaitForSingleObject

Monitoare

- Hoare (1974), Brinch Hansen (1975)
- O colecție de proceduri și structuri de date
  - un singur proces/thread poate rula la un moment dat
- Datele pot fi accesate doar prin intermediul procedurilor monitorului
- Procesele trec din starea READY în BLOCKED (și invers)
  - proceduri de forma signal/wait

Monitoare (2)


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

Operații pe monitor

- Intrarea în monitor: m.entry();
- Ieșirea din monitor: m.leave();
- Semnalarea unei condiții: m.signal(cond);
- Așteptarea unui condiții: m.wait(cond);
- Cozi de așteptare în monitor
  - cozi pentru așteptare pentru fiecare condiție
  - coadă de intrare în semafor

Funcționare monitor

- În cazul mai multor apeluri de proceduri într-un monitor, un singur thread rulează
  - thread-ul deține monitorul
- Când se termină de rulat procedura se spune că thread-ul a cedat monitorul
- Un apel wait
  - thread-ul curent se blochează
  - este dispus în coada de așteptare specifică
  - cedează monitorul
- Politici de planificare
  - signal and wait
  - signal and continue

Problema formării apei

- H2O problem
- Thread-urile reprezintă 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 trebui să aștepte un atom de oxigen
- Dacă există un atom de oxigen, va trebui să aștepte doi atomi de hidrogen

Problema formării apei - semafoare

```
Semaphore hsem;
Semaphore osem;
Mutex m;

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

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

Problema formării apei - monitoare

```
Monitor m;
Cond m.oxy_cond;
Cond m.hydro_cond;


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

```
void hydro_fun(void)
{
      m.enter();
      h_count++;
      if (h_count > 2 && o_count >= 1) {
            m.h_cond.signal();
            m.h_cond.signal();
            m.o_cond.signal();
            m.h_cond.wait();
      }
      else if (h_count == 2 && o_count >= 1) {
            m.h_cond.signal();
            m.o_cond.signal();
      }
      else
            m.h_cond.wait();
      m.leave();
      bond();
}
```

Cuvinte cheie

- thread
- multithreading
- Thread Local Storage
- thread-based
- event-based
- user-level threads
- kernel-level threads
- POSIX Threads
- apelul clone
- NPTL
- fibre
- reentranță
- thread-safe
- mutex
- semafor
- monitor
- variabilă condiție
- event

Exercițiu 1

- Indicați două zone de memorie (nu registre) care nu sunt partajate între thread-urile aceluiași proces.
- Câte cozi de așteptare există în cadrul unui monitor? Câte thread-uri pot aștepta la fiecare coadă la un moment dat?

Exercițiu 2

- Fie un sistem cu 4 procesoare și o aplicație cu 4 fire de execuție ce folosește un sistem de thread-uri implementat în userspace. Presupunând că nu mai există alte procese/fire de execuție în sistem, și că fiecare thread rulează timp de 10ms, apoi așteaptă 10ms la IO, și rulează apoi încă 10ms, determinați timpul minim de execuție a aplicației.

Exercițiu 3

- Care din următoarele funcții este, probabil, reentrantă?
  - char *ctime(const time_t *timep);
  - char *strdup(const char *s);
  - char *strchr(const char *s, int c);
  - void *memcpy(void *dest, const void *src, size_t n);

Întrebări

?