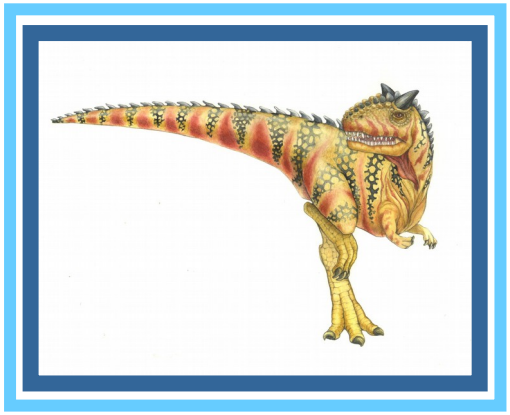
Chapter 4: Threads & Concurrency



Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designng multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Designing multithreaded applications using the Pthreads, Java, and Windows threading APIs

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Aplicatii concurente

- proces = un singur punct de executie in aplicatie (o singura instanta in rulare a aplicatiei)
- unele aplicatii pot profita de existenta mai multor puncte de executie simultana in cadrul aplicatiei (mai ales pe multiprocesoare)
- ex. 1: procesor cu 4 core-uri + aplicatie de filtrare de imagini care imparte imaginea in 4 cadrane, fiecare core filtrand un cadran
  - avantaj: descompunerea activitarilor mari in activitati mai simple care ruleaza simultan => reducerea timpului de rulare
- ex. 2: un singur procesor + program de gestiune a ferestrelor in GUI
  - verifica si proceseaza input-ul de la tastatura, mouse si retea pt. a produce output pe ecran
  - conceptual, exista mai multe activitati concurente: verificare/procesare input tastatura, verificare/procesare input mouse, verificare/procesare input retea, afisare bitmap-uri pe ecran
  - toate aceste activitati sunt codate in module separate, cu date private si comunica intre ele prin data partajate

Aplicatii concurente (cont’d)

- ex. 3: server de retea
  - activitatea principala: asteapta cereri de la client, le proceseaza si trimite inapoi raspunsurile
  - cu un singur punct de executie procesarea unei cereri particulare ia mult timp (de pilda, in asteptarea datelor de pe disc) => se intarzie foarte mult tratarea altor cereri (situatie similara cu cea care a condus la nevoia de multitasking)
  - solutie: fiecare cerere client e procesata in alt punct de executie al aplicatiei server

Procese si date partajate

- crearea de puncte multiple de executie in program se poate face cu procese care partajeaza date (sa zicem prin memorie partajata, cu acces protejat cu semafoare/locks)
  - procesele au date private (datorita protectiei MMU a spatiului de adresa)
  - comunica intre ele prin IPC (memorie partajata)
  - daca exista capacitare de multiprocesare, procesele pot rula simultan pe mai multe procesoare
- puncte nevralgice
  - crearea proceselor
  - context-switch-ul

Crearea proceselor

- contextul unui proces e stufos (stocat in PCB in kernel)
- include starea complete a CPU, registre de gestiunea memoriei, tabele de pagini de memorie, descriptori de fisiere, actiuni asociate semnalelor, etc

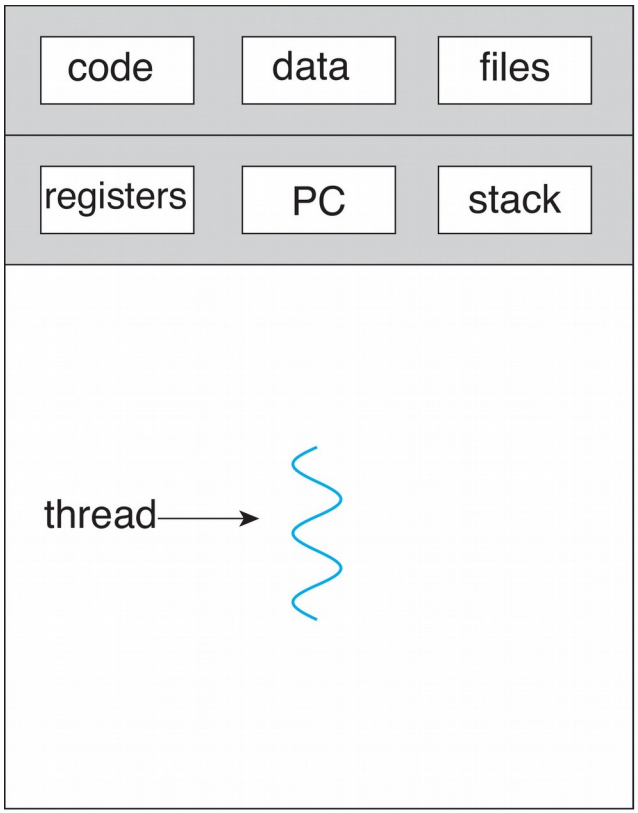
  => cost semnificativ de creare a unui proces

- daca acest cost e prea mare, aplicatiile pot sa nu foloseasca eficient procesoarele:
  - ex aplicatie de filtrare imagini
  - daca timpul de creare a unui proces e comparabil cu timpul de filtrare al unui cadran, e posibil ca filtrarea intregii imagini cu 4 procese sa dureze mai mult decat filtrarea ei cu un singur proces (“slowdown”)

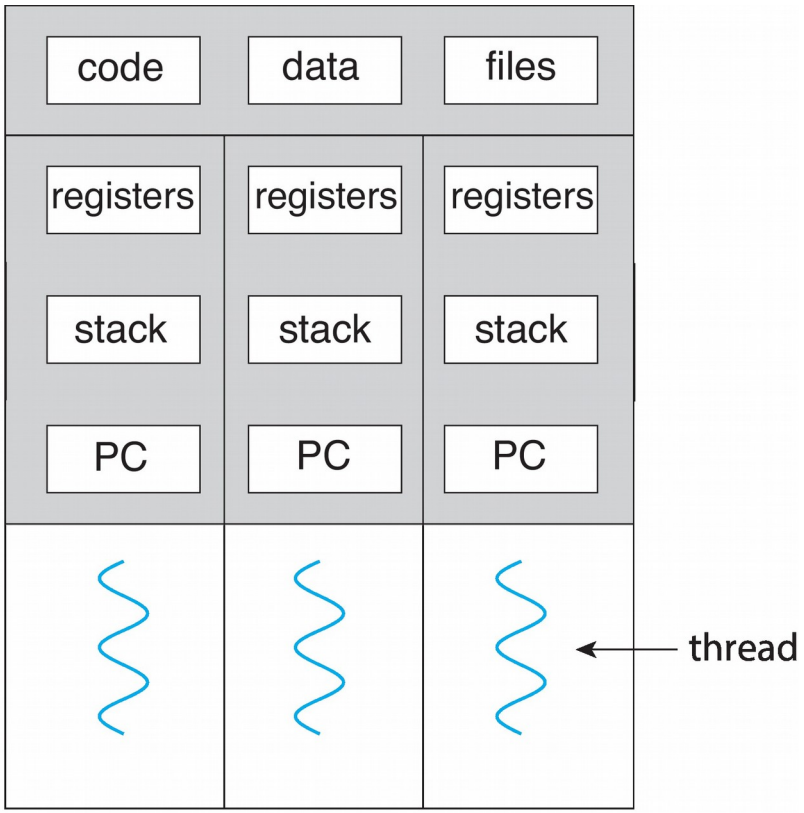
Context-switch

- salvarea contextul unui proces si incarcarea contextului unui nou proces poate implica un cost semnificativ avand in vedere bogatia de informatii din PCB
- ex: secventa de evenimente pt. producator-consumator cu zero buffering (“rendez-vous”)
  - producatorul produce un element si se blocheaza
  - sistemul face context-switch si aduce consumatorul pe procesor
  - consumatorul consuma elementul si se blocheaza
  - sistemul face context-switch si aduce producatorul inapoi pe procesor pt a produce un nou element
- daca timpul de context-switch e mare, viteza de transfer a datelor intre producator si consumator e serios afectata (fiecare transfer implica doua context-switch-uri); in orice caz, e imposibil sa se atinga viteza teoretica maxima de transfer

Single and Multithreaded Processes

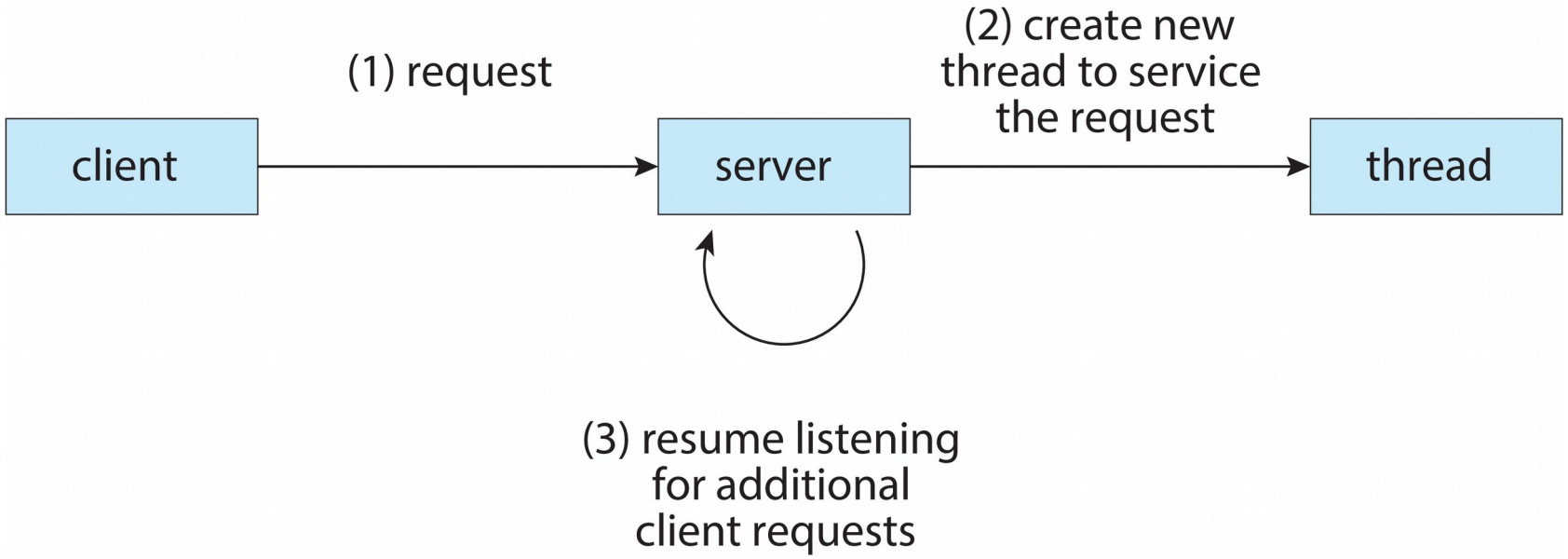


single-threaded process



multithreaded process

Multithreaded Server Architecture



Benefits

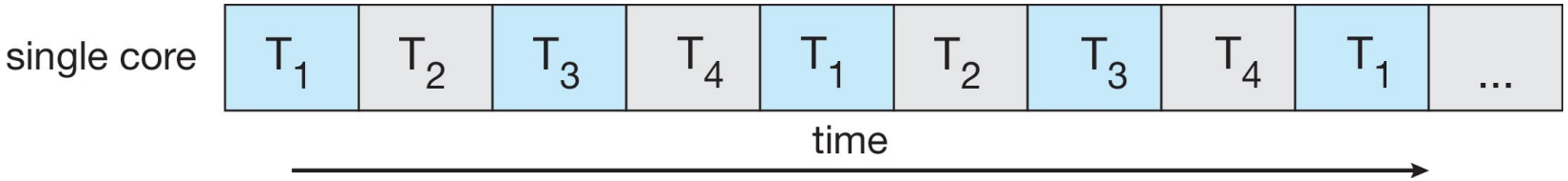
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multicore architectures

Multicore Programming

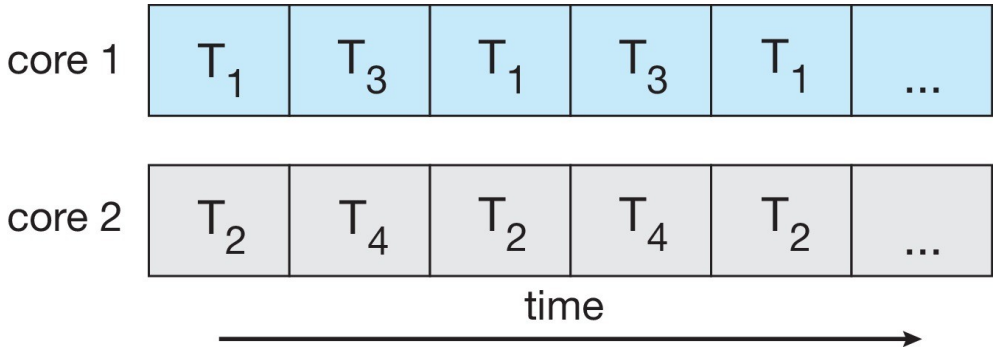
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



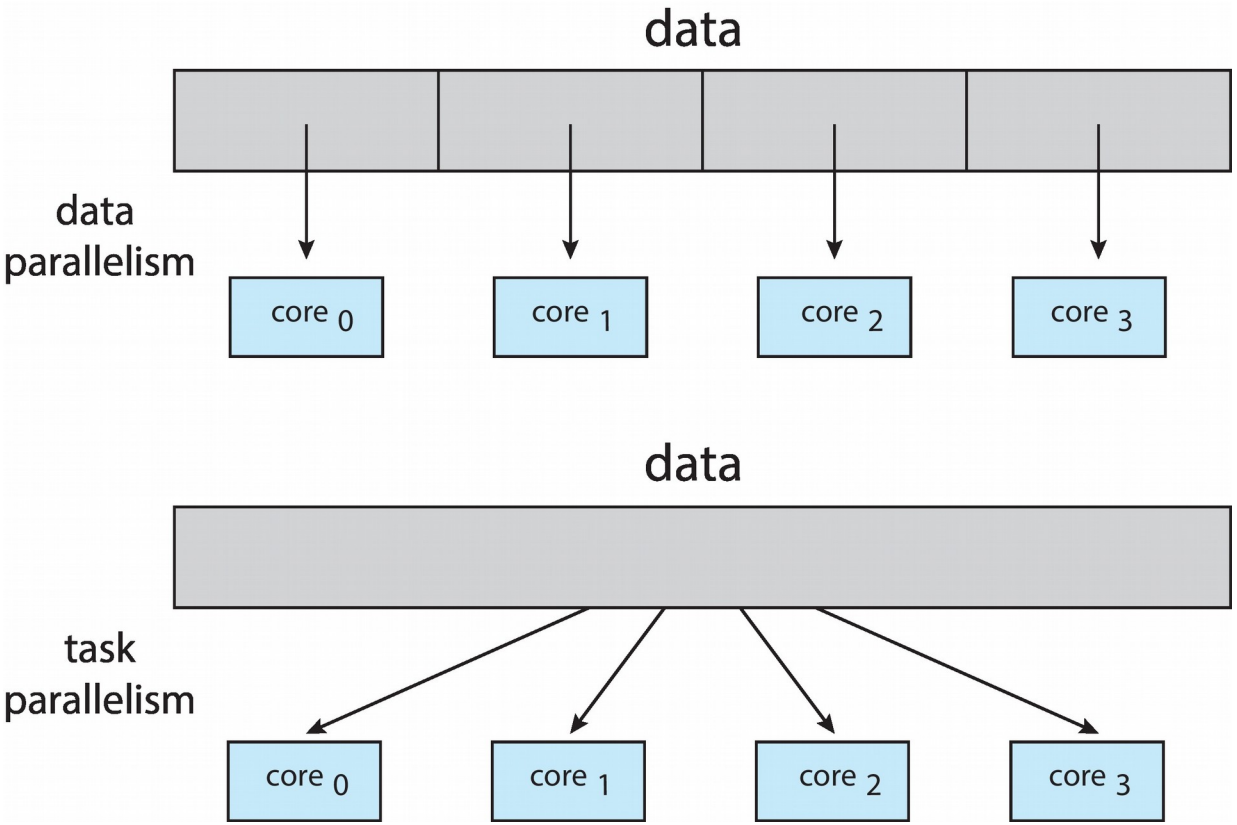
- **Parallelism on a multi-core system:**



Multicore Programming

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

Data and Task Parallelism



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

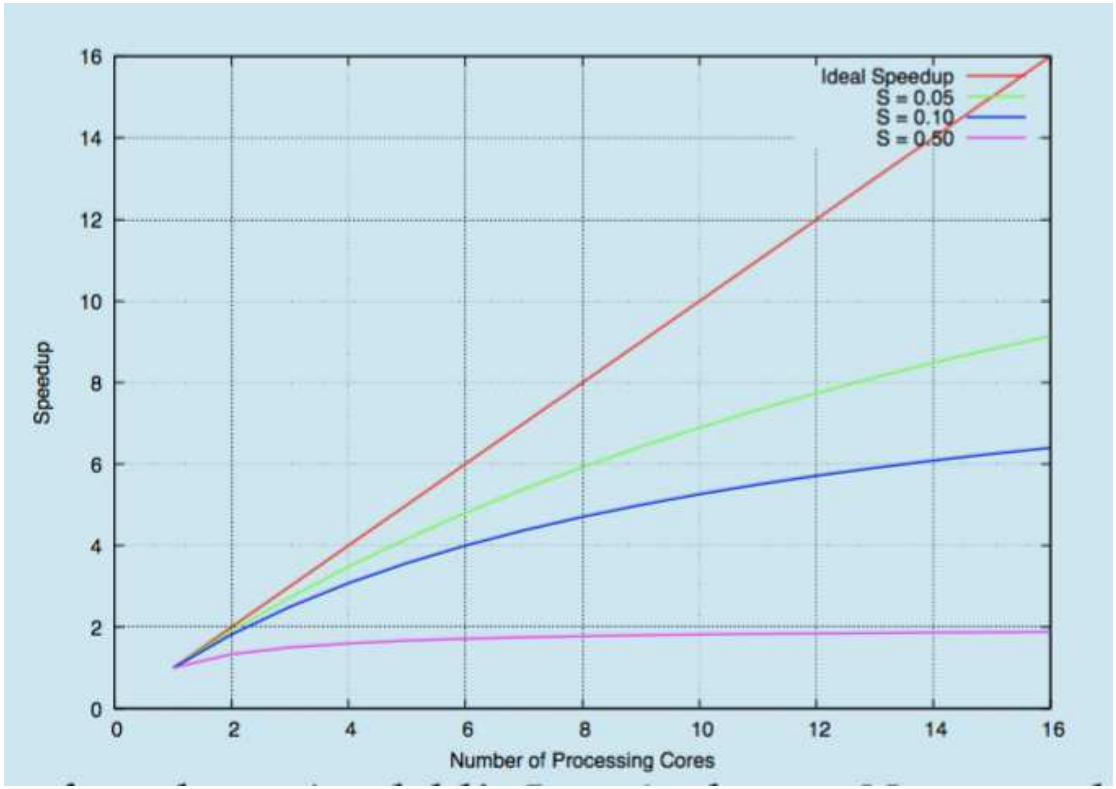
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

Amdahl's Law


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

Legea lui Gustafson

- Amdahl

$$Speedup = 1 / (s + p / N)$$

- presupunere implicita: *p* independent de *N (i.e., dimensiunea problemei e fixa)*
- presupunere nerealista in practica unde dimensiunea problemei scaleaza cu numarul de procesoare => o noua abordare, in care se pp ca timpul de rulare e constant, nu dimensiunea problemei
- acum, se considera ca *p* variaza liniar cu *N*
- observatie practica: partea seriala *s* (initializare vectori, incarcare program, bottleneck-uri seriale, I/O) nu creste odata cu dimensiunea problemei

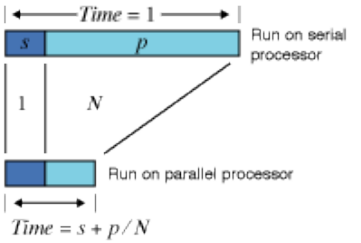


FIGURE 2a. Fixed-Size Model: $Speedup = 1 / (s + p / N)$

Legea lui Gustafson (cont'd)

- definitie noua pt speedup

  Scaled speedup = $s + pN = s + (1 - s)N = N + (1 - N)s$

- daca s e constant, dependenta de N e liniara cu panta $1 - s$

  => schimbare de paradigma: scaled speedup-ul este de fapt "slowdown-ul" (incetinirea) teoretica pe care o inregistreaza un program paralel atunci cand ar rula ipotetic pe o masina cu un singur procesor

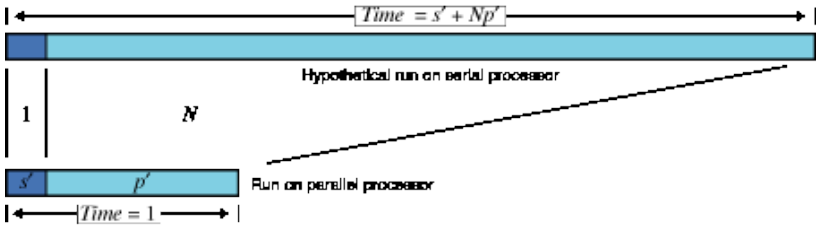


FIGURE 2b. Scaled-Size Model: Speedup = $s + Np$

Concluzii legea lui Gustafson

- accentul folosirii unui nr crescut de procesoare cade pe necesitatea de a rula programe mai mari in acelasi timp daca se poate, nu pe scaderea timpului de executie al unei probleme de dimensiune fixa
- obs: nu orice problema poate fi marita arbitrar, ca atare exista si limitari ale legii lui Gustafson

Fire de executie (threads of execution)

- modalitate de a reduce costurile crearii punctelor de executie in aplicatie si a schimbarii contextelor de executie intre ele
- idee: puncte de executie multiple din aplicatie partajeaza o parte din contextul programului (registrele de gestiune a memoriei, tabelele de pagini, descriptorii de fisiere deschise, samd)
- DAR, fiecare punct de executie are:
  - o copie individuala a unui subset al contextului de executie a aplicatiei (de ex. contextul CPU)
  - structuri de date necesare punctului de executie (de ex. stiva)
- apare o noua abstractie de nivel inalt, *firul de executie (thread)* = un punct de executie cu context redus in cadrul programului
  - referit uneori din cauza acestei viziuni ca fiind un “proces usor” (lightweight process, LWP)

Caracteristici threaduri

- ruleaza secvential, au program counter si stiva propria
- multiplexeaza accesul la CPU ca si procesele (pe multiprocesoare ruleaza in paralel)
- pot crea alte threaduri
- pot executa apeluri de sistem
  - daca un thread se blocheaza intr-un apel sistem, alt thread primeste procesorul
- analogie posibila
  - threadul este fata de proces ceea ce procesul e in raport cu procesorul
  - procesul actioneaza ca un procesor virtual pe care ruleaza thread-ul

Diferente fata de procese

- threadurile aceluiasi proces partajeaza spatiul de adresa al procesului (de ex, partajeaza variabilele globale) => un thread poate distruge usor alt thread (nu exista protectia MMU ca in cazul proceselor diferite)
- lipsa protectiei intre threaduri
  - e inevitabila prin design
  - nu e necesara (threadurile sunt parte a aceluiasi program al unui anumit utilizator)
  - nici nu e de dorit (impunerea unor domenii de protectie conduce la probleme similare proceselor)
- alte resurse partajate: acelasi set de fisiere deschise, timere, semnale, etc

Alte caracteristici ale threadurilor

- stari (la fel ca la procese): in rulare, gata de rulare, blocat, terminat
- modele de utilizare
  - cooperativ, lucru in echipa (ex filtrare de imagini)
  - master-slave/worker (ex server)
  - pipeline (ex producator-consumator)
- avantaj principal: datele partajate sunt datele globale din proces (nu e nevoie de setarea unor mecanisme IPC de tipul memoriei partajate)
  - buffer global pt. producator-consumator
  - argument puternic pt sistemele multiprocesor unde threadurile pot rula pe CPU-uri diferie => partajarea implicita a datelor prin spatiul comun de adresa (nu e nevoie de mecanisme speciale de partajare ca in cazul proceselor, eg memorie partajata IPC)

Design-ul pachetelor de threaduri

- pachet de threaduri = colectie de primitive (apeluri de biblioteca) pt lucrul cu thread-uri
- (1) gestiunea threadurilor
  - creare thread: primeste ca argumente functia care reprezinta punctul de executie initial, o stiva privata si o prioritate de planificare pe procesor; intoarce un TID (Thread ID)
  - terminare thread: explicit prin apel “exit” sau semnal de tip kill de la alt thread/proces
  - primitive pt. mecanisme de sincronizare, necesare datorita existentei datelor partajate (uzual mutex-uri, dar in mod notabil si variabile de conditie folosite in conexiune cu un mutex)
  - ex: acquire/release resource folosind mutex+condition variable
- (2) planificare (aceeasi algoritmi ca si la procese, vom discuta la planificarea proceselor/threadurilor)

Design-ul pachetelor de threaduri

- (3) probleme de reentranta
  - scenariu: doua threaduri T1 si T2 executa concurent apeluri sistem, T1 reuseste, T2 esueaza
  - daca T1 nu evalueaza *errno* inainte ca T2 sa execute apelul sistem care esueaza, T1 va crede eronat ca apelul sau sistem a esuat
  - problema principiala: *errno* e variabila globala
  - solutii posibile: (a) protejarea *errno* cu mutex-uri

  (b) crearea unei copii private a lui errno prin intermediul unor variabile globale thread-ului apelant, dar private (invizibile) celorlalte thread-uri => TLS (Thread-Local Storage)

Implementarea threadurilor kernel

- pachetele de threaduri se pot implementa in kernel sau in spatiul utilizator
- threadurile kernel separa campurile din PCB care ajuta la crearea unui punct de executie si le stocheaza intr-un TCB
- astfel, un proces cu un singur punct de executie e reprezentat in kernel de un PCB si un TCB
- operatia de creare a unui thread (*thread_create*) este un apel sistem
  - aloca un TCB
  - aloca stive kernel si user
  - le leaga la PCB-ul procesului in care s-a facut apelul sistem

Exemplu cu doua threaduri kernel

- un thread suspendat in kernel
- altul ruland in spatiul utilizator
- pt ca sunt implementate in kernel si seamana cu procesele, threadurile kernel se mai cheama si *procese usoare* (*lightweight processes, LWP*)

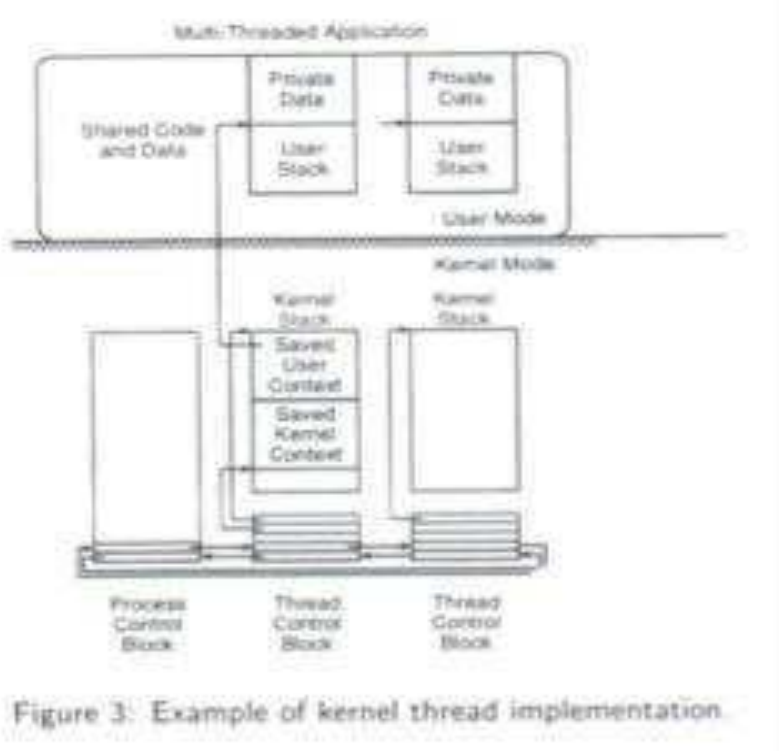


Figure 3: Example of kernel thread implementation

Costuri threaduri kernel

- costul crearii unui thread kernel << costul crearii unui proces
  - se aloca si initializeaza doar TCB-ul si stivele
  - restul contextului exista deja creat in PCB
- context switch-ul intre doua threaduri **ale aceluiasi proces** dureaza << schimbarea contextului a doua procese, pt ca nu trebuie schimbat nimic din PCB
- context switch-ul intre doua threaduri din procese **diferite** are acelasi cost ca si context switch-ul de procese (nu se schimba doar TCB-urile, ci si PCB-urile)

Planificarea kernel threadurilor pt executie

- threadurile ruleaza asincron unele fata de celelalte si pot pierde procesorul la fel ca si procesele

=> accesul la datele partajate trebuie sincronizat cand se doreste IPC

- planificatorul kernel alege urmatorul thread care trebuie sa ruleze => daca aplicatia are propria politica de planificare (de ex bazata pe prioritati) trebuie sa o comunice intr-un fel sau altul kernelului
- in cazul multiprocesoarelor, planificatorul poate asigna mai multe CPU-uri unui singur proces pt. ca threadurile sa ruleze in paralel (“gang scheduling”)
- daca un thread se blocheaza in kernel (de ex prin apel de sistem I/O) si cuanta de timp alocata procesului nu a expirat, planificatorul cauta in lista de TCB-uri un thread gata de rulare *din acelasi proces* si ii acorda procesorul

Protectia kernel threadurilor

- observatia generala despre threaduri e valabila si pt threaduri kernel
- un thread poate corupe stiva altui thread, de pilda => se distrug datele private ale altui thread (variabilele automatice/locale)
- solutie: implementarea stivelor in spatii de adresa diferite, dar asta mareste costul context switch-ului
- mai exact, ar fi nevoie de salvarea si reincarcarea contextelor de executie referitoare la gestiunea memoriei, pe langa contextul uzual din TCB

Dezavantajele kernel threadurilor

- desi mai putin costisitoare ca procesele, anumite aspecte le fac nepotrivite pt utilizatori
- (1) *thread_create* este apel sistem => costisitor pt procese care creeaza multe threaduri
- (2) context switching-ul de threaduri necesita intrarea si iesirea in/din kernel mode => overhead aditional context switching-ului obisnuit
- (3) implementarea in kernel e **inflexibila**
  - impune un model de threaduri care nu e potrivit pt orice aplicatie
  - codul planificatorului (scheduler) nu e accesibil (fiind in kernel) => greu de adaptat pt cerintele specifice ale unei anumite aplicatii (politica de planificare nu se poate schimba usor)

User-level threads

- daca planificatorul si TCB-urile sunt implementate in spatiul utilizator costurile scad pentru ca nu mai e nevoie de transgresarea granitei kernel/user
- comparatie calitativa, intr-un ex in care un apel de procedura cost 7 usec, iar un apel sistem (trap) 19 usec:

Operatie	Thread user	Thread kernel	Proces Unix
fork	34 usec	948 usec	11300 usec
IPC synch	37 usec	441 usec	1840 usec

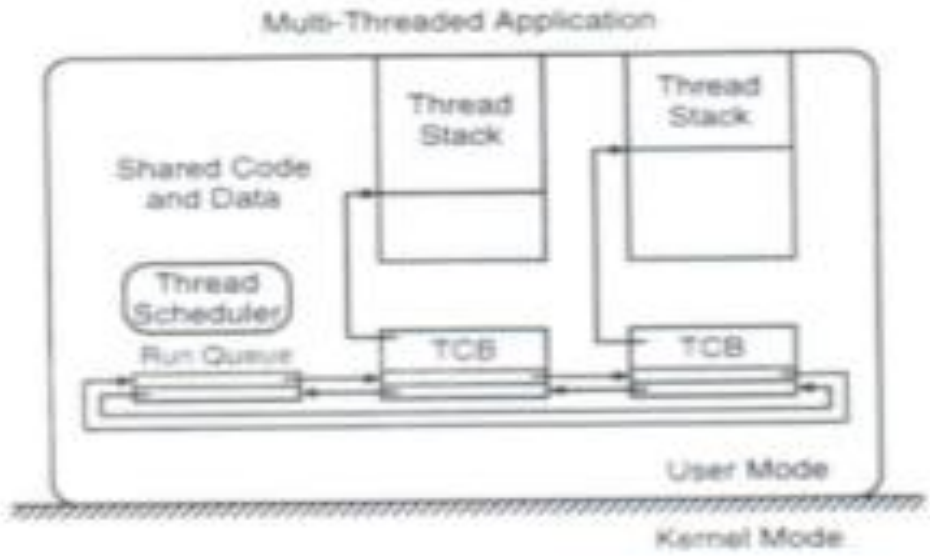


Figure 4: Example of user-level thread implementation.

Caracteristici threaduri utilizator

- nu apeleaza serviciile kernel pt creare si context switch => aceste operatii sunt f. rapide (nu se trece granita user-kernel si nu e nevoie de verificare parametrilor apelului sistem de catre kernel)
- aplicatiile pot furniza propriul planificator (*thread scheduler*) customizat cf unor cerinte specifice
- nu necesita nici un fel de suport explicit din partea kernelului; procesul e privit ca un procesor virtual pt. threaduri
- fiind mult mai rapide decat threadurile kernel, de regula threadurile user se implementeaza deasupra threadurilor kernel

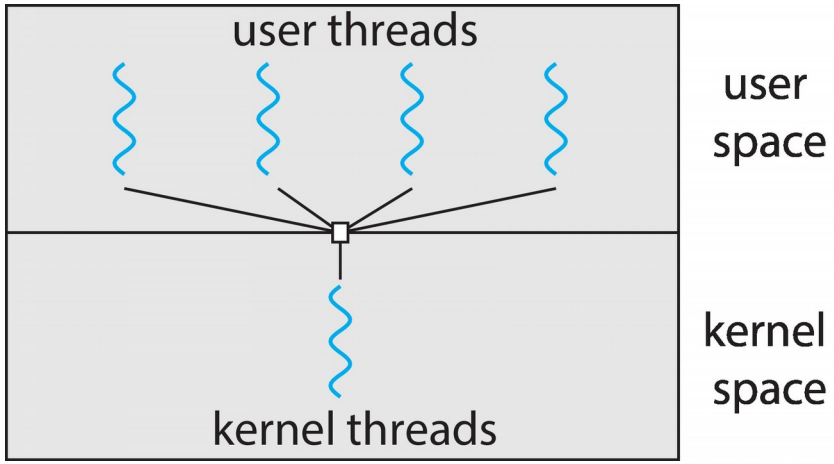
=> planificatorul de threaduri user trateaza threadurile kernel ca pe procesoare virtuale si multiplexeaza mai multe threaduri user pe unul sau mai multe threaduri kernel

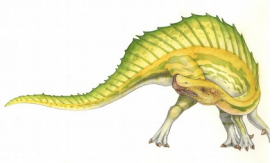
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

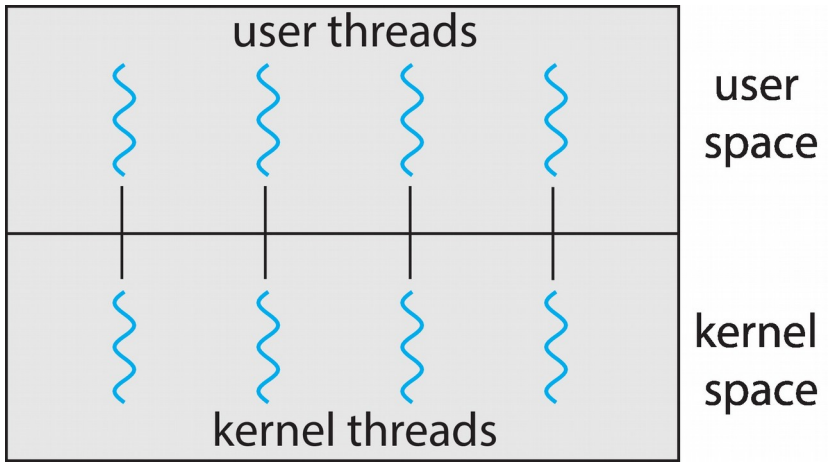
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**





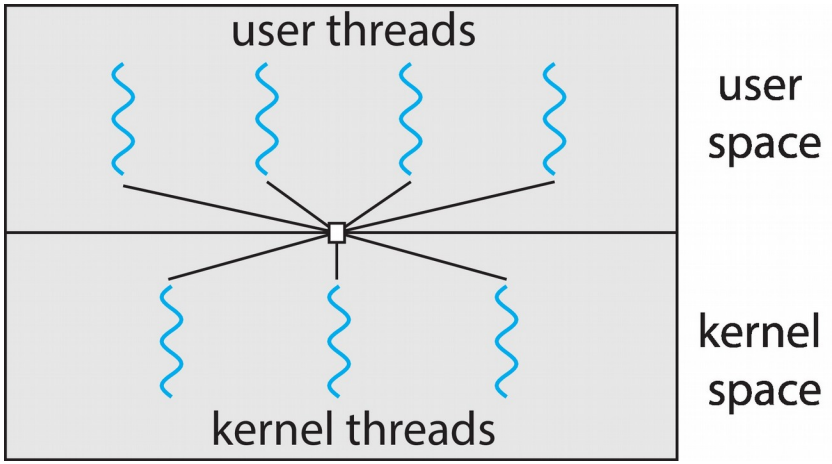
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



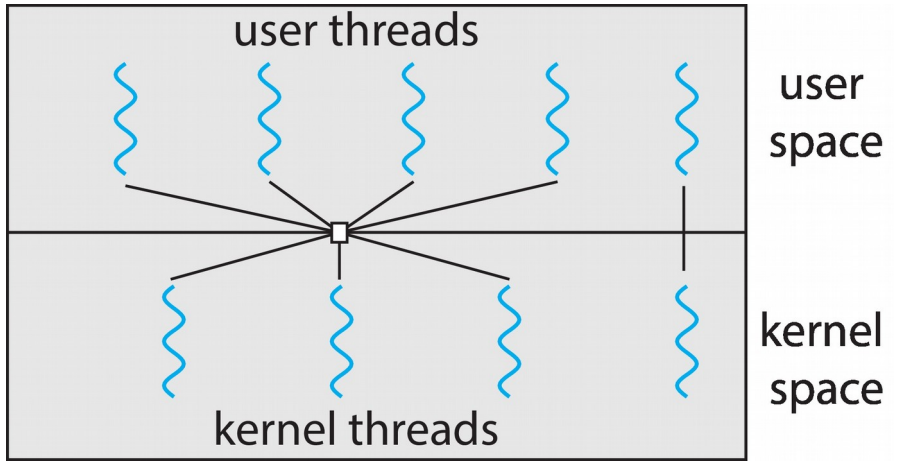
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



Probleme comune threadurilor kernel si user

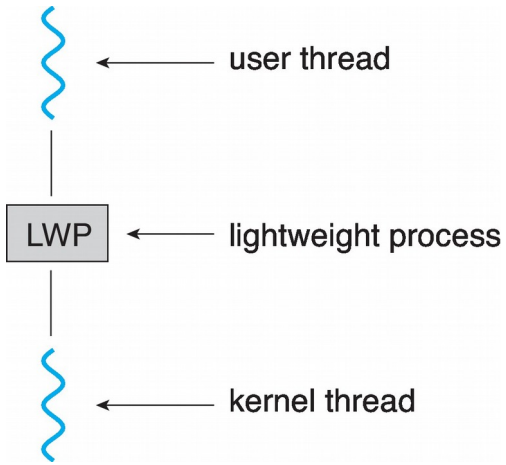
- reentranta: multe apeluri de biblioteca sunt ne-reentrante (apelurile reentrante sunt desemnate in paginile de manual ca fiind “thread safe”)
  - ex: trimiterea unui mesaj in retea in 2 pasi: (a) asamblare mesaj in buffer + (b) transmisie (syscall)
  - pierderea procesorului intre (a) si (b) poate conduce la suprascrierea bufferului
  - alte ex: errno, malloc, apeluri stdio
- tratarea semnalelor
  - ex: un thread trateaza un semnal in vreme ce alt thread vrea ca semnalul respectiv sa termine aplicatia (procesul, mai exact)
  - se poate intampla daca se folosesc apeluri de biblioteca si runtime user impreuna
  - problema deriva din faptul ca semnalele sunt definite per proces si nu per thread

Dezavantaje threaduri utilizator

- determinate de faptul ca existenta lor e necunoscuta kernelului
- (1) daca un thread user executa un apel sistem care se blocheaza in kernel, planificatorul kernel (agnostic cu privire la threadurile user) considera ca intreg procesul s-a blocat si aloca CPU altui proces (sau kernel thread), *chiar daca procesul curent mai are si alte threaduri user gata de rulare si nu si-a consumat toata cuanta de timp CPU*
- (2) daca un thread user comite page fault (acces la pagina de memorie inexistenta/nealocata) => acelasi efect ca mai sus, planificatorul kernel alege alt processor/kernel thread in vreme ce pagina de memorie e adusa de pe disc => aplicatia ruleaza cu mai putine CPU decat e necesar
- (3) nefiind constient de existenta threadurilor user, kernelul poate lua procesorul unui thread user care detine un spinlock pe care nu l-a eliberat => scadere dramatica de performanta pt aplicatiile care folosesc threaduri user paralele
  - efectul e si mai dramatic daca schedulerul kernel alege sa ruleze alte threaduri care vor sa obtina acelasi spinlock
- problema de fond: lipsa de coordonare intre schedulerul kernel si implementarea pachetului de threaduri user

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads



Scheduler activations

- metoda de coordonare kernel – pachet de threaduri utilizator
- model: aplicatia ruleaza pe un multiprocesor virtual
  - exista apeluri sistem pt a suplimenta/diminua nr de procesoare virtuale alocate de kernel (“adauga CPU”, “acest CPU este idle”)
  - kernelul decide daca onoreaza cererea sau nu
- scheduler activation e approximarea unui kernel thread
  - are stive kernel si user
  - ofera context de executie pt un thread user
  - in plus, ofera conceptul de *upcall* pt evenimente de mai multe tipuri
  - fiecare upcall creeaza o noua activare (optimizare: folosirea vechilor activari)

Tipuri de evenimente upcall

- adauga procesor – consecinta este executia unui thread user
- procesor preemptat – adauga threadul user care se executa in activarea care a pierdut CPU in coada de threaduri gata de rulare
- activare blocata – activarea s-a blocat (eg, apel sistem blocant) si nu mai utilizeaza procesorul
- activare deblocata – pune in lista gata de rulare threadul care se executa in contextul activarii blocate

Cum functioneaza scheduler activations

- la pornirea procesului
  - kernelul aloca o activare + notifica aplicatia (upcall “adauga CPU”) dupa ce i-a asigurat un CPU
  - sistemul de gestiune al threadurilor user primeste notificarea si foloseste noua activare drept context de executie pt initializarea sa si a threadului *main* (ulterior se pot crea noi threaduri si cere noi procesoare)
- la cerere de suplimentare a concurentei (la adaugarea CPU sau pt. ca un thread se blocheaza)
  - kernelul salveaza starea threadului user in activarea curenta
  - aloca o noua activare si cheama aplicatia in contextul noii activari
- la blocarea unui thread user in kernel
  - nu se continua in kernel la deblocare
  - se creeaza o noua activare
  - se notifica aplicatia
  - schedulerul user copiaza starea threadului blocat din vechea activare si notifica kernelul ca vechea activare poate fi refolosita

Cum functioneaza scheduler activations

- cand un thread user care detine un spinlock pierde procesorul
  - kernelul genereaza un upcall
  - sistemul de gestiune a threadurilor verifica daca threadul e in sectiune critica
  - in caz afirmativ, threadul este continuat temporar printr-un context switch in user space
  - la iesirea din sectiunea critica, se da controlul inapoi upcall-ului tot prin context switch in user space

Exemple de pachete de threaduri user

- Solaris si POSIX
- threadurile Solaris
  - exista si la nivel de kernel (s.n. LWPs, sunt multiplexate pe CPU existente) si la nivel de utilizator
  - pachetul de threaduri (*libthread*) planifica threadurile user pe o colectie de LWPs
  - uzual, threadurile user sunt create nelegate (*unbound*), adica se pot muta intre LWP-uri si asa se si folosesc in general
  - daca un thread user e asignat unui LWP se spune ca e legat (*bound*)
  - controlul kernelului asupra threadurilor user se exercita prin bound threads cu ajutorul setarii prioritatii LWP-ului asociat (de ex, thread user pt stream audio cu prioritate mare in aplicatie multimedia)
  - biblioteca de threaduri asigura ca exista suficiente LWP-uri active pt ca procesul sa poata continua
  - daca un proces determina toate LWP-urile sale sa se blocheze, biblioteca va crea LWP-uri aditionale pt. ca threadurile neblocate sa poata rula
  - daca LWP-urile sunt idle prea mult timp, biblioteca le da inapoi kernelului pt a fi folosite de alte aplicatii

Pachete de threaduri (cont'd)

- exista apel de setare explicita a nr de LWP-uri la un nou nivel
  - thr_setconcurrency(int new_level)
- mecanisme IPC: mutex si variabile conditie
- implementeaza TLS (Thread-Local Storage), capacitate privata de stocare a variabilelor in afara stivei
  - variabile locale nepartajate, "statice" (globale pt threadul respectiv)
  - declarate cu #pragma, de ex #pragma unshared errno;
- threaduri POSIX (pthreads)
  - standard IEEE, Portable Operating System Interface (e o specificatie, nu o implementare)
  - asemanatoare threadurilor Solaris

Exemplu de pachete de threaduri user

```
int thr_create(void *stack_base, size_t stack_size,
               void *(*function)(void *), void *arg, long flags)
void thr_exit(void *status)
int thr_join(thread_t wait_for, thread_t *joiner_id, void *status)
int mutex_init(mutex_t *mutex, int type, void *arg)
int mutex_lock(mutex_t *mutex)
int mtex_unlock(mutex_t *mutex);
int cond_init(cond_t *cond, int type, int arg)
int cond_wait(cond_t *cond, mutex_t *mutex)
int cond_signal(cond_t *cond)
int cond_broadcast(cond_t *cond)
```

Figure 5: Some commonly used UI thread operations.

```
int pthread_create(pthread_t *thread_pointer,
                   pthread_attr_t *attributes,
                    void *(*function)(void *), void *arg)
void pthread_exit(void *status)
int pthread_join(pthread_t thread, void **status)
void pthread_mutex_init(pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr)
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr)
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_signal(pthread_cond_t *cond)
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Figure 6: Some commonly used Pthreads operations.

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define ITEMS           8
#define BUFFER_SIZE     4
int buffer[BUFFER_SIZE];

void *producer(void *);
void *consumer(void *);

int spaces;
int items;
int tail;
pthread_cond_t space;
pthread_cond_t item;
pthread_mutex_t buffer_mutex;

char *cmd;

int
main(int argc, char *argv[])
{
    pthread_t producer_thread;
    pthread_t consumer_thread;
    void *thread_return;
    int result;

    cmd = argv[0];

    spaces = BUFFER_SIZE;
    items = 0;
    pthread_mutex_init(&buffer_mutex, NULL);
    pthread_cond_init(&space, NULL);
    pthread_cond_init(&item, NULL);

    if (pthread_create(&producer_thread, NULL, producer, NULL) ||
        pthread_create(&consumer_thread, NULL, consumer, NULL)) {
        fprintf(stderr, "%s: can't make thread\n", cmd);
        exit(1);
    }
    if (pthread_join(producer_thread, &thread_return)) {
        fprintf(stderr, "%s: can't join producer\n", cmd);
        exit(1);
    } else {
        printf("producer returns with %d\n", (int)thread_return);
    }
    if (pthread_join(consumer_thread, &thread_return)) {
        fprintf(stderr, "%s: can't join consumer\n", cmd);
        exit(1);
    } else {
        printf("consumer returns with %d\n", (int)thread_return);
    }
    exit(0);
}
```

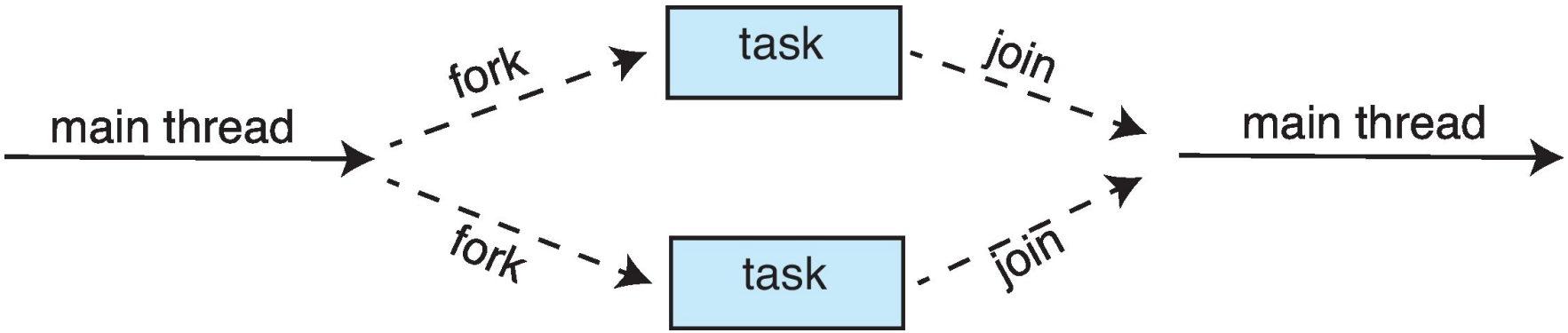
Figure 7: Main body of producer/consumer code.

```
void *
producer(void *arg)
{
    int i;

    for (i = 0; i < ITEMS; i++) {
        pthread_mutex_lock(&buffer_mutex);
        while (spaces == 0) {
            pthread_cond_wait(&space, &buffer_mutex);
        }
        buffer[(tail + items) % BUFFER_SIZE] = i;
        items += 1;
        spaces -= 1;
        fprintf(stderr, "producer: puts %d\n", i);
        pthread_mutex_unlock(&buffer_mutex);
        pthread_cond_signal(&item);
    }
    pthread_exit(0);
    return (void *)0;
}

void *
consumer(void *arg)
{
    int i, b;

    for (i = 0; i < ITEMS; i++) {
        pthread_mutex_lock(&buffer_mutex);
        while (items == 0) {
            pthread_cond_wait(&item, &buffer_mutex);
        }
        b = buffer[tail % BUFFER_SIZE];
        tail += 1;
        items -= 1;
        spaces += 1;
        fprintf(stderr, "consumer: gets %d\n", b);
        pthread_mutex_unlock(&buffer_mutex);
        pthread_cond_signal(&space);
    }
    pthread_exit(0);
    return (void *)0;
}
```

Figure 8: Code for producer and consumer threads.

Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.

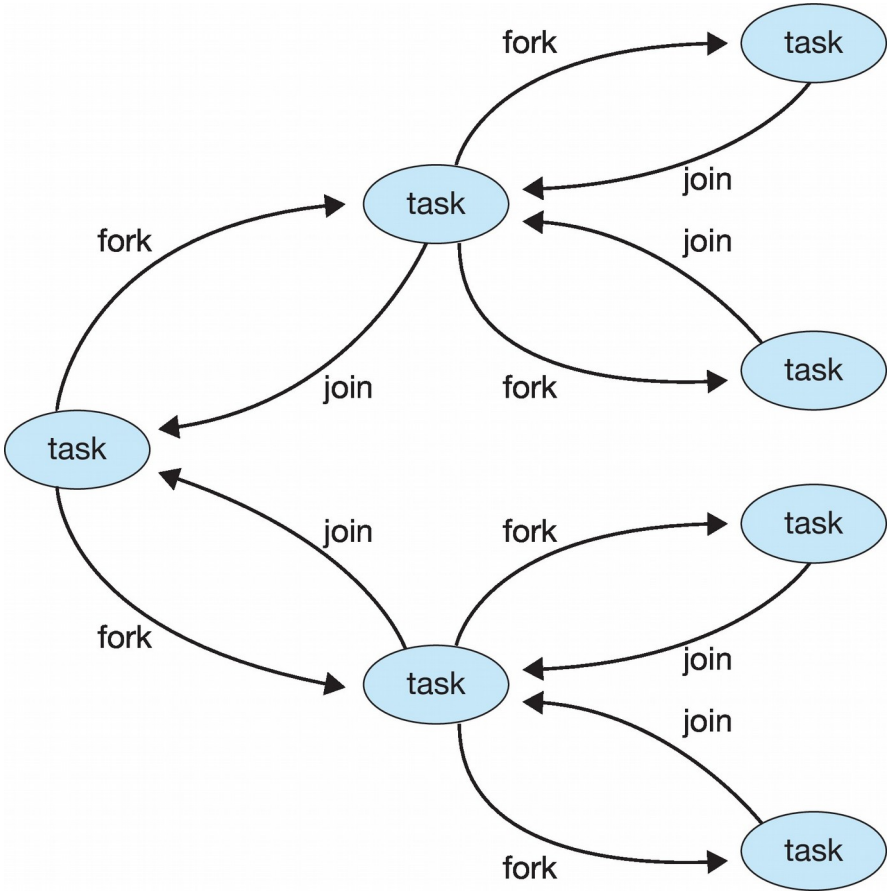


Fork-Join Parallelism

- General algorithm for fork-join strategy:

```
Task(problem)
   if problem is small enough
     solve the problem directly
   else
     subtask1 = fork(new Task(subset of problem)
     subtask2 = fork(new Task(subset of problem)

     result1 = join(subtask1)
     result2 = join(subtask2)

     return combined results
```

Fork-Join Parallelism



Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e,Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Tratarea semnalelor in pachetele de threaduri

- fiecare thread are propria masca de semnale
- toate threadurile unui proces partajeaza handlerele de semnal
- cand SIG_IGN/SIG_DFL sunt setate, se aplica tuturor threadurilor procesului
- semnalele de tip trap (sincrone cu executia threadului, ex SIGFPE, SIGSEGV) executate exclusiv de threadul care le-a generat => mai multe threaduri pot genera si trata acelasi tip de semnal simultan
- semnalele de tip interupere (generate asincron cu executia threadului, ex SIGIO, SIGINT) pot fi tratate de orice thread care nu mascheaza/blocheaza semnalul respectiv (in cazul mai multor astfel de threaduri, se alege unul dintre ele pt tratarea semnalului)
- daca toate threadurile au mascat un anume semnal, se asteapta dupa primul thread care deblocheaza tratarea semnalului respectiv

Tratarea semnalelor in pachetele de threaduri (cont'd)

- thread_kill: trimite un semnal care un thread din acelasi proces
  - semnalul e de tip trap, tratat doar de threadul caruia ii este adresat
  - nu se pot trimite semnale unui *anume* thread dintr-un *alt* proces
- threadurile Solaris implementeaza un nou semnal, SIGWAITING (SIG_IGN implicit)
  - trimis procesului cand toate LWP-urile sunt blocate indefinit in asteptarea unui eveniment extern
  - poate fi folosit pt crearea de noi LWP-uri pt a evita blocarea intregului proces
  - idee similara cu scheduler activations, dar coarse-grain (nu merge pt short-term blocking, doar pt evenimente de tip page faults, filesystem I/O)

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e., **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)