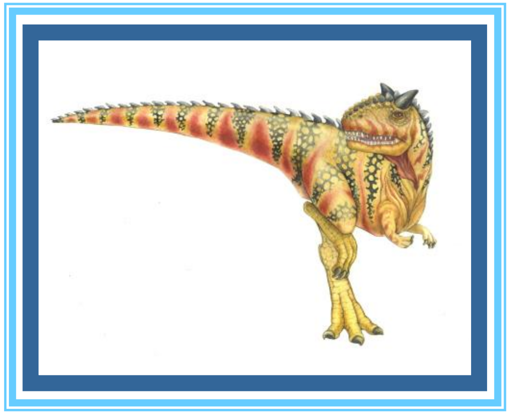
Chapter 4: Threads & Concurrency



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 designing 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

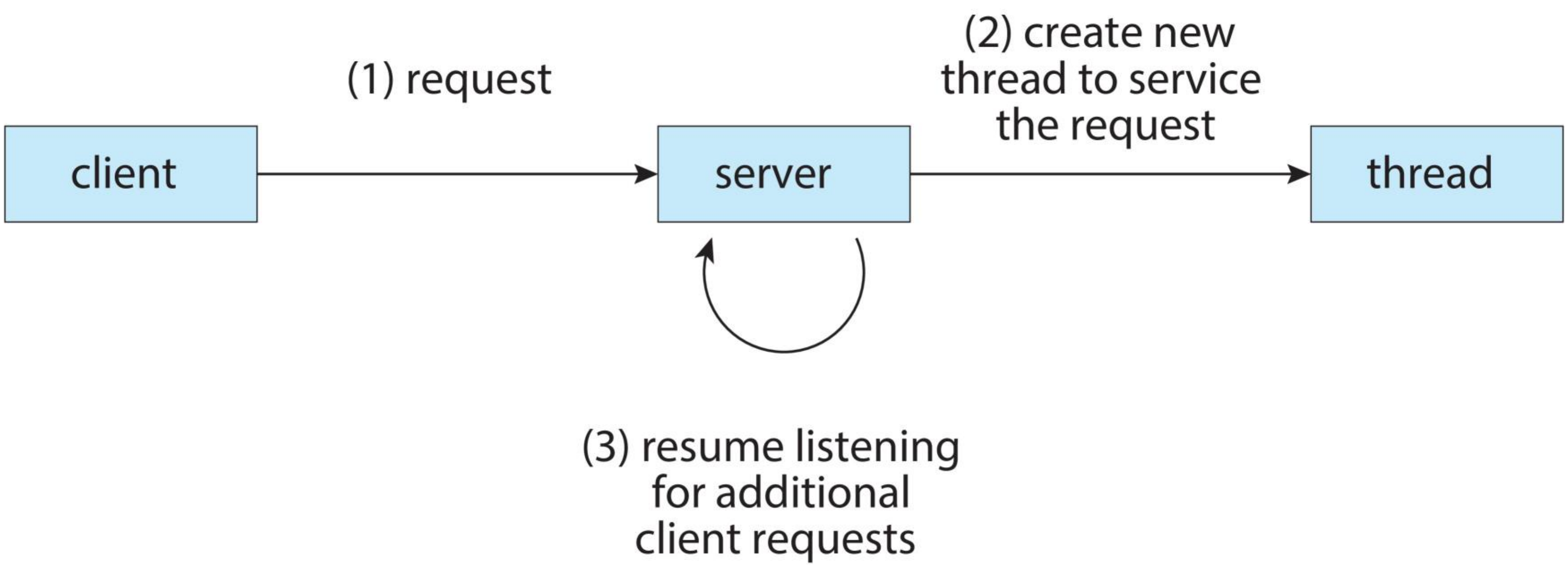
Aplicatii concurente

- 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
  - activitati concurente: verificare/procesare input tastatura, mouse si , retea, afisare bitmap-uri pe ecran
  - toate aceste activitati sunt codate in module separate, cu date private si comunica intre ele prin date partajate

Aplicatii concurente (cont.)

- 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

Multithreaded Server Architecture



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
  - IPC-ul

Crearea proceselor

- contextul unui proces e stufos (stocat in PCB in kernel)
- include starea completa 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")

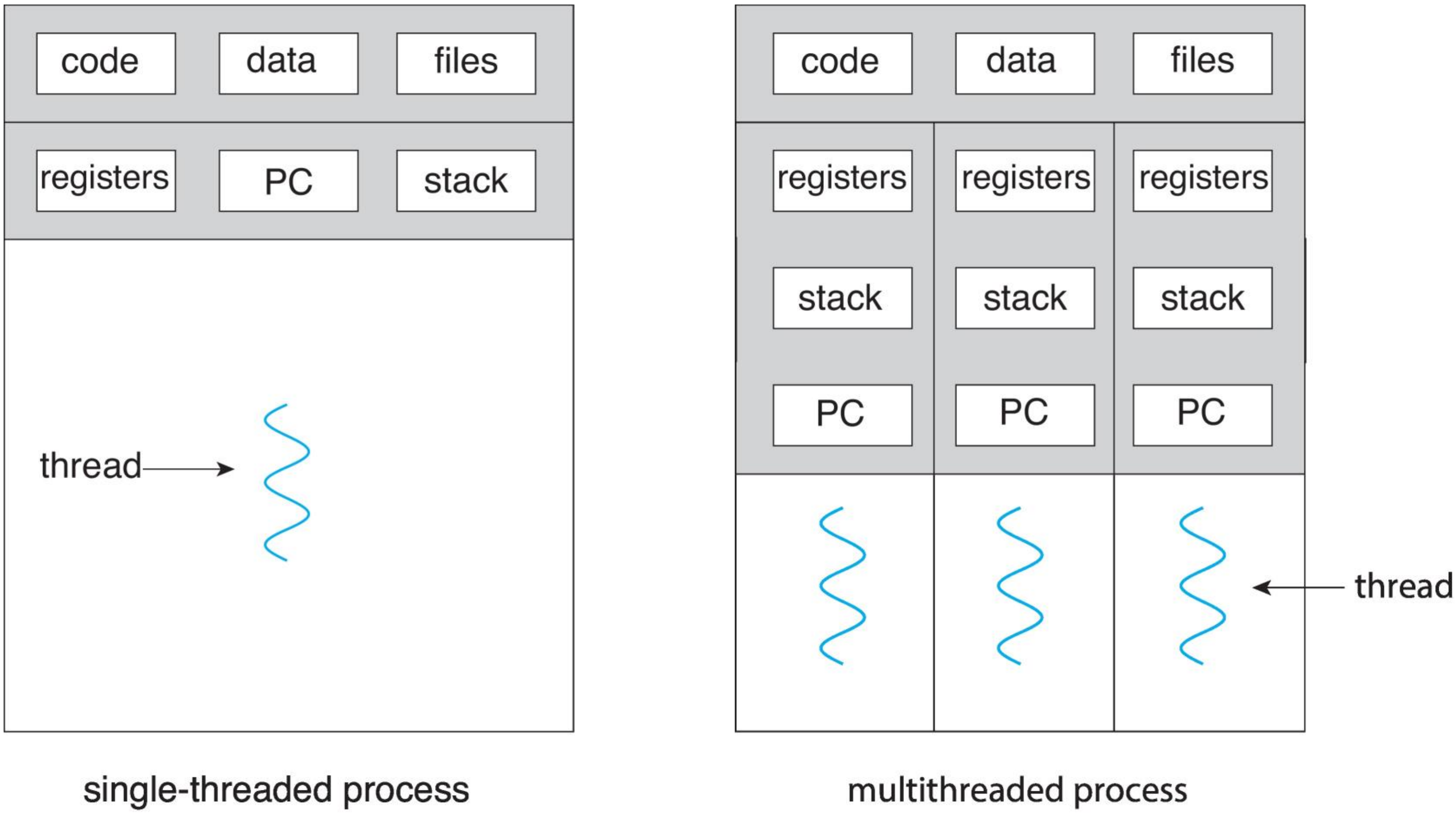
Context-switch

- 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

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)

Single and Multithreaded Processes



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

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 (cont.)

- (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*)

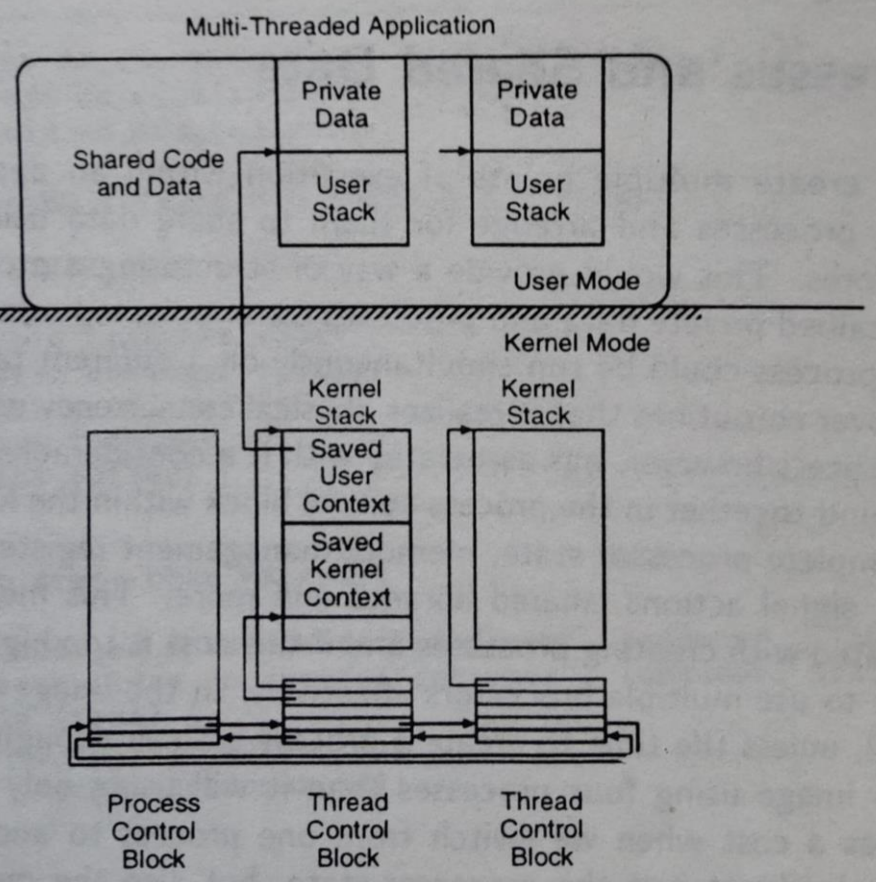


Figure 3: Example of kernel thread implementation.

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 restul contextului 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 kernel

- 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 threadurilor kernel

- 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 threadurilor kernel

- 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

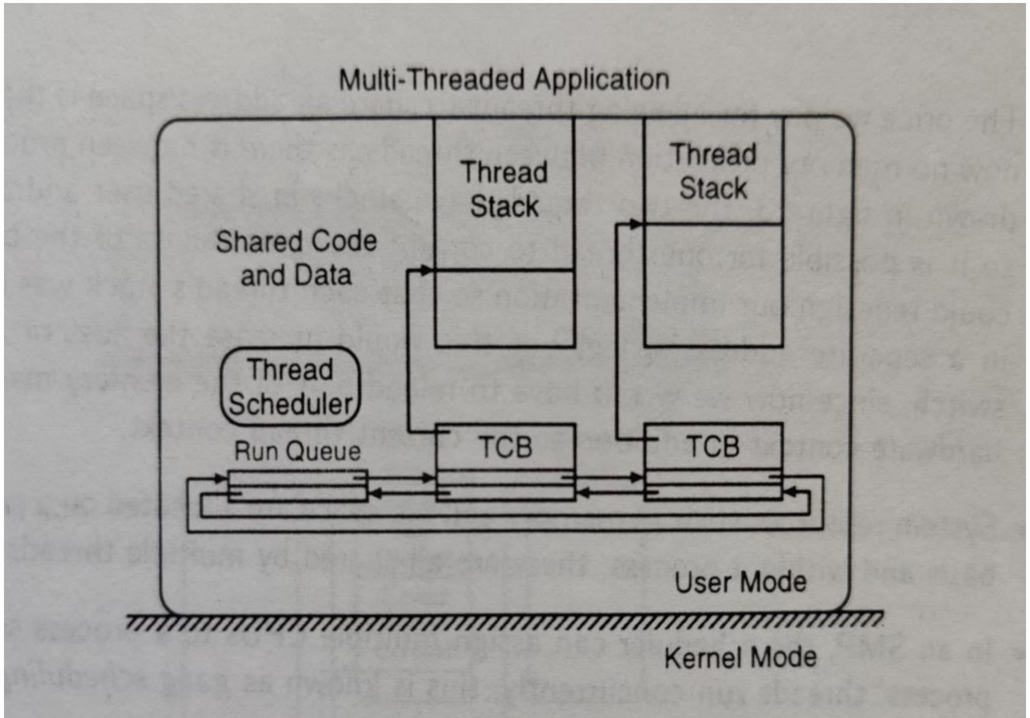


Figure 4: Example of user-level thread implementation.

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

Caracteristici threaduri utilizator

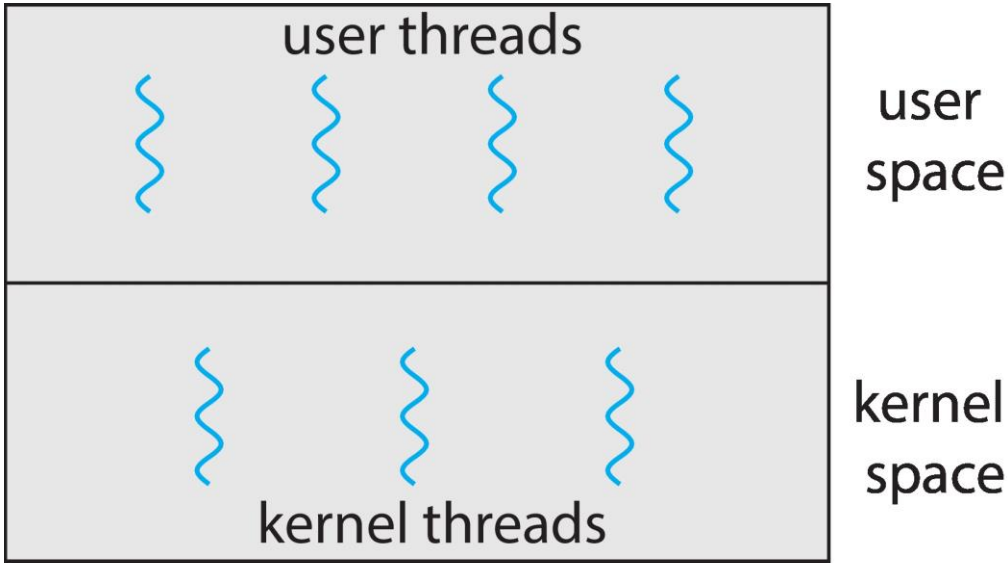
- nu apeleaza serviciile kernel pt creare si context switch => aceste operatii sunt f. rapide (nu se trece granita user-kernel si nu e nevoie de verificarea parametrilor apelului sistem de catre kernel)
- aplicatiile pot furniza propriul planificator (*thread scheduler*) customizat cf unor cerinte specifice
- 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

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads

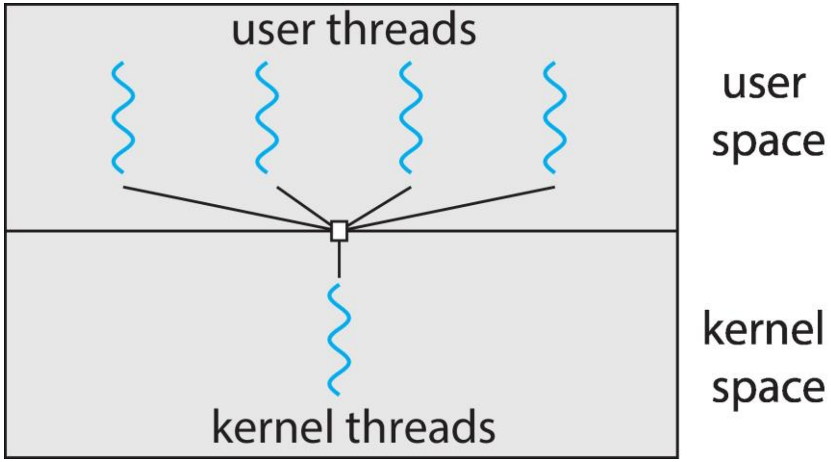


Multithreading Models

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

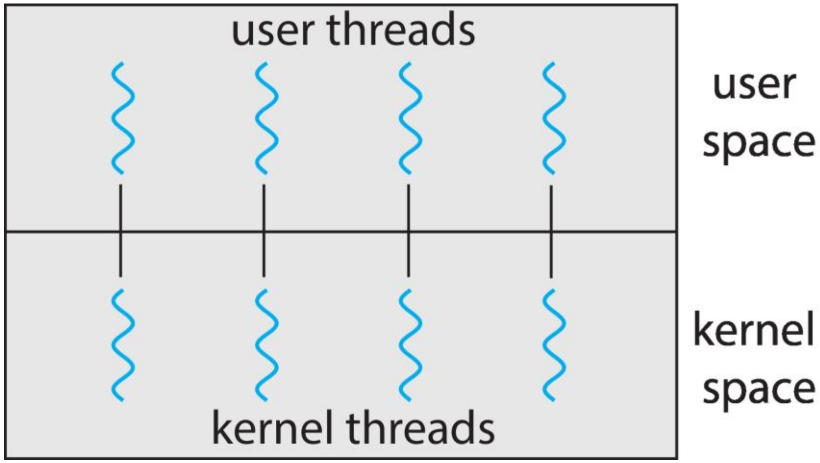
Many-to-One

- 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**



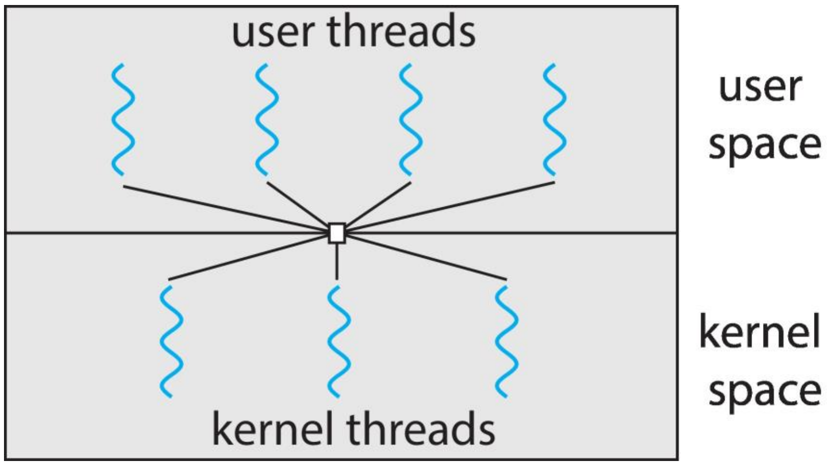
One-to-One

- 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



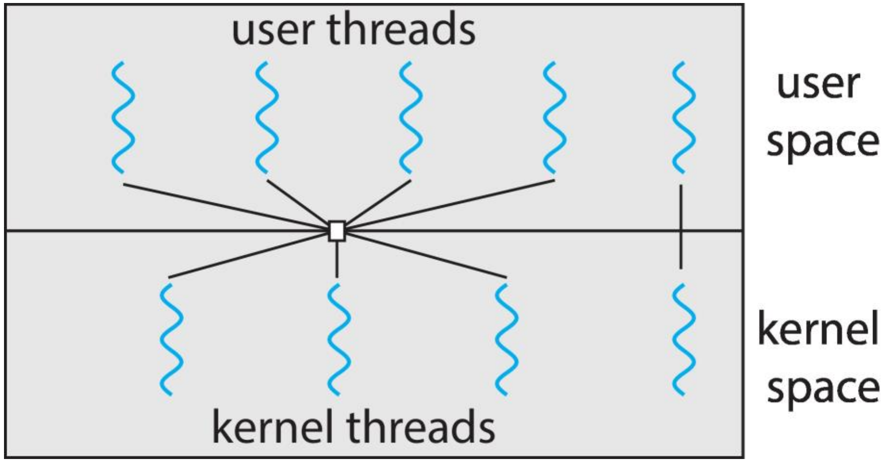
Many-to-Many Model

- 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 k. si u.

- 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) thread user executa apel sistem blocant in kernel, planificatorul kernel (agnostic cu privire la threadurile user) considera intreg procesul 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) thread user comite *page fault* (acces la pagina de memorie inexistenta/nealocata) => acelasi efect ca mai sus, planificatorul alege alt procesor/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 in paralel
  - efectul e si mai dramatic daca schedulerul alege sa ruleze alte threaduri care vor sa obtina acelasi spinlock
- problema de fond: lipsa de coordonare intre schedulerul si implementarea pachetului de threaduri user

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 aproximarea 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

Functionarea scheduler activations

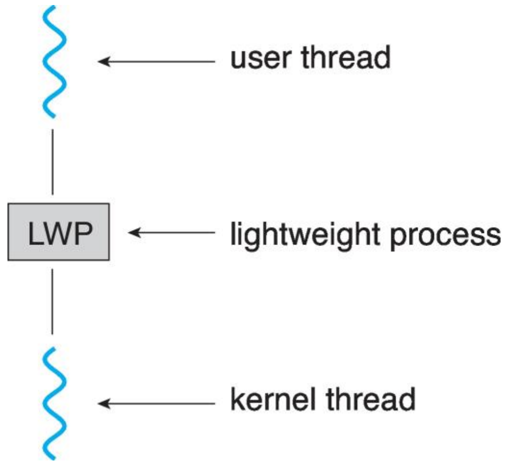
- la pornirea procesului
  - kernelul aloca o activare + notifica aplicatia (*upcall* “adauga CPU”) dupa ce i-a asigurat un CPU
  - sistemul de gestiune al threadurilor user primeste notificarea si foloseste noua activare drept context de executie pt initializarea sa si a threadului *main* (ulterior 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

Functionarea scheduler activations (cont)

- 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
- 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

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



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Exemple de pachete de threaduri user

- Solaris si POSIX
- threaduri 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)

Exemple de pachete de threaduri user

- threaduri Solaris
- ….
  - 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

Exemple de pachete de threaduri user

- 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

Solaris threads

```
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.

Tratarea semnalelor

- 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 (cont.)

- *thread_kill*: trimite un semnal catre 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)

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)

POSIX threads

```
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.

Pthreads Example

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  /* set the default attributes of the thread */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}
```

Pthreads Example (Cont.)

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

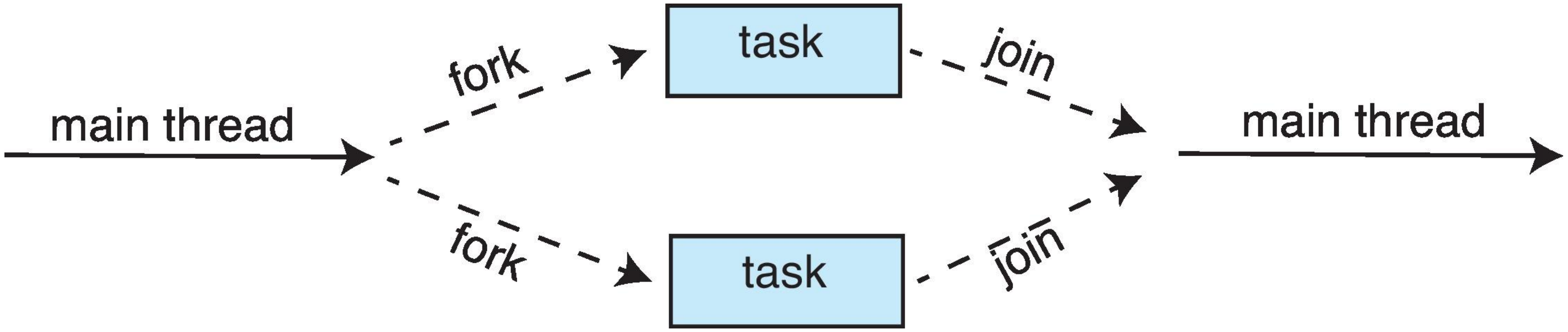
Thread Pools

- 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.
   */
}
```

Fork-Join Parallelism

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

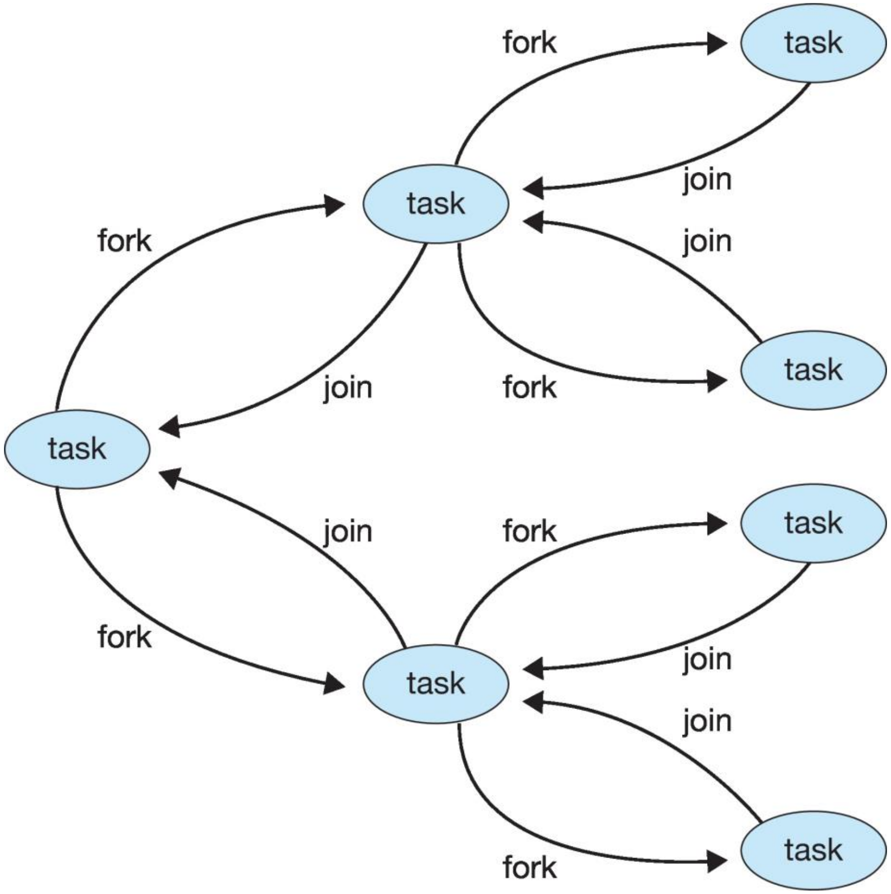


Fork-Join Parallelism

- 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



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

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

Thread-Local Storage

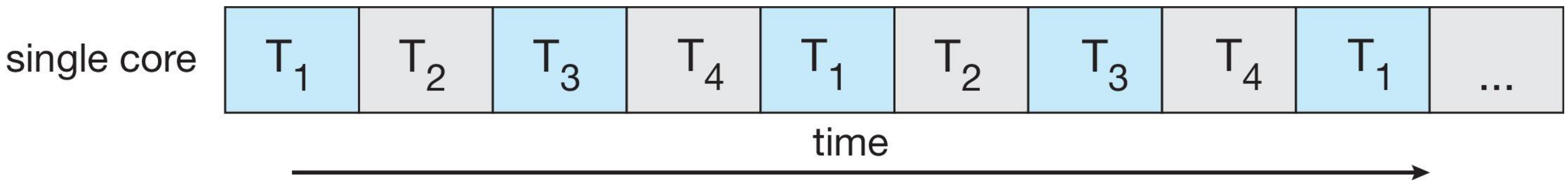
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

Multicore Programming

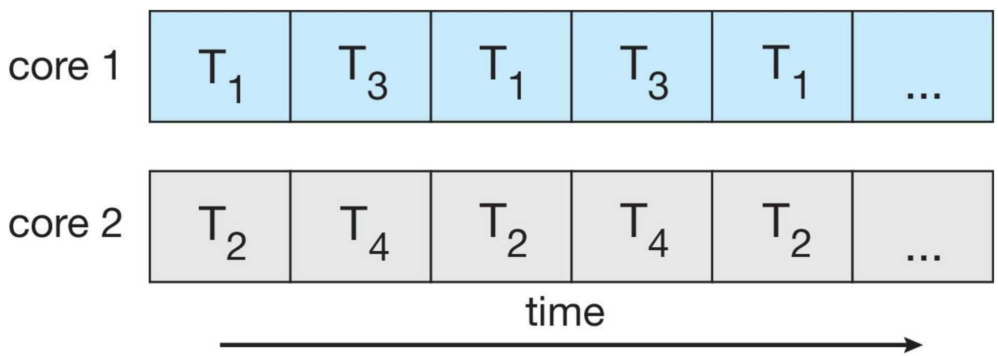
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
-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

Concurrency vs. Parallelism

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



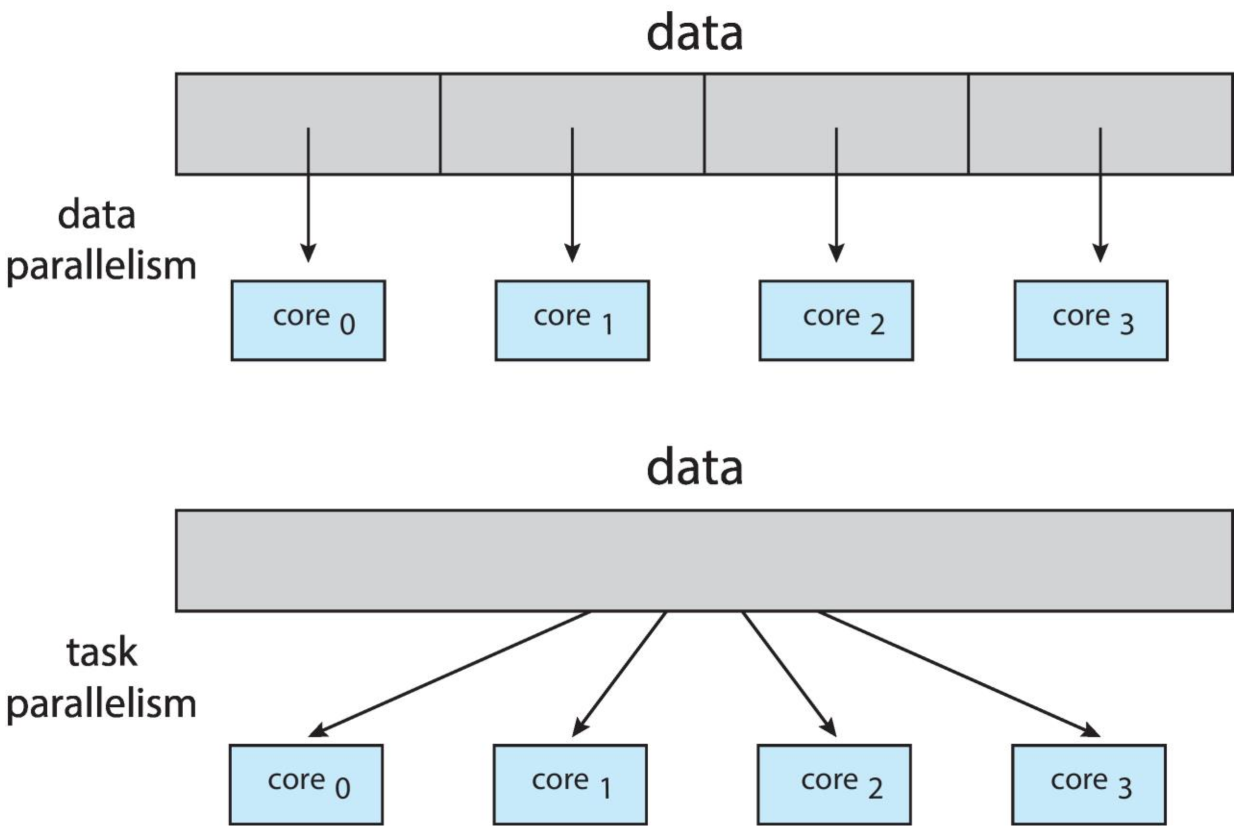
- **Parallelism on a multi-core system:**



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

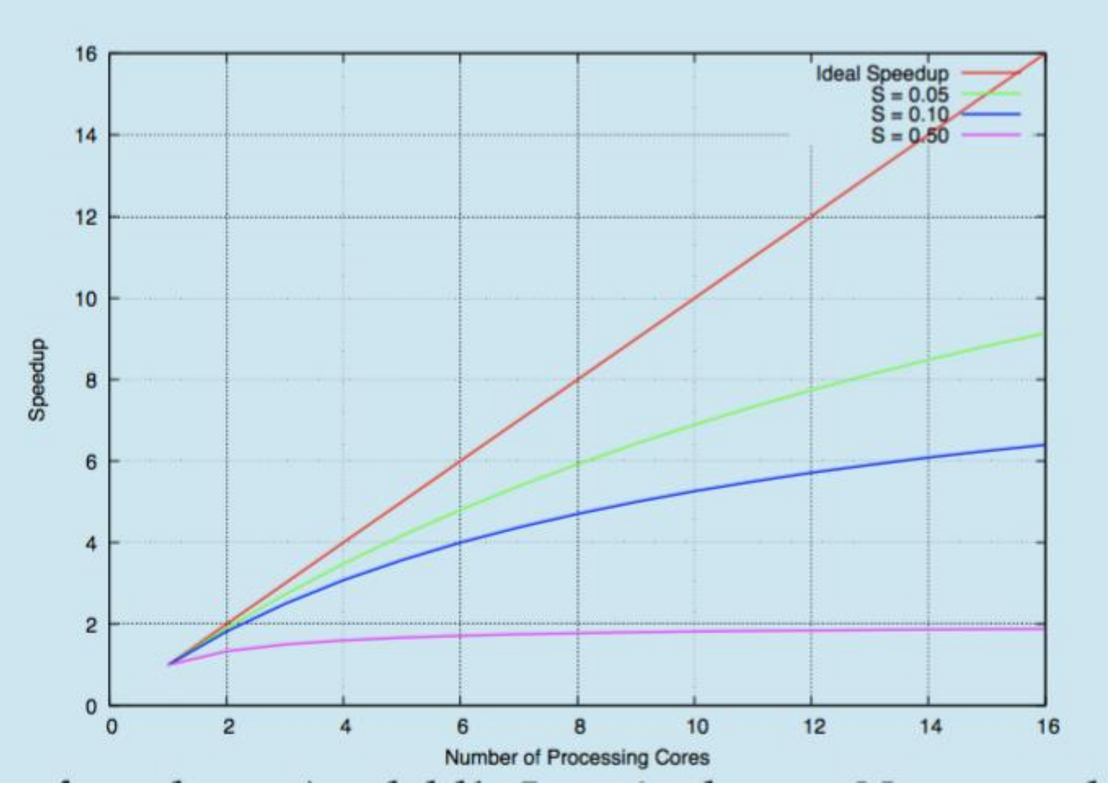
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- 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

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;

  Param = atoi(argv[1]);
  /* create the thread */
  ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

   /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

  /* close the thread handle */
  CloseHandle(ThreadHandle);

  printf("sum = %d\n",Sum);
}
```

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

  - Standard practice is to implement Runnable interface

Java Threads

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Java Executor Framework

- Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:

```
public interface Executor
{
   void execute(Runnable command);
}
```

- The Executor is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```

Java Executor Framework

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}
```

Java Executor Framework (Cont.)

```
public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

Java Thread Pools

- Three factory methods for creating thread pools in Executors class:
  - `static ExecutorService newSingleThreadExecutor()`
  - `static ExecutorService newFixedThreadPool(int size)`
  - `static ExecutorService newCachedThreadPool()`

Java Thread Pools (Cont.)

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
public static void main(String[] args) {
   int numTasks = Integer.parseInt(args[0].trim());

   /* Create the thread pool */
   ExecutorService pool = Executors.newCachedThreadPool();

   /* Run each task using a thread in the pool */
   for (int i = 0; i < numTasks; i++)
      pool.execute(new Task());

   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```

Fork-Join Parallelism in Java

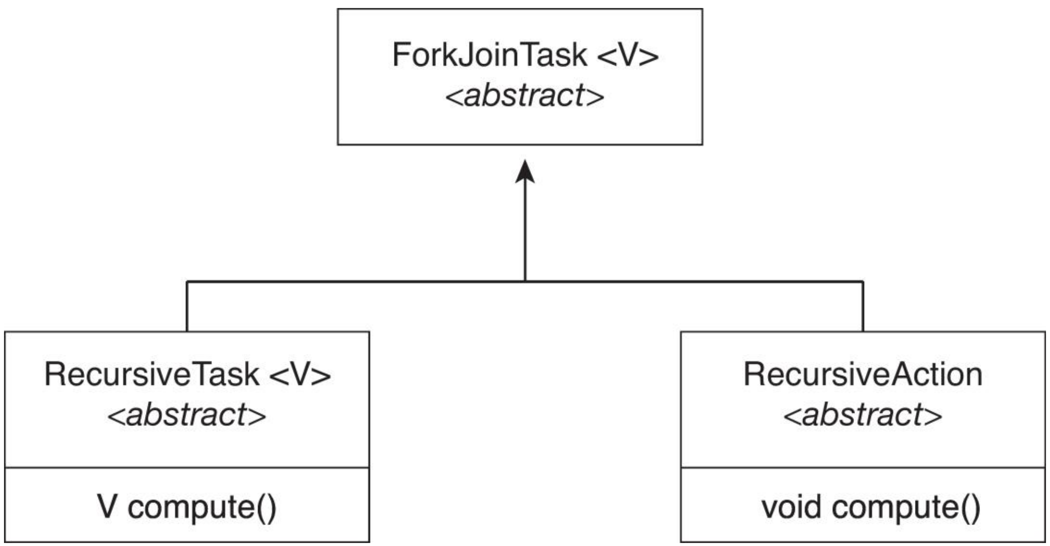
```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

Fork-Join Parallelism in Java

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

Fork-Join Parallelism in Java

- The **`ForkJoinTask`** is an abstract base class
- **`RecursiveTask`** and **`RecursiveAction`** classes extend **`ForkJoinTask`**
- **`RecursiveTask`** returns a result (via the return value from the **`compute()`** method)
- **`RecursiveAction`** does not return a result



OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

`#pragma omp parallel`

Create as many threads as there are cores

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

- Run the for loop in parallel

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

Grand Central Dispatch

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-C languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }” :

```
^{ printf("I am a block"); }
```

- Blocks placed in dispatch queue
    - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch

- Two types of dispatch queues:
  - **serial** – blocks removed in FIFO order, queue is per process, called **main queue**
    - Programmers can create additional serial queues within program
  - **concurrent** – removed in FIFO order but several may be removed at a time
    - Four system wide queues divided by quality of service:
      - `QOS_CLASS_USER_INTERACTIVE`
      - `QOS_CLASS_USER_INITIATED`
      - `QOS_CLASS_USER_UTILITY`
      - `QOS_CLASS_USER_BACKGROUND`

Grand Central Dispatch

- For the Swift language a task is defined as a closure – similar to a block, minus the caret
- Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- The same for loop written using TBB with **`parallel_for`** statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

Thread Cancellation in Java

- Deferred cancellation uses the **`interrupt()`** method, which sets the interrupted status of a thread.

```
Thread worker;

  . . .

/* set the interruption status of the thread */
worker.interrupt()
```

- A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

Operating System Examples

- Windows Threads
- Linux Threads

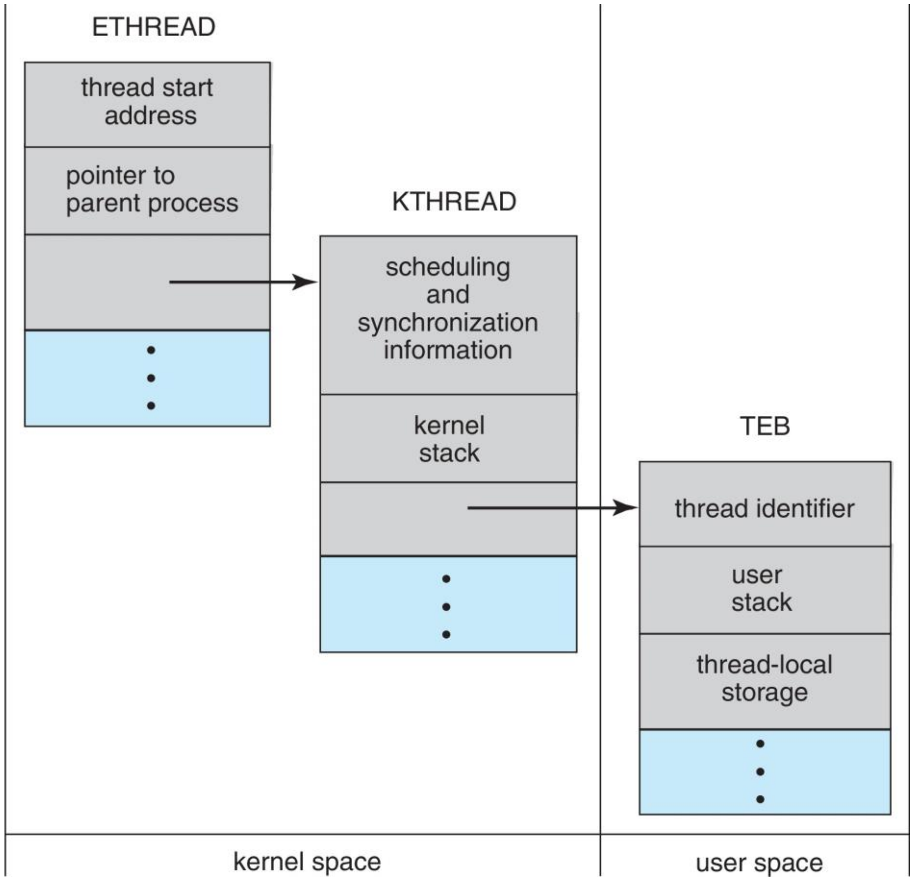
Windows Threads

- Windows API – primary API for Windows applications
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



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)

End of Chapter 4