CURS 5

Proces = un singur punct de executie in aplicatie (o sg instanta in rulare a aplicatiei)

pot exista mai multe puncte de executie (procese) mai ales pe multiprocesoare

Crearea de puncte de executie multiple se face cu procese care PARTAJEAZA DATE (exemplu: memorie partajata cu acces protejat de SEMAFOARE / LOCKS).

Observatie! Procesele au date private (datorita protectiei Memory Manag Unit) si pot comunica intre ele prin IPC (shared memory).

Crearea proceselor

Contextul unui proces e stocat in Process Control Block (in kernel) si include date precum (fapt care duce la un cost semnificativ de creare a acestuia):

starea complete a CPU

registre de gestiune a mem

tabele de pag de mem

descriptori de fisiere

Pot aparea situatii in care din cauza unui cost prea mare pentru crearea unor noi procese, folosirea unui singur proces sa fie mai eficienta.

Context-switch

Din cauza volumului mare de info din PCB => salvarea unui context al unui proces si incarcarea unuia nou poate implica un cost semnificativ

Exemplu! Secventa de evenimente producator-consumator cu zero buffering

Producatorul => prod element si se blocheaza
Sistemul => Context-switch si aduce consumatorul pe procesor
Consumatorul => consuma elementul si se blocheaza
Sistemul => Context-switch si aduce pro dinapoi pt a face un nou element

Deci, in functie de viteza cu care se face context-switchului vom avea diferente semnificative in timpul de rulare.

Programarea multicore

- ➔ mai dificila deoarece programatorul trebuie sa tina cont de mai multe concepte precum Balance, Data splitting, Data dependency, iar testarea si debuggingul se realizeaza mai dificil
- ➔ mai rapida ca timp (se utilizeaza paralelismul)
- ➔ data paralelism (se distribuie datele)
- ➔ task paralelism (se distribuie taskuri => threaduri)

Legea lui Amdahl

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

Legea lui Gustafson

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

Fire de executie (THREADURI)

Modalitate de a reduce costurile crearii noilor procese si astfel se elimina si un numar mare de context-switchuri.

Fiecare proces are o stiva proprie si o copie proprie al contextului de executie a aplicatiei proprie.

Un thread = fir de executie = punct de executie cu context redus in timpul programului = proces usor = lightweight process

Caracteristici

Ruleaza secvential (au program counter si stiva proprie)
Pot crea alte threaduri
Pot executa apeluri de sistem
Multiplexeaza accesul la CPU ca si procesele
Threadul e pt proces cum e procesul pt procesor (procesul e un procesor virtual pe care ruleaza threadul)
Stari (la fel ca la procese): in rulare, gata de rulare, blocat, terminat
Modele de utilizare: cooperativ (filtrare de imagini) SAU master-slave (server) SAU pipeline (producator-consumator)
Datele partajate sunt datele globale din proces

Diferente fata de procese

Threadurile aceluiasi proces partajeaza spatiul de adresa al procesului (adica se pot distruge usor unele pe altele), dar si acelasi set de fisiere deschise, timere, semnale, etc.
Lipsa protectiei intre threaduri e inevitabila prin design dar nici nu e necesara sau de dorit (se doreste consumarea aceleasi resurse de exemplu)

Designul pachetelor de threaduri

= colectie de primitive (apeluri de biblioteca)
= se pot implementa in KERNEL sau in USER

1. Gestiunea threadurilor
    a. Creare (primeste stiva privata, prioritate de executie si un id (TID))
    b. Terminare (apel exit sau semnal de tip kill de la alt thread/proces)
    c. Primitive pt mecanisme de synch (datorita existentei datelor partajate) in principiu = mutex uri
2. Planificare
3. Probleme de reentranta (T1 si T2 executa concurent apeluri de sistem, iar unul esueaza e posibil ca celalalt sa creada ca el a esuat din cauza ca errno (eroarea produsa) e o variabila globala). Solutii: protejarea errno cu mutexuri, crearea unei copii private a errno pt fiecare thread

Threaduri KERNEL

➔ Separa campurile din PCB ce ajuta la crearea unui proces si le stocheaza in TCB => un proces e reprezentat in kernel de un PCB si un TCB
➔ Thread_create => aloca TCB -> aloca stive kernel si user -> le leaga la PCB

Costurile crearii threadurilor kernel

<<costul crearii unui proces (se initializeaza doar TCB si stivele, restul contextului fiind deja in PCB)

OBS: Context switch-ul intre 2 threaduri ale aceluiasi proces << context switchul dintre doua procese (PCB ul e acelasi), insa context switchul dintre 2 threaduri ale unor procese diferite au acelasi cost cu context switchul dintre cele 2 procese deoarece se schimba PCB ul.

Planificarea kernel threadurilor

Threadurile ruleaza asincron, iar accesul la datele partajate trebuie sincronizat. Planificatorul kernel alege un thread care trebuie sa ruleze (daca aplicatia are o planificare trebuie comunicata kernelului).

OBS: Daca un thread se blocheaza in kernel si cuanta de timp alocata procesului nu a expirat, planificatorul cauta in lista de TCB uri in thread gata de rulare DIN ACELASI PROCES si il acorda procesorului.

Dezavantaje

Thread_create => cost mare pt procese care fac multe astfel apeluri de sistem
Context switch-kernel threaduri => intrare/iesire din kernel mode => cost mare
Implementarea in kernel e inflexibila (nu se potrivesc pt orice aplicatie)

USER-LEVEL threads

Nu apeleaza serviciile kernel pt creare si context switch => viteza foarte buna (nu se mai face trecerea user-kernel)

Aplicatiile pot avea propriul planificator (thread scheduler) (se elimina inflexibilitatea de la kernel-threaduri).

Nu necesita suport explicit de la kernel=>procesul = procesor virtual pt threads

Fiind mult mai rapide decat threadurile kernel => user level threads se implementeaza deasupra

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

Modele multithreading:

Many to one: (ex: solaris green, gnu portable)
Mai multe threaduri user legate de un singur thread kernel
One to one: windows, linux
Fiecare user thread legat de un kernel thread
Many to one: Windows cu threadfiber (mai multe threaduri user la mai multe threaduri kernel)

Probleme comune threadurilor kernel si user

Reentranta (apelurile de bibilioteca sunt ne-reentrante = nu sunt thread-safe)

Exemple: trimiterea unui mesaj in 2 pasi (asamblare + transmisie). Daca se pierde procesorul asamblare <-> transmisie => suprascrierea bufferului mesaj. Alte erori: errno, malloc, apeluri stdio

Tratarea semnalelor : un thread trateaza un semnal in vreme ce alt thread vrea ca semnalul respectiv sa termine aplicatia (problema apare din cauza ca semnalele sunt definite per proces nu per thread)

Dezavantaje threaduri user (Existenta lor nu e cunoscuta kernelului)

1. Daca un thread user executa un apel de sistem care se blocheaza in kernel, planificatorul kernel considera ca intreg procesul s-a blocat si aloca CPU altui proces. (indiferent daca procesul curent mai are alte threaduri ready)

2. Daca un thread user comite un page fault similar ca la 1 => planificatorul kernel alege alt proces.

3. kernelul poate lua procesorul unui thread user care are un spinlock (un fel de wait) pe care nu l-a eliberat => scadere dramatica de performanta pt aplicatiile cu threaduri user paralele.

OBS: exista o lipsa de coordonare intre schedulerul kernel si implementarea pachetului de threaduri user (care se rezolva cu SCHEDULER ACTIVATIONS)

Scheduler Activations (metoda de coord. kernel-pachet threaduri user)

=>reprezinta aproximarea unui kernel thread

Are stive kernel si user
Ofera context de executi pt un thread user
Ofera upcall pt evenimente de mai multe tipuri
Fiecare upcall => noua activare

Tipuri de evenimente upcall: adauga procesor, procesor preemptat (adauga threadul user care se executa in activarea care a pierdut CPU in coada de threaduri ready), activare blocata, activare deblocata.

Cum functioneaza:

La pornirea procesului

Kernelul aloca o activare + notifica aplicatia (upcall adauga CPU)

Sistemul de gestiune al user threadurilor primeste notificarea si declanseaza threadul „main”

La cerere de suplimentare a concurentei

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 creaza o noua activare

Se notifica aplicatia

Schedulerul user copiaza starea threadului blocat din vechea activare si notifica kernelul ca aceasta poate fi refolosita

Cand un thread user care detine un spinlock pierde procesorul

Kernelul genereaza un upcall

Se verifica daca threadul e in critical section

Daca da, threadul e continuat printr un context switch in user space

La iesire din critical section, se da controlul inapoi upcall ului tot prin context switch in user space

Fork-Join Parallelism

```
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
```

Threading issues

Semantici gresite pentru fork() si exec()
Erori de tratare a semnalelor
Anularea unui thread target
Erori care pot proveni de la thread-local storage (TLS)
Erori din scheduler activations

Tratarea semnalelor

Prin semnale se aduce la cunostinta faptul ca un anumit eveniment s-a petrecut. Tratarea unui semnal:

1. Semnalul este generat de un anumit eveniment
2. Semnalul este trimis catre un proces
3. Semnalul este trata de unul dintre signale handlers
    1. Default
    2. user-defined

Fiecare thread are propria masca de semnale.
Toate threadurile aceluiasi proces partajeaza handlerele de semnal.
SIG_IGN/SIG_DFL => se aplica tuturor threadurilor unui proces
Semanlele de tip interupere => executate de orice thread care nu blocheaza semnalul respectiv.
Daca toate threadurile au mascat un anume semnal, se asteapta dupa primul thread care deblocheaza tratarea semnalului respectiv.
Semnalele de tip trap executate exclusiv de threadul care le-a generat.
Thread_kill = trimite un semnal catre un thread din acelasi proces
Semnalul e de tip trap => tratat doar de threadul pt care e adresat
Nu se pot trimite semnale unui anume thread din alt proces

Thread Cancellation

Terminarea unui thread inainte de a se finisa.

```
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);
```