Curs 09: Sincronizare

Sumar curs anterior

acţiunile în sistemul de calcul sunt de lucru cu I/O şi prelucrare: I/O intensive şi CPU intensive
în mod tradiţional, acţiunile sunt încapsulate în procese
pe un sistem multi-core putem executa mai multe acţiuni (procese) diferite sau un proces clonat şi cu memorie partajată între el şi procesele copil
dezavantajele proceselor sunt overhead de creare, planificare şi consum de memorie (heavyweight)
thread-urile sunt lightweight processes cu timp de creare şi rulare (activare) rapid, partajează spaţiul de adresă
thread-urile au probleme din cauza lipsei de izolare între ele: corupere de memorie, coruperea întregului proces
un thread abstractizează procesorul: stivă/stack pointer şi instruction pointer
un thread e definit de un TCB (thread control block)
activarea unui thread se face prin rularea unei funcţii specifice thread-ului
întreg spaţiul de adresă este partajat între thread-uri ale aceluiaşi proces; TLS (Thread Local Storage) şi stiva sunt specifice fiecărui thread, dar pot fi în continuare accesate de un alt thread
thread-urile sunt implementate în forma kernel-level threads sau user-level threads
kernel-level threads au suport complet la nivelul nucleului, pot folosi suportul multiprocesor
user-level threads nu au suport în kernel (sunt implementate în user space), dar au timp de activare mai rapid

Resurse comune

facilitează interacţiunea între procese şi thread-uri
un thread trebuie să aibă în vedere că nu este singurul care accesează; în caz contrar apar inconsevenţe sau date corupte
dacă toate thread-urile citesc, nu e nici o problemă; când cel puţin unul scrie apar probleme:
* care este informatia corectă: cea de dinainte de scriere sau de citire
când scriu două thread-uri, e posibil ca operaţiile de scriere să nu fie atomice, să se intercaleze rezultând în date inconsecvente:
* dacă un tread face a++, şi altul a++ e posibil ca rezultatul final să fie a_initial+1 în loc de a_initial+2
problemele sunt mai mari la structuri de date mai mari
+ demo cu folosirea listelor fără acces exclusiv
problemele sunt greu de depanat, nu sunt deterministe
probleme ce pot apărea
* acces concurent de scriere la date comune
* acces de citire când datele nu au fost încă actualizate, sau acces după ce datele au fost curăţate/refolosite: race conditions

* pierderea notificărilor şi aşteptare nedefinită
+ demo cu TOCTOU
nevoie de mecanisme de acces exclusiv la date, de serializare/atomizare a unor acţiuni şi de ordonare a acţiunilor (notificare şi aşteptare)
+ de ce close(1) şi open() sau close(1) şi dup(3) sunt problematice dar dup2(3, 1) nu
trebuie ca toate thread-urile să respecte aceeaşi politică: dacă folosim biblioteci care nu sunt conştiente de asta, vor fi probleme
+ demo cu reentranţă din cursul 08
dacă nu folosim biblioteci thread safe, nu folosim thread-uri; în Python există un GIL (Global Intepreter Lock) are serializează tot accesul

Primitive de sincronizare

reminder de la APD
pentru mecanisme de acces exclusiv şi de atomizare de acţiuni: variabile atomice, mutex (lock / unlock), spinlock (lock / unlock)
pentru ordonarea acţiunilor: variabile de condiţie (wait, notify), semafoare (up, down), monitoare (enter, leave, wait, notify), cozi de aşteptare (wait, wake_up)
asigură funcţionarea corectă a programului, dar:
* e dificil de avut în vedere toate cazurile şi folosit cum trebuie toate mecanismele
* folosirea necorespunzătoare duce la probleme şi mai greu de depanat
* codul serial încetineşte programul (legea lui Amdahl)
* funcţiile de sincronizare au overhead
ideal este de micşorat folosirea primitivelor de sincronizare prin partiţionarea datelor
regiunile critice sunt cele în care sunt date la comun sau operaţii care trebuie atomizate:
* regiuni de granularitate mică înseamnă că au cod serial puţin dar overhead-ul de lock()/unlock() este semnificativ
* regiuni de granularitate mare însemnă că overhead-ul de locking e mic, dar o parte mare e serializată (şi se aplică legea lui Amdahl)
+ demo cu granularitate mică sau mare
ne conentrăm în continuare pe interne, partea de utilizare şi bune practici o ştiţi de la APD, o veţi aprofunda la ASC şi APP (anul 4 C1)

= Internele primitivelor de sincronizare

pentru ca primitivele de sincronizare să funcţioneze, implementarea lock() şi unlock() trebuie să fie atomică; oul şi găina: cum implementăm lock() şi unlock() atomic fără să folosim primitive de sincronizare
întreruperile pot întrerupe fluxul de execuţie al unui program în orice moment; pe sistemele multicore, diferitele core-uri pot "intercala" comunicarea datelor pe magistrala partajată
instrucţiunile de procesor pot să nu fie atomice
instrucţiunea a += 5 se traduce în add [ebp-20], 5: atomică pe single core (read-update-write), dar neatomică pe multi-core (poate fi întreruptă de alt core, magistrala este partajată)
instructiunea a += 5 se traduce pe ARM în load r1, r2; add r1, r1, 5; store r2, r1; neatomică

atomizarea operaţiilor se realizează la nivel hardware:
* x86 single core: nu e nevoie
* x86 multi core: lock pe magistrală: prefixul lock în faţa unei operaţii
* ARM: operaţii tranzacţionale: ldrex, strex; dacă nu reuşeşte eşuează şi încerci iar
în GCC se foloseşte __sync_fetch_and_add() ca wrapper peste cazurile de mai sus
+ demo cu operaţii atomice (directorul sum-threads)
+ demo cu sum-threads-arm; toolchain-ul de ARM se ia de aici:
https://releases.linaro.org/components/toolchain/binaries/latest-7/aarch64-elf/gcc-linaro-7.4.1-2019.02-x86_64_aarch64-elf.tar.xz

Implementarea lock (spinlock)

implementare naivă:

```
lock = 0; /* init */

while (lock == 1)
; /* do nothing */
lock = 1; /* get lock */
```
---
poate apărea un TOCTOU; este nevoie de atomizarea comparaţiei şi iniţializării
procesoarele oferaţia de tipul compare-and-swap / compare-and-exchange: CAS(lock, 0, 1):

```
if value == to_compare
   value = to_update
return initial_value
```

```
while (CAS(lock, 0, 1) == 1)
  ;
```

+ demo cu operaţii atomice şi spinlock (directorul lock)
reparcurs demo cu sum-threads

Spinlock vs mutex

spinlock-ul este primitiva de bază pentru asigurarea atomicităţii
face busy waiting în aşteptarea eliberării, se bazează pe compare-and-swap
mutex-ul este o primitivă mai complexă, cu o coadă de thread-uri în aşteptare; dacă mutex-ul este luat, thread-ul intră în sleep
pentru protejarea structurilor interne, un mutex are un spinlock
spinlock-ul este util pentru regiuni critice de mici dimensiuni
mutex-ul este util pentru regiuni critice mai mari sau unele în care thread-ul de blochează

+ demo cu overhead de sincronizare spinlock vs mutex (directorul spinlock-mutex)
de avut în grijă la cache thrashing pe multicore pentru spinlock-uri: atunci când un spinlock este folosit pentru thread-uri de pe mai multe core-uri aceste pot "muta" variabila spinlock-ului de pe un core pe altul afectând performanţa cache-ului

Performanţa transferului. Producător-consumator

componente hardware sau software comunica şi îşi transferă informaţii: placă de reţea - procesor, server - client, dispozitiv de I/O - memorie
una (unele) produce (produc) informaţie, alta (altele) consumă
adesea una este mai rapidă decât cealaltă; există riscul ca cea mai înceată să dicteze comunicaţia
pentru aceasta se foloseşte buffering între cele două: cea care produce pune elemente în buffer, cealaltă le consumă; dacă e lent consumatorul, producătorul are spaţiu de depunere; dacă e lent producătorul, consumatorul poate consuma din ce s-a strâns până atunci
în mod uzual buffer-ul folosit este un buffer circular: un buffer cu două capete: unul de citire şi unul de scriere
cele două capete cresc independent: capătul de citire creşte atunci când acţionează consumatorul, capătul de scriere când acţionează producătorul
când capătul de scriere a ajuns la capătul de citire, buffer-ul este plin
când capătul de citire a ajuns la capătul de scriere, buffer-ul este gol
accesul la buffer trebuie să fie protejat
+ demo cu buffer circular

Sumar

datele comune permit comunicare rapidă dar pot genera probleme
problemele apar când avem cel puţin un thread care scrie
pot să apară şi probleme de race-condition când două acţiuni nu sunt atomizate şi se interpune un alt thread care şterge / modifică o informaţie critică
zona care trebuie să fie atomizată este regiunea critică
primitivele de sincronizare urmăresc atomizarea accesului la o dată sau a unor operaţii (mutex, spinlock) sau ordonarea operaţiilor (variabile condiţie, semafor)
granularitatea regiunii critice afectează performanţa sistemului: overhead de metode de sincronizare şi zonă serială în cod
operaţiile se atomizează la nivel hardware
diferă comportamentul pe single core şi multi core şi între arhitecturi
pe sistemele multi core o acțiune atomică single core este neatomică din cauza magistralei partajate
implementarea operaţiei de lock() (spinlock) este realizată cu ajutorul operaţii compare-and-swap (CAS) prezentă pe toate arhitecturile
pe sistemele multi core operaţia CAS este prevăzută de lock pe magistrală
pe ARM accesul exclusiv la magistrală se realizează cu ajutorul unor instrucţiuni tranzacționale (de tipul totul sau nimic)

spinlock-urile (mai simple) sunt folosite pentru regiuni critice mici, mutex-urile (mai complexe) pentru regiuni critice mari sau acolo unde thread-ul se va bloca
pentru comunicarea între componente (hardware sau software) se foloseşte modelul consumator productător
diferenţa de viteză este compensată uzual de un buffer circular