

Capitol 11: Dispozitive de intrare/ieșire

Sumar curs anterior (Sincronizare)

datele comune permit comunicare rapidă dar pot genera probleme
problemele apar când avem cel puțin un thread care scrie
pot să apară și probleme de race-condition când două acțiuni nu sunt atomizate și se
interpune un alt thread care șterge / modifică o informație critică
zona care trebuie să fie atomizată este regiunea critică
primitivele de sincronizare urmăresc atomizarea accesului la o dată sau a unor operații
(mutex, spinlock) sau ordonarea operațiilor (variabile condiție, semafor)
granularitatea regiunii critice afectează performanța sistemului: overhead de metode de
sincronizare și zonă serială în cod
operațiile se atomizează la nivel hardware
diferă comportamentul pe single core și multi-core și între arhitecturi
pe sistemele multi-core o acțiune atomică single core este neatomică din cauza magistralei
partajate
implementarea operației de lock() (spinlock) este realizată cu ajutorul operații
compare-and-swap (CAS) prezentă pe toate arhitecturile
pe sistemele multi-core operația CAS este prevăzută de lock pe magistrală
pe ARM accesul exclusiv la magistrală se realizează cu ajutorul unor instrucțiuni
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 folosește modelul
consumator producător
diferența de viteză este compensată uzual de un buffer circular

Dispozitive și interfețe de I/O

cele trei resurse ale sistemului de calcul sunt: procesorul, memoria și I/O
procesorul este folosit pentru prelucrarea datelor; memoria este folosită pentru a reține
datele și codul proceselor (și nucleului), I/O este folosit pentru comunicarea cu exteriorul
un proces care nu are mod de comunicare cu exteriorul este inutil (nu citește nimic, nu
generează nimic)

procese comunică prin interfețe de I/O cu:

- * utilizatorul: tastatură, mouse, monitor, imprimantă, cameră video
- * alte procese
 - * de pe același sistem: folosind fișiere, pipe-uri/fifo-uri, socket-uri UNIX
 - * sau pe alte sisteme: folosind rețeaua, dispozitive unde sunt
- * dispozitive dedicate:
 - * de informare/investigare: senzori (elemente care culeg informație din exterior): de temperatură, umiditate

* de control: actuatori (elemente mecanice sau electrice care execută o acțiune): de obicei în sistemele robotice sau industriale

* de stocare: HDD, USB drive, CD-ROM

comunicarea cu dispozitivele I/O este critică și este mediată de sistemul de operare; sistemul de operare are grijă ca datele transferate către/de la proces să nu fie interpellate de alt proces; comunicarea cu I/O necesită, de obicei, tranziția în modul privilegiat (kernel space)

pe lângă multiplexarea/demultiplexarea datelor, nucleul oferă mecanisme de "scheduling" a transmiterii datelor pentru îmbunătățirea performanței

pentru I/O vorbim în general, de niveluri: nivelul hardware, nivelul sistemului de operare / kernel, nivelul aplicație / interfața user space; le vom parcurge bottom up

I/O în hardware

recapitulare de la PM

la nivel hardware, comunicarea cu dispozitivele de I/O are loc pe axa: procesor -> magistrală -> controller -> dispozitiv

magistrala (de I/O) reprezintă un număr de linii electrice pe care se transmit

datele/comenzile/informațiile de la/către dispozitiv

datele ajung la un controller, un chip dedicat care are informații despre specificul fizic/electric al dispozitivului și îl comandă corespunzător

controller-ul dispune de zone de memorie (registre) care rețin datele sau comenzile sau stările; aceste registre sunt adresate de procesor printr-o schemă dedicată, similar memoriei registrele sunt de 4 tipuri:

* citire

* scriere

* comandă

* stare

procesorul folosește schema de adresare pentru a scrie sau a citi informații din registrul dedicat

sunt folosite două scheme de adresare a registrelor controller-ului:

* port-mapped I/O (isolated I/O): un spațiu dedicat de adresare este folosit

* memory-mapped I/O: un spațiu din adresarea *fizică* a procesorului este folosit pentru I/O; adică o parte din adresele fizice ajung în memoria principală, altele ajung în I/O

+ diagramă cu port-mapped I/O și memory-mapped I/O

memory-mapped I/O este folosit cel mai mult, simplifică circuitele interne pentru că folosește o singură schemă de adresare; de avut în vedere că datele ajung în memoria cache și este posibil să fie reordonate

memory-mapped I/O a crescut în prezență odată cu trecerea la arhitecturi pe 32 de biți și 64 de biți când spațiul fizic de adresare a devenit mai generos

pe x86 există și port-mapped I/O (instrucțiunile IN și OUT din x86 assembly) și

memory-mapped I/O

+ de văzut `/proc/ioports` și `/proc/iomem` (`sudo cat ...`); de făcut suma zonelor unde apare "system RAM" la `/proc/iomem`

Comunicarea hardware cu dispozitivele I/O

comunicarea cu dispozitivele de I/O trece prin controller: procesorul scrie în registrele controller-ului, apoi controller-ul le transferă dispozitivului; sau procesorul citește din registrele controller-ului datele primite de la dispozitiv

procesorul trebuie să știe când poate să citească sau să scrie; riscă să citească date nevalide sau să suprascrie datele care nu au fost trimise; procesorul trebuie să se sincronizeze cu controller-ul

sincronizarea se realizează în două moduri:

- * polling: procesorul investighează constant starea controller-ului și când are de citit/scriș date, operează

- * întreruperi: procesorul este notificat de controller printr-un semnal electric pe o linie dedicată de disponibilitatea de a primi sau transmite date

în general preferăm folosirea întreruperilor, pentru că nu țin procesorul ocupat

întreruperile sunt însă dezavantajoase la un trafic mare de date (de exemplu plăcile de rețea 10Gbit, 40Gbit) și atunci se preferă polling; DPDK, un framework de prelucrare rapidă de pachete, folosește polling

se poate tranzita din polling în întrerupere în funcție de situație (la trafic mic întreruperi, la trafic mare polling)

întreruperile sunt livrate de controller către controller-ul de întreruperi al procesorului (APIC pe x86, GIC pe ARM) care apoi livrează întreruperea procesorului; întreruperile sunt livrate pe o "linie" de întrerupere care identifică astfel întreruperea (IRQ); primirea unei întreruperi duce la rularea unei rutine de tratare a întreruperii (ISR: Interrupt Service Routine) aflată în memoria sistemului de operare în tabela vectorilor de întrerupere (IDT: Interrupt Descriptor Table)

+ `sudo cat /proc/interrupts`

pentru a minimiza interacțiunea cu procesorul, anumite dispozitive folosesc DMA (Direct Memory Access): un chip dedicat realizează transferul unor blocuri mari de date din controller în memoria principală la comanda procesorului fără ca procesorul să se ocupe de realizarea transferului

procesoarele I/O sunt componente specializate care au și rol de comandă (au instrucțiuni proprii) pentru realizarea transferului; mai mult decât DMA au secvențe de instrucțiuni pe care le execută; de exemplu PLX IOP 480 încorporează un core PowerPC

+ diagramă cu procesor I/O

Device drivere

rutinele de tratare ale întreruperilor și părțile ce realizează transferul de la / către controller sunt parte din sistemul de operare (kernel), mai precis din drivere

un driver este o componentă ce rulează uzual în kernel space care se ocupă de comunicarea sistemului de operare cu dispozitivul; este în mod tipic un modul de kernel

+ `/proc/devices` arată asocierea între anumite dispozitive (identificate prin major) și driverul corespunzător (identificat prin nume)

În general există o interfață generică de comunicare cu device driver-urile, iar device driver-urile traduc această interfață în comunicarea cu controller-ul dispozitivului la device driver ajung întreruperile de la hardware și cererile de comunicare declanșate prin apel de sistem din user space (de exemplu apeluri de forma `read()` sau `write()`) device driver-ul este astfel o colecție de funcții apelate la primirea întreruperilor sau a unor cereri de la aplicații

mai multe despre drivere la cursul de Sisteme de operare 2 (anul 4 C3, semestrul 2)

Niveluri intermediare pentru prelucrarea I/O

comenzile și datele transferate de la / către user space pot ajunge la device driver direct de la handler-ul de apel de sistem sau printr-un nivel intermediar la nivelul nucleului, nivel responsabil cu prelucrarea datelor din considerente de performanță sau pentru a reduce complexitatea driver-ului

acest nivel intermediar se numește, generic, I/O manager

este întâlnit des la discuri (spații de stocare) și la plăcile de rețea

la plăcile de rețea de la un apel de sistem până la driver, se trece prin stiva de networking; stiva se ocupă cu implementarea protocoalelor IP, TCP, în vreme ce driverul comunică cu placa de rețea care implementează protocolul Ethernet; vom mai detalia la cursul 11:

Networking în sistemul de operare

la discuri, există un nivel intermediar (disk layer, block layer) și un nivel de implementare a sistemului de fișier deasupra sa, independente de drivere

implementarea sistemului de fișiere este generică și se ocupă de maparea unui fișier (văzut ca secvență de octeți) pe blocuri pe disc; driverul se ocupă de transferul efectiv către controller

block I/O layer se ocupă de operații de ordonarea și unificarea cererilor (sorting and merging) pentru a face cât mai puține cereri cu dispozitivul de stocare; ordonarea este utilă la discuri mecanice (hard disks) dar nu la discuri flash (solid-state drive)

block I/O layer gestionează buffer cache-ul: cache de memorie ale datelor de pe disc pentru a scrie / citi mai repede; astfel operațiile de scriere din user space vor ajunge în buffer cache și ulterior vor fi sincronizate pe disc; la fel, citirea de date se face tipic din buffer cache, doar când datele nu sunt prezente se aduc de pe disc

+ demo cu disk cache

tipic la citiri de pe disc se face read-ahead pentru a aduce date mai multe pentru accese viitoare

Interfața user space

peste drivere și nivelurile intermediare nucleul expune interfața de apel de sistem aplicațiilor (system (call) API)

un proces abstractizează procesorul (thread-uri), memoria (spațiu virtual de adrese) și I/O (tabela de descriptori de fișiere)

operațiile expuse de kernel se realizează uzual peste un descriptor de fișier: fișier, director, char/block device, socket UNIX, socket de rețea, pipe, FIFO

interfața de file descriptori a fost extinsă în Linux la timere, semnale (timerfd, signalfd): universal I/O

în Windows, majoritatea componentelor sunt gestionate de un HANDLE, incluzând și evenimente, procese, thread-uri

operațiile uzuale pentru I/O expuse în system (call) API sunt de forma: open, close, read, write, seek (pentru block device-uri), ioctl

operația seek plasează un cursor/pointer; este specifică dispozitivelor cu acces aleator (block devices) nu la cele cu acces secvențial (char device-uri)

pentru a obține un descriptor de fișier, dispozitivele trebuie deschise (folosind open); se folosește o cale (o intrare în sistemul de fișiere)

aceea cale poate să nu fie un fișier: poate fi un socket UNIX, un FIFO, un block device sau un char device

block/char device-urile se găsesc uzual în /dev/; sunt identificate de un major și un minor (acesta este modul în care un device driver este înregistrat, calea este doar ceva pentru înțelesul utilizatorului; putem folosi orice nume cât timp avem majorul și minorul corespunzător:

<https://www.quora.com/What-is-a-major-and-minor-number/answer/Jyoti-Singh-277?ch=99&share=56a118ea&srid=oLql>)

+ vizualizare intrări în /dev

+ demo cu ioctl pe CD-ROM

Creșterea performanței lucrului cu I/O

dispozitivele de I/O sunt lente; operațiile de I/O sunt adesea blocante și împiedică performanțe ridicate

de aceea sunt folosite soluții pentru a eficientiza lucrul cu I/O

operațiile read / write sunt numite operații sincrone și blocante

* sincrone: primesc atunci rezultatul operației

* blocante: dacă nu sunt date disponibile (pentru citire) sau nu este loc să se scrie (pentru scriere), thread-ul curent se blochează, se trece în starea WAITING, se schimbă contextul de execuție

alternativ, se pot folosi operații non-blocante (sincrone): se primește cât era disponibil sau se scrie cât era disponibil și se întoarce; dacă nu era disponibil se întoarce cu o semnalizare și se încearcă mai târziu

o altă variantă este folosirea de operații asincrone (implicit non-blocante): se livrează o cerere către sistemul de operare și apoi operația se întoarce; ulterior se primește o notificare de la sistemul de operare sau se verifică încheierea operației

operațiile asincrone au avantajul eficienței: nu se blochează thread-ul, se execută altceva în paralel cu operația executată de kernel

au dezavantajul unui mod de programare complicat: este nevoie de un automat de stări care să fie actualizat pe măsură ce operația evoluează

modelul de programare asincronă este alternativă la modelul de thread-uri pentru prelucrarea de cereri multiple; de fapt, în biblioteca standard de Linux, API-ul POSIX de operații asincrone e implementat cu thread-uri; programarea cu thread-uri e mai simplă, dar

are un overhead de schimbare de context și probleme de sincronizare; programarea cu operații asincrone este mai complexă dar are un overhead mai redus

În aplicațiile cu multe operații I/O este nevoie de un API scalabil pentru multe conexiuni/cereri care să multiplexeze între file descriptori / handle-uri: epoll pe Linux, CompletionPorts pe Windows, kqueue pe *BSD

O altă abordare este reducerea numărului de apeluri de sistem efectuate pentru transfer, realizată în două moduri:

- * scatter / gather I/O: se colectează un vector de buffere și se face un singur apel pentru scrierea (scatter) sau citirea (gather) acestora

- * zero-copy: se evită transferul de date din kernel în user space și invers preferându-se transferul direct în kernel space de la un dispozitiv la altul fără intermedierea user space (dar sub comanda user space); se face acest lucru prin memory mapped files sau funcții precum sendfile() pe Linux/Unix sau TransmitFile() pe Windows

Sumar

dispozitivele de I/O permit comunicarea proceselor cu exteriorul

există mai multe niveluri de prelucrarea I/O, de la hardware către aplicații

la nivel hardware, comunicarea dintre procesor și dispozitiv este intermediată de un controller

controller-ul conține registre care sunt citite/scrise de procesor

adresarea registrelor controller-ului se face prin memory-mapped I/O sau port-mapped I/O pentru a ști când să scrie sau să citească date la sau de la controller, procesorul primește întreruperi sau folosește polling

Întreruperile vin ca un semnal electric pe o linie de întrerupere ducând la rularea unei rutine (ISR)

rutina este o componentă de cod de obicei parte dintr-un device driver, parte din kernel space care gestionează hardware-ul

driver-ul primește date de la hardware prin controller și din user space prin apel de sistem

nucleul sistemului de operare are niveluri software intermediare care implementează

componente generice sau eficientizează accesul: stiva de rețea, filesystem, block I/O layer

block I/O layer este responsabilă de buffer cache, caching în memorie a datelor de pe disc pentru operații cât mai rapide

la nivelul user space, se expune o interfață de tip descriptor de fișier și operațiile open, read, write, close, seek, ioctl

operațiile clasice sunt sincrone și blocante; pentru performanță ridicată se folosesc operații neblocante sau asincrone

tot pentru performanță ridicată se folosesc soluții de tipul scatter-gather și zero-copy