Curs 10: Dispozitive de intrare/ieşire

Sumar curs anterior

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 acţiune 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 tranzacţionale (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

Dispozitive şi interfeţe de I/O

cele trei resurse ale sistemului de calcul sunt: procesorul, memoria şi I/O
procesorul este folosit pentru prelucrarea datelor; memoria este folosită pentru a reţine datele şi codul proceselor (şi nucleului), I/O este folosit pentru comunicarea cu exteriorul
un proces care nu are mod de comunicare cu exteriorul este inutil (nu citeşte nimic, nu generează nimic)
procesele comunică prin interfeţe de I/O cu:
* utilizatorul: tastatură, mouse, monitor, imprimantă, cameră video
* alte procese
  * de pe acelaşi sistem: folosind fişiere, pipe-uri/fifo-uri, sockeţi UNIX
  * sau pe alte sisteme: folosind reţeaua), dispozitive unde sunt
* dispozitive dedicate:
  * de informare/investigare: senzori (elemente care culeg informaţie din exterior): de temperatură, umiditate

  * de control: actuatori (elemente mecanice sau electrice care execută o acţiune): de obicei în sistemele robotice sau industriale
  * de stocare: HDD, USB drive, CD-ROM
comunicarea cu dispozitivele I/O este critică şi este mediată de sistemul de operare; sistemul de operare are grijă ca datele transferate către/de la proces să nu fie interpelate de alt proces; comunicarea cu I/O necesită, de obicei, tranziţia în modul privilegiat (kernel space)
pe lângă multiplexarea/demultiplexarea datelor, nucleul oferă mecanisme de "scheduling" a transmiterii datelor pentru îmbunătăţirea performanţei
pentru I/O vorbim în general, de niveluri: nivelul hardware, nivelul sistemului de operare / kernel, nivelul aplicaţie / interfaţa 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/informaţiile de la/către dispozitiv
datele ajung la un controller, un chip dedicat care are informaţii despre specificul fizic/electric al dispozitivului şi îl comandă corespunzător
controller-ul dispune de zone de memorie (registre) care reţin 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 foloseşte schema de adresare pentru a scrie sau a citi informaţii din registrul dedicat
sunt folosite două scheme de adresare a registrelor controller-ului:
* port-mapped I/O (isolated I/O): un spaţiu dedicat de adresare este folosit
* memory-mapped I/O: un spaţiu 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 şi memory-mapped I/O
memory-mapped I/O este folosit cel mai mult, simplifică circuitele interne pentru că foloseşte o singură schemă de adresare; de avut în vedere că datele ajung în memoria cache şi este posibil să fie reordonate
memory-mapped I/O a crescut în prezenţă o dată cu trecerea la arhitecturi pe 32 de biţi şi 64 de biţi când spaţiul fizic de adresare a devenit mai generos
pe x86 există şi port-mapped I/O (instrucţiunile IN şi OUT din x86 assembly) şi memory-mapped I/O
+ de văzut /proc/ioports şi /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 citeşte din registrele controller-ului datele primite de la dispozitiv
procesorul trebuie să ştie 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 şi când are de citit/scris 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 ţin procesorul ocupat
întreruperile sunt însă dezavantajoase la un trafic mare de date (de exemplu plăcile de reţea 10Gbit, 40Gbit) şi atunci se preferă polling; DPDK, un framework de prelucrare rapidă de pachete, foloseşte polling
se poate tranzita din polling în întrerupere în funcţie de situaţie (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 înteruperii (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 interacţiunea 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 şi rol de comandă (au instrucţiuni proprii) pentru realizarea transferului; mai mult decât DMA au secvenţe de instrucţiuni 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 şi părţile 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) şi driverul corespunzător (identificat prin nume)

în general există o interfaţă generică de comunicare cu device driverele, iar device driverele traduc această interfaţă în comunicarea cu controller-ul dispozitivului
la device driver ajung întreruperile de la hardware şi cererile de comunicare declanşate prin apel de sistem din user space (de exemplu apeluri de forma read() sau write())
device driver-ul este astfel o colecţie de funcţii apelate la primirea întreruperilor sau a unor cereri de la aplicaţii
mai multe despre drivere la cursul de Sisteme de operare 2 (anul 4 C3, semestrul 2)

Niveluri intermediare pentru prelucrarea I/O

comenzile şi 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 performanţă sau pentru a reduce complexitatea driver-ului
acest nivel intermediar se numeşte, generic, I/O manager
este întâlnit des la discuri (spaţii de stocare) şi la plăcile de reţea
la plăcile de reţea 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 reţea 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) şi un nivel de implementare a sistemului de fişier deasupra sa, independente de drivere
implementarea sistemului de fişiere este generică şi se ocupă de maparea unui fişier (văzut ca secvenţă de octeţi) pe blocuri pe disc; driverul se ocupă de transferul efectiv către controller
block I/O layer se ocupă de operaţii de ordonarea şi unificarea cererilor (sorting and merging) pentru a face cât mai puţine 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 operaţiile de scriere din user space vor ajunge în buffer cache şi 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
din acest motiv spunem că operaţiile de tip read/write nu sunt, în general, blocante

Interfaţa user space

peste drivere şi nivelurile intermerdiare nucleul expune interfaţa de apel de sistem aplicaţiilor (system (call) API)
un proces abstractizează procesorul (thread-uri), memoria (spaţiu virtual de adrese) şi I/O (tabela de descriptori de fişiere)
operaţiile expuse de kernel se realizează uzual peste un descriptor de fişier: fişier, director, char/block device, socket UNIX, socket de reţea, 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 şi evenimente, procese, thread-uri
operaţiile uzuale pentru I/O expuse în system (call) API sunt de forma: open, close, read, write, seek (pentru block device-uri), ioctl
operaţia seek plasează un cursor/pointer; este specifică dispozitivelor cu acces aleator (block devices) nu la cele cu acces secvenţial (char device-uri)
pentru a obţine un descriptor de fişier, dispozitivele trebuie deschise (folosind open); se foloseşte o cale (o intrare în sistemul de fişiere)
acea cale poate să nu fie un fişier: 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 şi un minor (acesta este modul în care un device driver este înregistrat, calea este doar ceva pentru înţelesul utilizatorului; putem folosi orice nume cât timp avem majorul şi minorul corespunzător: https://www.quora.com/What-is-a-major-and-minor-number/answer/Jyoti-Singh-277?ch=99&share=56a118ea&srid=oLqI)
+ vizualizare intrări în /dev
+ demo cu ioctl pe CD-ROM

Creşterea performanţei lucrului cu I/O

dispozitivele de I/O sunt lente; operaţiile de I/O sunt adesea blocante şi împiedică performanţe ridicate
de aceea sunt folosite soluţii pentru a eficientiza lucrul cu I/O
operaţiile read / write sunt numite operaţii sincrone şi blocante
* sincrone: primesc atunci rezultatul operaţiei
* 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 execuţie
alternativ, se pot folosi operaţii non-blocante (sincrone): se primeşte cât era disponibl sau se scrie cât era disponibil şi se întoarce; dacă nu era disponibil se întoarce cu o semnalizare şi se încercă mai târziu
o altă variantă este folosirea de operaţii asincrone (implicit non-blocante): se livrează o cerere către sistemul de operare şi apoi operaţia se întoarce; ulterior se primeşte o notificare de la sistemul de operare sau se verifică încheierea operaţiei
operaţiile asincrone au avantajul eficienţei: nu se blochează thread-ul, se execută altceva în paralel cu operaţia executată de kernel
au dezavantajul unui mod de programare complicat: este nevoie de un automat de stări care să fie actualizat pe măsură ce operaţia 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 operaţii asincrone e implementat cu thread-uri; programarea cu thread-uri e mai simplă, dar

are un overhead de schimbare de context şi probleme de sincronizare; programarea cu operaţii asincrone este mai complexă dar are un overhead mai redus
în aplicaţiile cu multe operaţii 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 şi se face un singur apel pentru scrierea (scatter) sau citirea (gather) acestora
* zero-copy: se evită transferul de date din kernel în user space şi 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 funcţii 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 aplicaţii
la nivel hardware, comunicarea dintre procesor şi dispozitiv este intermediată de un controller
controller-ul conţine 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 şti când să scrie sau să citească date la sau de la controller, procesorul primeşte întreruperi sau foloseşte 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 primeşte date de la hardware prin controller şi din user space prin apel de sistem
nucleul sistemului de operare are niveluri software intermediare care implementează componente generice sau eficientizează accesul: stiva de reţea, filesystem, block I/O layer
block I/O layer este responsabilă de buffer cache, caching în memorie a datelor de pe disc pentru operaţii cât mai rapide
la nivelul user space, se expune o interfaţă de tip descriptor de fişier şi operaţiile open, read, write, close, seek, ioctl
operaţiile clasice sunt sincrone şi blocante; pentru peformantă ridicată se folosesc operaţii neblocante sau asincrone
tot pentru performanţă ridicată se folosesc soluţii de tipul scatter-gather şi zero-copy