Curs 11: Networking in SO

Sumar curs anterior

Pentru a controla dispozitivele de Intrare/Iesire procesorul comanda controller-ul dispozitivului via magistrala (PCI, PCIe, etc).
Comunicatia procesor - controller se poate face:

- Port-mapped - dispozitivele de I/O au un spatiu separat de adrese (numit porturi) si necesita instructiuni speciale de procesor pentru a citi / scrie (instructiunile IN/OUT la x86).
- Memory-mapped - dispozitivele de I/O sunt mapate in memoria fizica, iar comunicatia se face prin scrierea/citirea la adrese prestabilite; orice instructiune de procesor poate fi folosita.

Dispozitivele de I/O functioneaza la viteza diferita de procesor; dupa ce o operatie de I/O este initiata, procesorul trebuie sa astepte completarea acesteia (e.g. citire de pe disk, e.g. un pachet nou vine pe placa de retea). Pentru a implementa aceasta sincronizare intre procesor si dispozitiv, se poate folosi:

- Polling: procesorul sta in bucla si verifica daca exista date noi. Avantaj: latenta scazuta, dezavantaj: cicli irositi atunci cand dispozitivul este lent sau datele vin cu frecventa mica.
- Intreruperi: procesorul isi continua alte taskuri (daca are, daca nu doarme) si este intrerupt atunci cand datele sunt disponibile. Avantaj: salveaza cicli; Dezavantaj: cand vin foarte multe intreruperi (e.g. una per pachet, un pachet la fiecare 60ns pe NIC de 10Gbps) overhead-ul intreruperilor este foarte mare - se trece in polling.

Pentru transferul datelor de la dispozitiv - memorie (sau invers), se poate folosi procesorul (costisitor) sau se poate folosi hardware de DMA (care permite dispozitivelor sa scrie in memorie direct).

Interactiune hardware - OS

O placa de retea (Network Interface Card, sau NIC pe scurt) expune mai multe perechi de ring-buffers (o pereche tx/rx) prin care SO trimite si primeste pachete. Comunicarea SO cu NIC se face folosind memory-mapped I/O, intreruperi sau polling (la incarcare mare) si DMA.

La initializarea placii, SO alloca o zona de memorie pentru pachete, si una pentru ring-uri, si le transmite placii scriind adresele memoriei nou alocate in zona de memorie monitorizata de placa.

RX: placa primeste un pachet si il salveaza intr-un buffer mic intern. Vede daca exista sloturi libere in RX; daca nu arunca pachetul. Ia primul slot liber din ringul de RX, si in el afla adresa zonei de memorie unde poate scrie pachetul. Folosind DMA, copiaza pachetul in memorie; pe unele arhitecturi pachetul este copiat direct in cache-ul L3 al procesorului (se numeste DDIO). Cand pachetul a fost copiat, avanseaza pointerul din ringul RX pentru a

notifica SO ca exista un pachet disponibil. Daca intreruperile sunt activate, genereaza intrerupere.

TX: NIC-ul verifica sloturile din ringul de TX pe care le poate transmite. Daca exista un astfel de slot, se initiaza DMA din memorie catre bufferul intern; atunci cand pachetul este copiat, se incrementeaza pointerul pentru a anunta SO ca poate refolosi bufferul; se genereaza intrerupere daca ele sunt configurate.

Odata copiat in bufferul intern, pachetul este transmis atunci cand mediul devine liber; inainte de transmisie pachetul poate fi modificat, in functie de capabilitatile placii de retea:

- Se calculeaza checksum si se adauga in packet.
- Se poate sparge pachetul in mai multe pachete mai mici, cand TSO este folosit (vezi discutie despre imbunatatire performanta mai jos).
- Se calculeaza FCS, se adauga la trailerul pachetului, si acesta este gata de a fi pus pe fir.

Multiplexare

- O placa de retea moderna poate expuna zeci de perechi de cozi RX/TX catre SO; fiecare din acestea se comporta ca o placa de retea separata pentru a comunica cu SO, e.g. genereaza propriile intreruperi.
- La transmisie, placa va lua pachete din cozile de TX folosind round-robin.
- La receptie, in mod normal placa va aplica o functie de hash la pachet pentru a decide in ce coada RX trebuie pus pachetul. Exista insa posibilitatea de a demultiplexa bazat pe adresa destinatie, i.e. fiecare coada RX sa primeasca pachetele pentru o anumita adresa IP.

Cozile multiple sunt in general folosite pentru a balansa pachetele catre toate core-urile din sistem - intreruperile diferitelor cozi pot fi directionate catre core-uri diferite. Cozile multiple pot fi folosite si pentru a simula mai multe placi de retea, fiecare cu adresa proprie de IP (vezi utiltarul ethtool).

Procesare SO - bottom half

RX: Atunci cand SO primeste pachete noi (anuntat via intrerupere), se va rula un cod care analizeaza fiecare pachet si hotaraste procesarea urmatoare. Se analizeaza adresa IP destinatie din pachet: daca aceasta este adresa uneia din adresele interfetelor active ale SO, packetul va fi procesat de partea “host” a stivei. Altfel, pachetul va fi procesat si eventual trimis mai departe (daca exista o intrare in tabela de rutare care se potriveste si SO este configurat astfel) sau va fi aruncat (drop).

In partea host, SO trebuie sa gaseasca socketul care va procesa pachetul si sa il livreze acestui socket. Selectia socketului se face urmand pasii urmatori:

- Se verifica daca exista o intrare in tabela de conexiuni deschise (folosind pe post de cheie un hash al 5-tuple din pachet - ip sursa, port sursa, ip destinatie, port

destinatie, protocol). Daca da, se foloseste socketul destinatie din tabela de conexiuni deschise, se livreaza pachetul acestui socket si procesarea bottom half se incheie.
- In caz contrar, se verifica in tabela de porturi asignate (apelul bind) - daca se gaseste un socket, pachetul va fi livrat stivei pentru acest socket.
- Altfel, pachetul este aruncat si eventual un pachet de raspuns generat si trimis catre sursa (ICMP).

Procesare stiva UDP

Fiecare socket UDP (creat ca urmare a apelului socket(,,PF_UDP) are o coada de pachete primite care nu au fost inca livrate. Atunci cand aplicatia executa recvfrom, va primi primul pachet din aceasta lista; daca nu exista nici un pachet, procesul va fi blocat pana cand un pachet vine; atunci procesul va fi deblocat si apelul recvfrom va intoarce noul pachet.

Procesare stiva TCP

Exista doua tipuri de sockets TCP: de ascultat (“listening”) si per conexiune. Dupa secventa socket/bind/listen avem un socket de ascultare. Apelurile accept si connect intorc un socket conectat. send / recv se pot executa doar pe un socket de conexiune, atunci cand conexiunea este in starea “CONNECTED”.

Atunci cand un segment TCP este receptionat, daca el are flag-ul SYN, se va folosi portul destinatie pentru a gasi singurul socket de ascultare; daca acesta exista, se va trimite SYN/ACK si va fi creata structura de date cu starea “half-open” care va fi asociata socketului de ascultare.

Atunci cand este primit ACK-ul pentru SYN/ACK, el va fi procesat de socketul de ascultare, conexiunea half-open este acum conectata, si este adaugata la coada de conexiuni disponibile corespunzatoare socketului de ascultare. Se adauga o intrare deasemenea in lista de conexiuni deschise (5-tuple) pentru noua conexiune. Aceasta intrare va fi stearsa atunci cand conexiunea este inchisa.

Atunci cand se executa accept, se ia prima conexiune disponibilia asociata socketului de ascultare, se creeaza un socket pentru ea, si se intoarce descriptorul asociat aplicatiei; daca nu exista conexiune, apelul se blocheaza.

Cand se primeste un pachet pentru o conexiune stabilita, se indentifica socketul si se vor salva datele intr-o zona de memorie per socket numita: ”receive buffer”. Apelurile recv vor lua date in ordine din aceasta structura; numarul de octeti primiti depinde de cantitatea datelor disponibile in receive buffer.

Daca exista “gauri” in receive buffer (de exemplu atunci cand un pachet se pierde iar pachetele urmatoare sunt receptionate), datele nu sunt livrate catre aplicatie decat atunci cand “gaura” este acoperita de retransmisia pachetului.

Receptie/Transmisie date din aplicatie

La TCP, pentru o conexiune stabilita, apelul send copiaza datele din spatiul utilizator intr-un buffer din nucleu, per conexiune, care se numeste **send buffer**. Apelul intoarce numarul de octeti copiati cu succes - acestia vor fi transmisi catre receptor atat timp cat reteaua functioneaza; insa nu este garantat ca vor fi primiti, de exemplu daca reteaua sau procesul la receptor pica.

Spre deosebire de UDP, unde un apel sendto creeaza un pachet IP, si un apel recvfrom intoarce continutul unui pachet IP, la TCP un apel send doar copiaza datele in buffer iar recv copiaza datele din buffer in userspace. La TCP pachetizarea este facuta automat de stiva, care creeaza un balans intre transmisia rapida (e.g. dupa primul octet primit de la utilizator) si overhead (transmiterea unui packet per fiecare octet primit ar irosi 64B per 1B de payload). Algoritmul lui Nagle este folosit in acest scop.

Transmisie date

Atunci cand un segment este gata de transmisie (exista date suficiente), se verifica daca fereastra de congestie (cwnd) si fereastra de receptie (receive window) permit transmiterea segmentului. Daca da, pachetul va fi creat, headerele adaugate si pus in coada ip_output a interfetei de iesire. De aici va fi copiat in ring-ul TX al placii de retea atunci cand exista slot-uri disponibile.

Considerente de performanta

Pentru UDP, se executa un apel de sistem pentru fiecare pachet; costul de tranzitie in/din kernel space este destul de mare, astfel incat viteza maxima este de 1Mpps.

Pentru TCP, se poate amortiza costul apelului de sistem trimitand si primind mai mult de 1500 octeti / syscall (e.g. 10000 sau 100000).

Demo TCP versus UDP.

Totusi pachetizarea se face la dimensiunea pachetului suportat de interfata (1500 in mod uzual) - iar munca stivei este proportionala cu numarul de pachete. De aceea a aparut conceptul de segmentation offload, prin care stiva pachetizeaza la pachete mari (64KB) care sunt sparte fie de placa de retea (hardware offload), fie de partea de jos a stivei (generic segmentation offload) inainte de transmisie. Aceasta creste performanta semnificativ.

Demo cu TSO/GSO/LRO.

Discutia pana acum a avut in vedere o singura conexiune. Atunci cand un server trebuie sa trateze un numar mare de conexiuni simultan, exista trei variante de implementare:

1. 1 proces per conexiune
2. 1 thread pe conexiune
3. 1 proces/thread pentru multiple conexiuni, eventual cu mai multe threaduri pentru a folosi toate core-urile.

Exista demo la curs pentru toate variantele de mai sus. Varianta 1 are overhead mare de creare proces per conexiune (~1ms); numarul total de conexiuni este limitat de nr. maxim de procese din sistem. La varianta 2, overhead-ul de creare este mai mic (fara copierea tabelei de pagini, fara page faults, fara TLB flush la context switch), insa schimbarea de context este de asemenea costisitoare si creeaza latenta.

Varianta 3 foloseste epoll pentru a detecta cand se poate citi dintr-un socket fara blocare, insa este necesara mentinerea unei structuri de stare per client, ceea ce face programarea mai dificila.