Sisteme de operare

Laborator 13

Semestrul I 2024-2025

Laborator 13

- advanced I/O
- programare client-server TCP/IP
  - servere concurente TCP
  - servere concurente UDP
  - servere multiprotocol

Multiplexarea I/O

- Q: cum citim date din doi descriptori de fisiere diferiti cand nu stim cand vor fi disponibile datele?
  - obs: citirea blocanta cu *read* nu functioneaza
- solutii posibile
  - *polling* (setam operatii neblocante pe descriptorii de fisiere si citim in bucla pe rand)
    - dezavantaje: proasta utilizare a CPU (putem folosi *sleep* pentru a ameliora situatia, dar nu stim nici cat sa asteptam !)
  - *I/O asincron*
    - instruim kernelul sa ne trimita un semnal (SIGPOLL/SIGIO) cand datele sunt disponibile
    - dezavantaj: exista un singur asemenea semnal per proces => la livrarea semnalului nu stim pe care descriptor sa citim (trebuie sa facem descriptorii non-blocanti si sa incercam citirea in secventa)
  - *multiplexarea I/O*
    - definim o lista de descriptori de fisiere de care suntem interesati
    - apelam o functie care se deblocheaza doar cand exista descriptori disponibili pt operatii de I/O (exista date disponibile, de ex)
    - dupa deblocare, exista functii care ne ajuta sa identificam acesti descriptori
    - ex: *select*, *poll*

select

- permite sa notificam kernelul in privinta
  - descriptorilor de interes
  - conditiilor in care ne intereseaza: *read/write/exceptii*
  - timpului de asteptare a disponibiliatii de I/O (deloc, pentru totdeauna sau pt o perioada specificata de timp)
- kernelul raspunde cu urmatoarele informatii
  - nr total de descriptori disponibili pt operatii de I/O
  - care descriptori sunt disponibili pentru fiecare tip de conditie: *read/write/exceptie*

*int select(int nfds, fd_set *readfds, fd_set *writefds,*

*fd_set *exceptfds, struct timeval *timeout);*

Valori timeout

- NULL => se asteapta la infinit
  - *select* se deblocheaza cand un descriptor e gata de I/O sau la primirea unui semnal
  - la primirea unui semnal *select* intoarce -1 si errno = EINTR
- timeout.tv_sec == 0 && timeout.tv_usec == 0
  - *select* se intoarce imediat, fara blocare
  - se foloseste in polling
- timeout.tv_sec != 0 || timeout.tv_usec != 0
  - asteapta durata de timp specificata in secunde si microsecunde
  - *select* se deblocheaza
    - cand un descriptor a devenit disponibil
    - cand a expirat perioada de timp
    - la primirea unui semnal (ca mai sus)
  - daca timeout-ul a expirat si niciun descriptor n-a devenit disponibil, *select* intoarce valoarea 0

Seturi de descriptori

- *readfds*, *writefds*, *exceptfds* specifica descriptorii de interes pt operatiile de I/O corespunzatoare
- *fd_set* = tip de date bitmap, 1 bit per descriptor de interes
- operatii cu seturi de descriptori
  - alocarea unei variabile
  - asignarea de variabile
  - una dintre urmatoarele operatii pe bitmap-uri

```
FD_ZERO(fd_set *fds);          // pune bitii din fds pe 0
FD_SET(int fd, fd_set *fds);   // pune pe 1 bitul fd din fds
FD_CLR(int fd, fd_set *fds);   // pune pe 0 bitul fd din fds
FD_ISSET(int fd, fd_set *fds); // testeaza daca bitul fd din fds e setat
```

Seturi de descriptori (cont.)

- secventa tipica de operare

```
fd_set rfds;
int fd;
...
FD_ZERO(&rfds);
FD_SET(fd, &rfds);
if(select(nfds, &rfds, ....) < 0)
        exit(1);
if(FD_ISSET(fd, &rfds))
        read(fd, buf, sizeof(buf) – 1); // read neblocant !
```

Nr max de descriptori si valoare de retur

- cand nu stim valorile descriptorilor primul parametru a lui *select* se alege ca fiind nr maxim de file descriptori posibil
  - FD_SETSIZE
- daca stim valoarea maxima a descriptorilor pe care ii folosim, atunci o folosim ca valoare a primului argument de apel al lui *select*
  - FD_SETSIZE poate fi mare (1024) => optimizare a cautarilor kernelului pt descriptori disponibili
- *select* intoarce:
  - -1 in caz de eroare
  - 0 pt timeout expirat si niciun descriptor disponibil
  - > 0 : nr de descriptori disponibili pt operatii de I/O
    - cei pt care operatiile de R/W nu se vor bloca
    - respectiv, descriptorii pt care exista o conditie de exceptie in asteptare (eg, OOB data in conexiuni de retea)

Operatii scatter/gather

- permit citirea/scrierea din/in buffere necontigue intr-un singur apel
- scatter *read*/ gather *write*
- folosesc vectori de buffere

```
struct iovec {
        void *iov_base;    // adresa de inceput a bufferului
        size_t iov_len;    // dimensiunea bufferului
}
ssize_t readv(int fd, const struct iovec iov[], int iovcnt);
ssize_t writev(int fd, const struct iovec iov[], int iovcnt);
```

Operatii scatter/gather (cont.)

ssize_t readv(int fd, const struct iovec iov[], int iovcnt);

- imprastie datele in buffere in ordine
- intotdeauna umple un buffer inainte sa treaca la urmatorul
- intoarce nr total de octeti cititi sau 0 daca nu mai sunt date (EOF)

ssize_t writev(int fd, const struct iovec iov[], int iovcnt);

- aduna datele de output din buffere in ordine *iov[0], iov[1], … iov[iovcnt -1]*
- intoarce nr total de octeti scrisi (in mod normal, suma dimensiunii bufferelor)

Memory-mapped files

- incarca continutul unui fisier intr-un buffer de memorie a.i. accesul la un element din buffer sa rezulte intr-un acces la disc
  - citirea unui element din buffer genereaza o citire din fisier
  - scrierea unui element din buffer rezulta intr-o scriere automata in fisier
  - astfel se executa I/O fara apelurile sistem *read*/*write*
- apel sistem care instruieste kernelul sa mapeze o regiune a unui fisier intr-o regiune din memoria RAM

caddr_t mmap(caddr_t addr, size_t len, int prot, int flag,

int fd, off_t off);

  - *caddr_t* e in general *char* *
  - intoarce adresa zonei de memorie in care e mapat fisierul

Argumente mmap

- *addr* = adresa de inceput e zonei de memorie unde vrem sa incarcam continutul fisierului
  - in general, *addr* = 0 pt a lasa kernelul sa aleaga adresa
  - daca *addr* != 0, trebuie sa fie multiplu al paginii de memorie virtuala obtinuta cu *sysconf(_SC_PAGESIZE)*
- *fd* = descriptorul fisierului al carui continut se mapeaza in memorie
  - fisierul trebuie deschis in prealabil
- *len* = nr de octeti mapati
- *off* = offsetul de start in fisier de unde se mapeaza octeti in memorie
  - multiplu al paginii de memorie virtuala
- *prot* = PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
  - trebuie sa se potriveasca cu modul in care a fost deschis fisierul
  - ex: *mmap(…,PROT_WRITE, …, fd,…)* nu merge cu *open(fd, O_RDONLY)*

Argumente mmap (cont.)

- *flag*
  - MAP_FIXED: valoarea de retur trebuie sa fie egala cu *addr*
    - daca nu e specificat si *addr* != 0, kernelul considera *addr* o sugestie
    - pt portatibilitate maxima *addr* = 0
  - MAP_SHARED: afecteaza operatiile de scriere in zona de memorie
    - daca flagul e setat => scrierea in buffer rezulta automat intr-o scriere pe disc
    - altfel spus, alte procese care partajeaza fisierul vad modificarea
  - MAP_PRIVATE: scrierile in zona mapata rezulta in crearea unei copii a fisierului mapat; toate referintele ulterioare la elementele modificate sunt efectuate asupra copiei
  - orice apel *mmap* **trebuie** sa foloseasca fie MAP_SHARED fie MAP_PRIVATE
  - MAP_ANONYMOUS: zona de memorie nu are asociat un fisier
    - initializata cu 0
    - *fd* si *off* sunt ignorate
    - impreuna cu MAP_SHARED implementeaza o zona de memorie partajata intre procese inrudite

Observatii

- Q: pt off != 0, ce se intampla daca dimensiunea zonei de memorie mapate nu e multiplu al paginii sistemului?
  - ex: pp. dimensiunea fisierului e 96 octeti si pagina sistemului 4096

  => dupa *mmap* se obtine o regiune de memorie de 4096 de octeti, dintre care ultimii 4000 sunt setati pe 0

  Obs: daca se scrie dincolo de offsetul 96, modificarile nu se reflecta in fisier

- semnale
  - SIGSEGV – generat pt acces la o zona de memorie la care nu avem permisiune de acces
    - generat si daca scriem intr-o zona de memorie RO
  - SIGBUS – generat de accese fara sens intr-o portiune a zonei de memorie mapate
    - ex: fisier mapat de un proces in intregime
      - alt proces trunchiaza fisierul inainte ca primul proces sa acceseze fisierul mapat
      - accesul la zona de memorie care a fost trunchiata rezulta in SIGBUS

- o regiune de memorie mapata pentru un fisier se mosteneste de catre copil dupa *fork*, evident nu si daca se apeleaza ulterior *exec* !

munmap

- zona de memorie se demapeaza:
  - implicit, la sfarsitul procesului (prin *exit*)
  - explicit cu *munmap*
  - inchiderea descriptorului de fisier nu demapeaza regiunea de memorie

int munmap(caddr_t addr, size_t len);

- nu afecteaza fisierul mapat (continutul ei nu e trimis pe disc)
- cu MAP_SHARED modificarile sunt automat scrise pe disc, nu cu *unmap* !

Concluzii mmap

- metoda mai rapida la copierea fisierelor (vizibil in cazul fisierelor mari)
- limitari: nu putem folosi *mmap* pentru conexiuni de retea sau pentru device-uri speciale (eg, terminal)
- e nevoie de atentie daca dimensiunea fisierului mapat se schimba dupa mapare
- in general simplifica algoritmii ori de cate ori manipularea unui buffer de memorie e preferabila operatiilor de I/O explicite (*read*/*write*)