Funcții sistem

Utilizarea Sistemelor de Operare

Paul Irofti

Universitatea din București
Facultatea de Matematică și Informatică
Department de Informatică
Email: paul.irofti@fmi.unibuc.ro

API

Application Programming Interface (API)

- set de rutine, structuri de date și obiecte
- descrie comportamentul, intrările și ieșirile rutinelor
- unul sau mai multe protocoale de apel
- bazat pe cod sursă, nu cod compilat
- documentația unui API la un moment dat este însoțită de o versiune
- descris formal într-un *header* (ex. `xml.h`)
- de-a lungul timpului apar sau dispar elemente din API
- implementarea unui API este împachetată într-o bibliotecă
- exemplu: `libc.so.11.2`
- două implementări pentru aceiași versiune de API sunt interschimbabile fără nevoia altor operații (ex. recompilare, repornire)

ABI

Application Binary Interface (ABI)

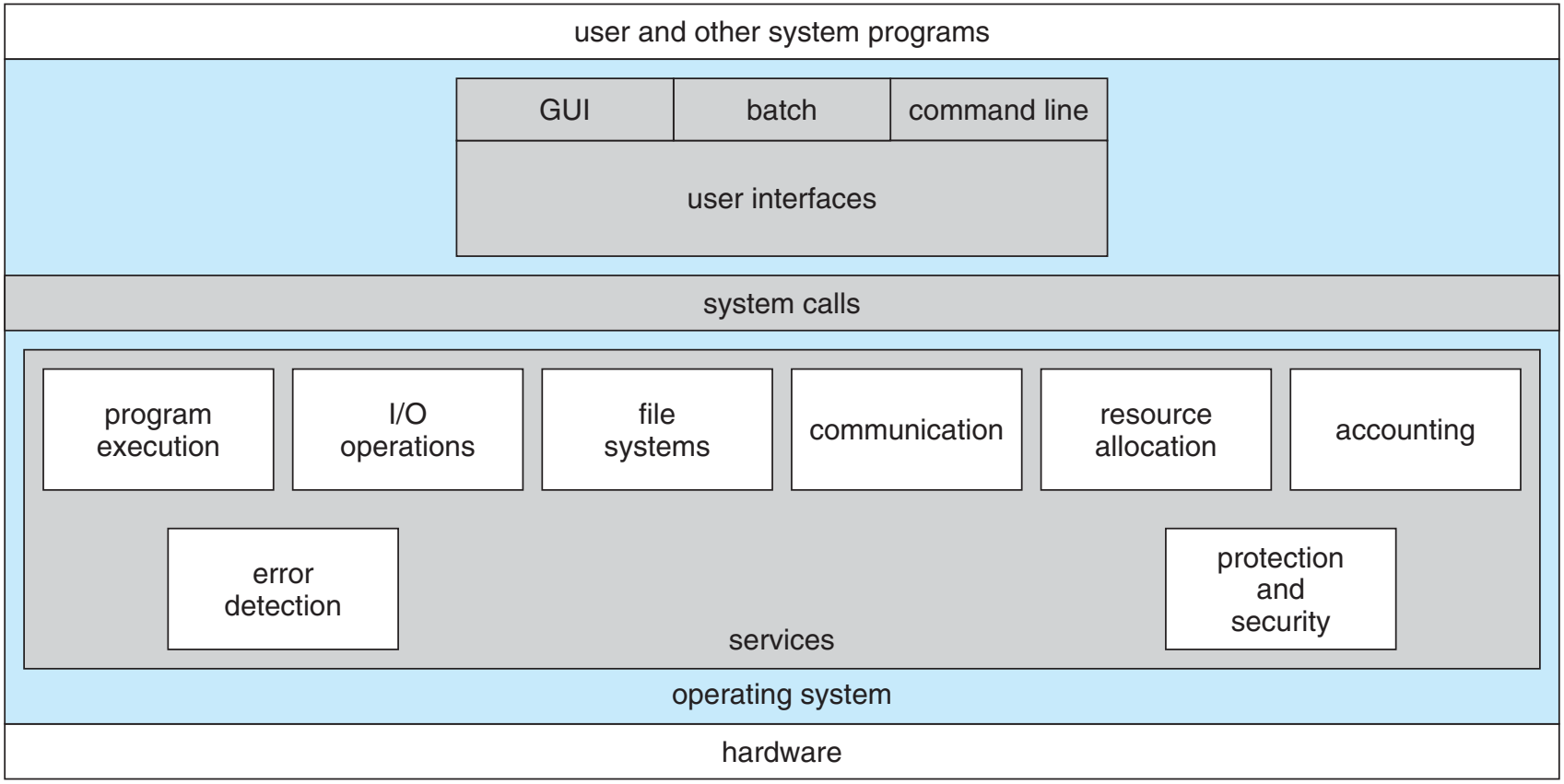
- asemănător cu principiile API
- bazat pe cod mașină (compilat), nu pe cod sursă!
- dependent de hardware
  - cum sunt împachetate și ordonate structurile de date
  - convenția de apel a funcțiilor (ex. unde sunt puse argumentele)
  - dimensiunea datelor (ex. `int` este `int32_t` sau `int64_t`)
  - aliniere la 16, 32 sau 64 de biți
- asigurat de compilatoare și sisteme de operare
- programe foarte vechi pot rula cu succes pe sisteme de operare moderne dacă ABI-ul a fost păstrat
- două programe binare pot comunica între ele fără acces la sursă dacă respectă același ABI
- cum rezolvă C++ problema ABI pentru funcții definite multiplu (*overloading*)?

Funcții sistem (`syscalls`)

API și ABI al sistemului de operare

- solicitarea unui serviciu din partea sistemului de operare
- arbitrează folosirea resurselor hardware și software între procese
- apelate cu ajutorul bibliotecilor de sistem (ex. biblioteca C)
- este interfața ce separă nivelul *userland* (al aplicaților) de sistemul de operare (nivelul *kernel*)
- API Win32 pentru Windows
- API POSIX pentru toate celelalte sisteme de operare: Linux, Android, Mac OS, OpenBSD etc.
- descrierea funcțiilor sistem se află în secțiunea 2 al manualelor Unix (ex. `man 2 open`)

Arhitectură și servicii



http://codex.cs.yale.edu/avi/os-book/

Exemplu: `read(2)`

```
ssize_t read(int d, void *buf, size_t nbytes);
```

`read()` folosește obiectul la care se referă descriptorul `d` din care încearcă să citească `nbytes` de date pentru a le scrie în bufferul indicat de `buf`

Întoarce numărul de bytes citiți efectiv sau -1 în caz de eroare. Codul de eroare este pus în variabila globală `errno`.

Observații:

- `d` – poate fi orice obiect, nu doar un fișier
- `buf` – conține date binare; semantic sistemul de operare nu știe ce reprezintă; programul care a solicitat serviciul știe cum să le interpreteze
- `nbytes` – este o rugăminte, nu o cerere fermă, nu trebuie respectată

Exemplu: `read(2)`

Apel când vreau să citesc o cantitate fixă:

```
char buf[BUFSZ];
left = sizeof(buf);
nread = 0;
while (left) {
        nr = read(fd, buf, left);
        if (nr == -1 || nr == 0)
                break;
        nread += nr;
        left -= nr;
}
```

Apel când vreau să citesc tot:

```
while ((nr = read(fd, buf, sizeof(buf))) != -1 &&
        nr != 0) {
        /* consume nr bytes from buf */
}
```

Exemplu: `read(2)`

Senzor de umiditate

- `d` – reprezintă legătura dintre aplicație și senzor
- `read(2)` cere date kernel-ului care la rândul lui, prin driver, cere date senzorului
- stivă apel:
  1. aplicație
  2. kernel syscall (*upper layer*)
  3. driver senzor (*lower layer*)

Exemplu: `read(2)`

Fișier text

- `d` – reprezintă legătura dintre aplicație și disc
- `read(2)` cere date kernel-ului care la rândul lui, prin mai multe nivele de abstractizare, cere date mediului de stocare
- stivă apel:
  1. aplicație
  2. kernel syscall (*upper layer*)
  3. sistem de fișiere (*middle layer*)
  4. driver stocare – disc, usb, flash (*lower layer*)

Tipuri de funcții sistem

- control procese
- organizarea fișierelor
- manipularea dispozitivelor hardware
- configurarea sistemului
- comunicații
- protecție și securitate

Control procese

- crearea și oprirea proceselor
- obținerea și setarea atributelor unui proces
- suspendarea procesului pentru un interval de timp
- așteptarea ca un eveniment să aibă loc
- semnalarea că un eveniment a avut loc
- alocarea și eliberarea memoriei
- *dump* memorie proces în caz de eroare
- depanarea unui proces activ
- mecanisme de sincronizare și acces exclusiv la resurse

Exemple: funcții proces

```
pid_t fork(void);
```

Pornește un proces nou identic din momentul în care a fost apelat:

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
else
        /* parent instructions */
```

`fork(2)` întoarce o valoare negtivă dacă a eșuat. Dacă procesul a fost creat, părintele primește înapoi `PID`-ul copilului iar copilul primește valoarea zero.

Pentru sincronizare și ieșire se folosesc `wait(2)` și `exit(3)`.

```
pid_t wait(int *status);
void exit(int status);
```

Exemple: funcții proces

```
int execve(const char *path, char *const argv[],
           char *const envp[]);
```

Pornește un proces nou și execută comanda din path cu argumentele argv.

```
pid_t pid = fork();
if (pid < 0)
        return errno;
else if (pid == 0)
        /* child instructions */
        char *argv[] = {"pwd", NULL};
        execve("/bin/pwd", argv, NULL);
        perror(NULL);
else
        /* parent instructions */
```

Organizarea fișierelor

- crearea și ștergere fișierelor
- deschidere și închiderea unui fișier
- citire, scriere, poziționare într-un fișier
- obținerea și setarea proprietăților

Exemple: organizarea fișierelor

```
int open(const char *path, int flags, int mode);
```

Deschide un fișier din `path` în modul citire, scriere conform `flags` și dacă trebuie să-l creeze folosește drepturile `rwx` conform `mode`. Întoarce un descriptor când este executat cu succes.

```
ssize_t write(int d, const void *buf, size_t nbytes);
```

Funcționează similar `read(2)`.

```
int close(int d);
```

Închide fișierul asociat descriptorului `d`.

Descriptori definiți implicit (ca cei din C)

- 0 – `stdin`
- 1 – `stdout`
- 2 – `stderr`

Manipularea dispozitivelor hardware

- cerere și eliberare acces la un dispozitiv
- citire, scriere, repoziționare
- obținerea și stabilirea diferitor proprietăți
- atașare, detașare dispozitive

Exemple: manipularea dispozitivelor hardware

Se folosesc tot `open(2)`, `read(2)`, `write(2)`, `close(2)`. În plus există `ioctl(2)` care are trei tipuri de apel

```
int ioctl(int d, unsigned long request);
int ioctl(int d, unsigned long request, int arg);
int ioctl(int d, unsigned long request, void *arg);
```

Dispozitivul asociat descriptorului `d` primește o cerere de tip `request` care are eventual și informație necesară în plus trimisă prin `arg`.

Exemple: manipularea dispozitivelor hardware

Operația `ioctl(2)` poate fi de tip intrare sau ieșire, caz în care al treilea argument este fie citit, fie scris. Tipul operației este dat implicit de numele cererii `request`.

Intrare:

```
char *font = "Comic Sans";
ret = ioctl(fd, SETFONT, font);
```

Ieșire:

```
char product_key[24];
ret = ioctl(fd, GETPRODKEY, product_key);
```

Configurarea sistemului

- informații privind ora și data
- setarea orei și a datei
- prelucrarea și configurarea proprietăților legate de procese, fișiere și dispozitive
- crearea, modificarea și eliminarea utilizatorilor și grupurilor
- configurarea și activarea serviciilor (ex. sshd, httpd)
- programarea executării unor comenzi la o anumită dată și oră
- stabilirea utilizării resurselor de către utilizatori (ex. disk quota)

Exemple: configurarea sistemului

Setarea orei și datei

```
int clock_gettime(clockid_t clock_id, struct timespec
    *tp);
int clock_settime(clockid_t clock_id, const struct
    timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *
    tp);
```

Obținerea PID-ului procesului curent și pe cel al procesului tată:

```
pid_t getpid(void);
pid_t getppid(void);
```

Semnalează procesul după ce se scurge un timp dat

```
unsigned int alarm(unsingned int seconds);
```

Comunicații

- crearea și eliminarea unei conexiuni
- trimiterea și primirea mesajelor (ex. client-server)
- crearea și obținerea accesului la zone de memorie partajate (*shared-memory*)
- atașararea și detașarea dispozitivelor la distanță (ex. samba, nfs)
- configurarea dispozitivelor de rețea
- configurarea rutelor de comunicație
- filtrarea pachetelor și mesajelor

Exemple: comunicații

Conectează două procese prin conectarea descriptorului scriere (ieșire) a primului la descriptorul citire (intrare) al celui de-al doilea:

```
int pipe(int fildes[2]);
```

Este în general apelat înainte de o operație de tip `fork(2)`.

Partajarea memoriei cu `mmap(2)`

```
void * mmap(void *addr, size_t len, int prot, int
    flags, int d, off_t offset);
```

Conținutul este luat prin intermediul descriptorului `d` începând de la poziția `offset`.

Exemple: comunicații

Crearea unui socket:

```
int socket(int domain, int type, int protocol)
```

Serverul deschide conexiunea:

```
int listen(int s, int backlog)
```

Clientul se conectează:

```
int connect(int s, const struct sockaddr *name,
    socklen_t namelen);
```

Trimiterea și recepționarea mesajelor dintr-un socket:

```
ssize_t send(int s, void *msg, size_t len, int flags);
ssize_t recv(int s, void *buf, size_t len, int flags);
```

Protecție și securitate

- controlul și accesul la resurse
- setarea și obținerea drepturilor asupra obiectelor (ex. fișiere, dispozitive)
- permiterea și blocarea accesului utilizatorilor și grupurilor
- încapsularea și limitarea unui serviciu
- elevarea și retrogradarea drepturilor unui serviciu sau utilizator

Exemple: protecție și securitate

Schimbarea drepturilor de acces:

```
int chmod(const char *path, int mode);
```

Fișierul sau directorul se găsește în `path` (ex. `/etc/passwd`), iar tipul de acces în `mode` (ex. 0775).

Schimbarea proprietarului

```
int chown(const char *path, uid_t owner, gid_t group);
```

`user` și `grup` sunt întregii corespunzători `ID`-urilor din fișierele `/etc/passwd` și `/etc/group`.

Exemple: protecție și securitate

- încapsulare:
  - `chroot(2)`, `chroot(8)` – schimbă rădăcina
  - FreeBSD jail și sysjail: crearea unui mediu minimal pentru execuție
  - containers: solaris, cgroups, docker, vmm
- W∧X: write or execute, not both! (NX bit)
- ASLR: address space layout randomization
  - kernel
  - userland
  - libraries
- limitarea funcților sistem ce pot fi apelate
  - pledge: promisiune de început ce tipuri de funcții sistem voi folosi în program
  - systrace: politică de acces la nivel de functie
- SELinux, AppArmor: extinderea drepturilor Unix pentru a crea politici complexe de acces la date și resurse