Rust sistemsko programiranje

Paralelne i distribuirane arhitekture i jezici
Zimski semestar, školska 2025./26.
Branislav Ristić

Sistemsko programiranje: definicija

- Sistemsko programiranje = pisanje softvera koji direktno koristi OS primitive:
  - fajlovi/FD-ovi, procesi, signali, memorijske mape, IPC, privilegije,
  - kod mora da poštuje ugovore sistemskih poziva i edge-case ponašanja.
- Ključna ideja:
  - Rust garantuje bezbednost u “safe” kodu,
  - ali OS API ugovori i dalje važe
    - `unsafe` je neizbežan na granici.

Sistemski pozivi

- Sistemski pozivi su API kernela:
  - open
  - read
  - write
  - mmap
  - …
- Kernel vraća:
  - uspeh (npr. broj bajtova)
  - grešku (tipično `-1` i `errno` u C svetu)

File Descriptor (FD)

- FD je mali ceo broj koji referencira deskriptor otvorene datoteke u kernelu.
  - Isti FD model važi za:
    - regularne fajlove, pipe, socket, terminal, itd.
- Resursi se moraju zatvarati (`close`) → RAII (`Drop`) je najbezbedniji obrazac.
- FD može “procureti” preko `exec` ako nema `CLOEXEC` / `FD_CLOEXEC`.
  - Program A otvori fajl i dobije FD=3.
  - Program A zatim pozove execve() da pokrene program B.
  - Ako FD=3 nema FD_CLOEXEC, program B i dalje ima FD=3 otvoren i može da čita/piše (ako ima prava).

`read` nije “sve ili ništa”

- Poziv: `read(fd, buf, count)` pokušava da pročita **do** `count` bajtova.
- Povratna vrednost `n` može biti:
  - `n > 0`: pročitano `n` bajtova (može biti manje od `count`)
  - `n == 0`: EOF (kraj fajla)
  - `n == -1`: greška (`errno`)
- Zašto `n` može biti manji od `count`:
  - EOF (ostalo je manje bajtova nego što se traži)
  - pipe/socket/tty: dostupno je “trenutno” manje
  - signal prekine syscall (`EINTR`), pa se često radi retry

`write` nije “sve ili ništa”

- Poziv: `write(fd, buf, count)` pokušava da upiše **do** `count` bajtova.
- Povratna vrednost `n` može biti:
  - `n > 0`: upisano `n` bajtova (može biti manje od `count`)
  - `n == -1`: greška (`errno`)
- Zašto može doći do partial write:
  - pipe/socket: ograničen kernel buffer
  - signal prekine blokirajući upis nakon delimičnog transfera

`errno`, `EINTR`, retry semantika

- `EINTR`: syscall prekinut signalom pre završetka
- Tipičan obrazac:
  - ako je `errno == EINTR` → retry (osim ako dizajn nalaže prekid)

`mmap`: kako radi uopšteno

- Ideja:
  - mapira fajl (ili anon memoriju) u virtualni adresni prostor procesa.
  - pristup = memorijski load/store, a ne `read/write`.
- Mehanizam:
  - kernel kreira virtualno mapiranje (VMA)
  - realne stranice se učitavaju “na zahtev” putem page fault-a (*demand paging*)
- Dve ključne varijante:
  - `MAP_SHARED`: izmene *mogu* biti vidljive drugim procesima i mogu se flush-ovati u fajl
  - `MAP_PRIVATE`: copy-on-write (COW); izmene su privatne i ne idu u fajl
- Tipične opasnosti:
  - `SIGBUS` ako se mapira region a fajl se skraćuje ili se pristupa nevažećem delu
  - offset mora biti page-aligned

Procesi: `fork/exec/wait`

- `fork()`:
  - duplira proces (child nastavlja od iste tačke kao parent)
- `execve()`:
  - zamenjuje trenutni proces novim programom (nema povratka ako uspe)
- `wait()/waitpid()`:
  - parent “reap”-uje child status (sprečava zombie)
- status treba tumačiti makroima (`WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, ...)
- Kritično pravilo:
  - posle `fork()` u multithread programu, child sme da poziva samo async-signal-safe funkcije dok ne uradi `execve()`.
  - async-signal-unsafe operacije: `printf/malloc/exit/…`

Signali i `sigaction`

- UNIX signali su asinhroni događaji koje kernel isporučuje procesu (ili niti) da bi ga obavestio o nečemu:
  - terminal (Ctrl-C),
  - drugi proces (kill),
  - hardware exception (invalid memory access).
- Signal handler mora biti ograničen:
  - ne sme da radi “svašta” (alokacije, lockovi, printf, itd.)
- Zašto `sigaction`:
  - preciznija kontrola od `signal()`, definisanije ponašanje.

Dizajn pravila za Rust sys-kod

- `unsafe` držati u **jednom modulu** (npr. `src/os.rs`).
- Izvesti bezbedne tipove:
  - `Fd` (close u `Drop`)
  - `Mmap` (munmap u `Drop`, `as_slice(&self) -> &[u8]`)
- Eksplicitno dokumentovati invariante (šta garantuje wrapper).

Zadatak 1: robustcat

- Implementirati cat-like alat preko libc:
  - open, read, write, close
  - korektno rukovanje partial I/O i EINTR
- `robustcat <path>`
- Test plan:
  - mali fajl (tekst)
  - veliki fajl (npr. 100MB)
  - pipe test
  - simulacija EINTR.
- Dodatne funkcionalnosti:
  - podrška za čitanje sa stdin kada path nije dat
  - --buf N veličina bafera

Zadatak 2: mmapgrep-lite

- Napraviti alat koji traži ASCII pattern u fajlu preko mmap.
- `mmapgrep <pattern> <path>`
- Test plan:
  - pattern postoji više puta
  - pattern ne postoji
  - prazan fajl
  - fajl sa binarnim sadržajem
- Dodatne funkcionalnosti:
  - --count (samo broj poklapanja)
  - --first (prvi match)

Zadatak 3: minish

- Minimalni runner komandi sa ispravnim status reporting-om.
- Koristeći `fork` i `exec`.
- `minish <cmd> [args...]`
- Test plan:
  - komanda koja uspe (/bin/echo hi)
  - komanda koja ne postoji
  - komanda koja se ubije signalom (npr. sh -c 'kill -SEGV $$')
- Dodatne funkcionalnosti:
  - --env KEY=VAL prosleđivanje env
  - --cwd DIR promena direktorijuma pre exec

Zadatak 4: graceful-follow

- tail -f stil praćenja fajla, izlaz na SIGINT (Ctrl-C) bez kršenja signal-safety pravila.
- `graceful-follow <path>`
- Test plan:
  - pokrenuti, dodati sadržaj u fajl (echo ... >> file), proveriti da se ispisuje
  - Ctrl-C → program se uredno gasi

Izvori

- Kerrisk, Michael, ed. Linux Man-Pages Project. man7.org, https://man7.org/linux/man-pages/

Rust sistemsko programiranje

Paralelne i distribuirane arhitekture i jezici
Zimski semestar, školska 2025./26.
Branislav Ristić