ALFS — Anushiravan-Level Fair Scheduler (User-space)

این مخزن یک **شبیه‌ساز User-space** از زمان‌بند ALFS است: نسخه‌ای تغییریافته از ایده‌ی Linux CFS که به‌جای RB-Tree از **Min-Heap** استفاده می‌کند و از طریق **Unix Domain Socket** رویدادها را به‌صورت **JSON** دریافت کرده و برای هر `vtime` یک خروجی زمان‌بندی **JSON** برمی‌گرداند.

> نکته: این پروژه «کرنل» نیست؛ ولی سعی کرده مفاهیم کلیدی CFS (vruntime + وزن + nice fairness) را با یک مدل deterministic پیاده کند.

Build

`make`

خروجی: `alfs/.`

Run

پارامترها:

- `--cpus N` تعداد CPU ها
- `--quanta-us Q` کوانتای هر tick (واحد: میکروثانیه)
- `--socket PATH` مسیر UDS (اختیاری)
  - اگر ندهید، برنامه اول `socket.event/.` و اگر نبود `event.socket/.` را امتحان می‌کند.
  - اگر هیچ‌کدام وجود نداشت/سرور بالا نبود، با پیام `connect: No such file or directory` و کد خروج `1` خارج می‌شود.
- `--burst-mode MODE` (اختیاری)
  - `freeze_pushback` (پیش‌فرض): هنگام `CPU_BURST` ابتدا `vruntime = max_vruntime` سپس **فریز** می‌شود.
  - `freeze`: فقط فریز vruntime (بدون pushback)
  - `track`: کلاً vruntime مثل حالت عادی update می‌شود (hint نادیده گرفتن)
- `--meta-minimal` فقط `{preemptions,migrations}` را در meta (**پیش‌فرض**) خروجی می‌دهد
- `--meta-extra` علاوه بر `{preemptions,migrations}` این‌ها را هم اضافه می‌کند:
  - `runnableTasks`
  - `blockedTasks`
- `--debug` لاگ‌های داخلی (stderr)

نمونه:

```
./alfs --cpus 4 --quanta-us 1000 --socket socket.event --debug
```

I/O Protocol (UDS + JSON framing)

- برنامه به عنوان **CLIENT** به UDS وصل می‌شود.
- ورودی: برای هر `vtime` یک JSON دریافت می‌شود (TimeFrame).
- خروجی: برای همان `vtime` یک JSON در قالب SchedulerTick تولید می‌شود و با newline خاتمه پیدا می‌کند.

Robust framing

چون delimiter دقیق تضمین نشده، ورودی به صورت stream خوانده می‌شود و با **brace/bracket counting** یک JSON کامل استخراج می‌کنیم:

- شروع از `{` یا `[`
- شمارش عمق `{}` و `[]` (با در نظر گرفتن string و escape)
- وقتی depth به ۰ رسید، یک JSON کامل داریم.

این روش هم با newline-delimited و هم بدون newline جواب می‌دهد.

Data Model

Task

هر تسک:

- `id` (string)
- `nice` در بازه‌ی `[-20..19]`
- `weight` از جدول `prio_to_weight` کرنل (شده hardcode داخل کد)
- `vruntime_us` («virtual microseconds» در واحد virtual runtime)
- `state`: `RUNNABLE | BLOCKED | EXITED`
- `cpuMask` (bitset)
- `cgroup` (pointer)
- `burst_remaining` و `burst_freeze` برای CPU_BURST

Cgroup

هر cgroup:

- `cpuShares` وزن گروه
- `cpuQuotaUs` و `cpuPeriodUs` برای throttling
- `cpuMask` (bitset)

- `vruntime_us` گروه
- `task_heap`: min-heap تسک‌های runnable داخل گروه (کلید: `task->vruntime_us`)
- خود cgroup هم داخل `cg_heap` قرار می‌گیرد (کلید: `cg->vruntime_us`)

Scheduling Algorithm (ELIAD: Explain Like I'm Defending It)

ایده‌ی CFS که نگه داشتیم

- CFS بگیرد CPU می‌خواهد هر تسک متناسب با وزنش.
  - معیار ساده‌ی پیشرفت: `vruntime`
  - هر بار که تسک روی CPU اجرا می‌شود:
    - `vruntime += delta_exec * NICE_0_WEIGHT / weight`
  - پس تسکی که کمتر CPU گرفته، `vruntime` کمتر دارد و زودتر انتخاب می‌شود.

در این پروژه:

- `delta_exec = quanta_us` (کوانتای هر tick)

چرا Heap؟

در CFS واقعی از RB-Tree استفاده می‌شود؛ ما اینجا به‌جای tree، heap داریم و `extract-min` را انجام می‌دهیم.

Group Scheduling (cgroup) چطور اضافه شد؟

دو سطح heap:

1. `cg_heap` (heap گروه‌ها): گروه با کمترین `cg->vruntime_us` را انتخاب می‌کند
2. داخل همان گروه، از `task_heap` کمترین `task->vruntime_us` انتخاب می‌شود

update ها:

- وقتی یک task از گروه روی CPU اجرا شد:
  - `cg->vruntime_us += quanta_us * NICE_0_WEIGHT / cg->cpuShares`
  - `task->vruntime_us += quanta_us * NICE_0_WEIGHT / task->weight` (مگر CPU_BURST)

چند CPU + Affinity

برای هر CPU به ترتیب `cpu=0..N-1`:

- از `cg_heap` pop می‌کنیم تا گروهی پیدا شود که:
  - throttled نباشد
  - runnable task داشته باشد
  - cpuMask آن گروه CPU را اجازه بدهد
- سپس از heap داخلی گروه pop می‌کنیم تا taskای پیدا شود که:

- را اجازه بدهد CPU خودش cpuMask
  - در همین tick روی CPU دیگر انتخاب نشده باشد (`current_cpu == -1`)
- برنمی‌گردد تا دوبار انتخاب نشود heap دوباره به tick انتخاب‌شده تا پایان task.
- بعد از اینکه همه CPUها انتخاب شدند، taskها دوباره به heap گروهشان push می‌شوند.

این کار deterministic است چون:

- انتخاب CPUها ترتیب ثابت دارد
- tie-break در heap با `(vruntime, seq, id)` است

Event Semantics

- `TASK_CREATE`: ایجاد + ورود به runnable heap گروه
- `TASK_BLOCK`: خروج از heap و رفتن به BLOCKED
- `TASK_UNBLOCK`: بازگشت به heap با همان vruntime
- `TASK_YIELD`: در heap همان گروه `vruntime = max_vruntime`
- `TASK_SETNICE`: فعلی دست نمی‌خورد vruntime وزن عوض می‌شود؛
- `TASK_SET_AFFINITY`: cpuMask عوض می‌شود
- `CGROUP_*`: ساخت/تغییر/حذف گروه (رد می‌شود non-empty حذف گروه)
- `TASK_MOVE_CGROUP`: اگر runnable باشد از heap قدیم حذف و به heap جدید اضافه می‌شود

CPU_BURST (تفسیر پیاده‌سازی)

به‌صورت پیش‌فرض:

- وقتی `CPU_BURST(taskId, duration)` آمد:
  - `task->vruntime` را به `max_vruntime` گروه می‌بریم (pushback)
  - سپس تا tick `duration` اگر روی CPU اجرا شد، **vruntime update نمی‌شود**
- بعد از تمام شدن duration، دوباره vruntime update می‌شود.

می‌توانید این را با `burst-mode freeze|track--` تغییر دهید.

Metadata دقیقاً چیست؟

در خروجی همیشه این دو مقدار وجود دارد:

- `preemptions`: تعداد CPUهایی که **tick** task دیگری/idle بوده و **این** task **قبل** رویشان یک **tick** قرار گرفته روی آن CPU.
  - نکته: تغییر `idle -> task` را preemption حساب نمی‌کنیم (همان الگوی مثال پروژه).
- `migrations`: تعداد taskهایی که CPUی قبلی‌شان این CPU قبل تغییر کرده **و** tick شان نسبت به `idle` شده است.

- این تعریف طوری انتخاب شده که swap دو task بین دو CPU را migration حساب نکند (مثل مثال پروژه).

پیش‌فرض، خروجی **minimal** است. اگر `meta-extra--` بدهید، این دو لیست هم اضافه می‌شود:

- `runnableTasks`
- `blockedTasks`

Debugging

`./alfs --cpus 4 --quanta-us 1000 --debug`

چیزهایی که چاپ می‌شود:

- ساخت task/cgroup
- انتخاب task برای هر CPU در هر tick

Tests (mini harness)

یک سرور ساده در `tests/server.py` وجود دارد که:

- یک UDS server ایجاد می‌کند
- فایل trace (مثلاً `tests/trace_demo.jsonl`) را line-by-line می‌خواند
- هر خط را به عنوان یک TimeFrame می‌فرستد
- پاسخ ALFS را می‌خواند و چاپ می‌کند

اجرای دقیق (دو ترمیناله)

ترمینال ۱ (سرور):

`rm -f socket.event event.socket` python3 tests/server.py socket.event tests/trace_demo.jsonl`

ترمینال ۲ (کلاینت ALFS):

`./alfs --cpus 1 --quanta-us 1000 --meta-minimal --socket socket.event`

برای تست‌های چند-CPU و meta کامل:

`./alfs --cpus 2 --quanta-us 1000 --meta-extra --socket socket.event`

Files

- `src/alfs.c`: همه چیز (heap + scheduler + JSON + UDS)
- `third_party/jsmn.h`: JSON parser کوچک
- `tests/server.py`: سرور تست UDS
- `tests/trace_demo.jsonl`: یک سناریو ساده
- `report/REPORT.md`: گزارش امتیازی (RB-Tree vs Heap + EEVDF + big.LITTLE)

"راهنمای دفاع" (Bulletهای طلایی)

۱) **مدل ذهنی**: «CFS تلاش می‌کند CPU را متناسب با وزن تقسیم کند؛ vruntime نشان می‌دهد چه کسی کمتر سهم گرفته.»

۲) **معادله‌ی vruntime**:

`vruntime += delta_exec * NICE_0_WEIGHT / weight`

)در این شبیه‌ساز: (`delta_exec = quanta_us`

۳) **چرا heap کار می‌کند**: چون انتخاب بعدی تقریباً همیشه "کمترین vruntime" است ⇒ `extract-min`.

۴) **مشکل heap نسبت به RB-tree** (برای گزارش): حذف دلخواه / traversal مرتب / عملیات load balancing / group scheduling و "یافتن نزدیک‌ترین‌ها" راحت‌تر است.

۵) **چند CPU + affinity**: چون heap فقط min می‌دهد، ما برای هر CPU pop می‌کنیم تا به گزینه‌ی eligible برسیم، موارد نامناسب را موقت کنار می‌گذاریم و برمی‌گردانیم.

۶) **cgroup**: دو سطح heap (group سپس task)، و vruntime گروه با shares scale می‌شود.

۷) **quota/period**: runtime گروه در هر period جمع می‌شود، وقتی از quota رد شد throttled و تا پایان period اجازه‌ی اجرا ندارد.

۸) **CPU_BURST**: policy با قابل تغییر + شفاف flag.

۹) **determinism**: tie-break مشخص (vruntime, seq, id) و انتخاب CPUها به ترتیب ثابت.