Программирование в Linux

Планирование процессов

Многозадачность

Кооперативная

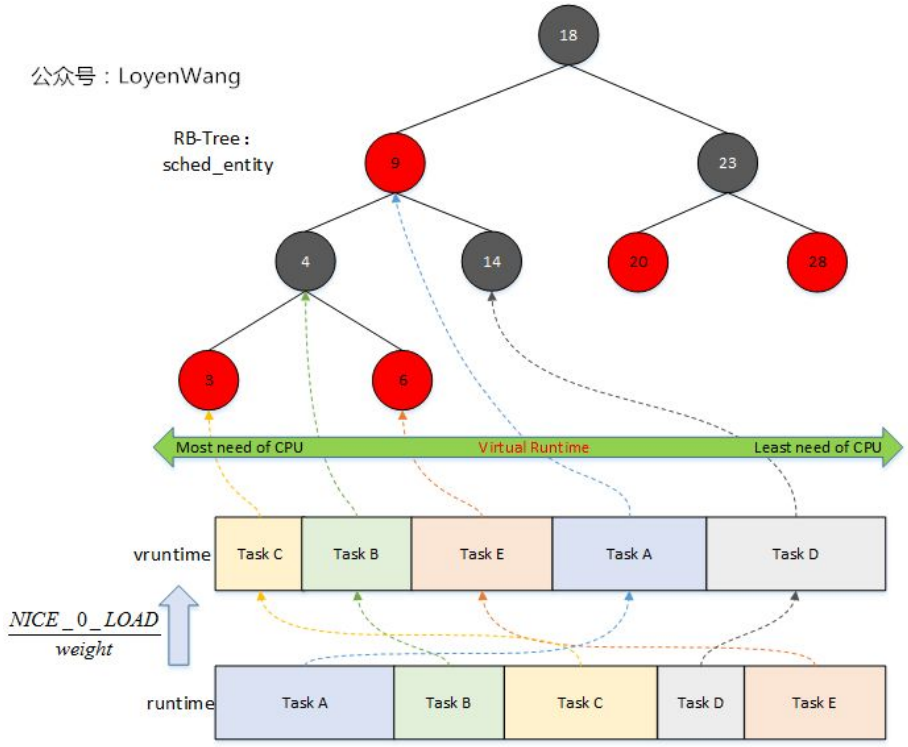
- Планировщик не нужен
- Процессы/треды сами передают управление друг другу
- Детерминированное, предсказуемое переключение задач
- Сбой в одном процессе может привести к отказу всей системы

Вытесняющая

- Процессы переключаются внешней силой (планировщик)
- Не всегда предсказуемое переключение (e.g. инверсия приоритетов)
- Ожидание в одном процессе не мешает исполнению других процессов
- БОльшая надежность

Современные ОС предпочитают вытесняющую многозадачность, но могут иметь средства для кооперативной.

Completely Fair Scheduler



$$curr->vruntime += delta\_exec \times \frac{NICE\_0\_LOAD}{curr->se->load.weight}$$

vruntime — оценочная величина сколько времени процесс “провел” на CPU
niceness (weight) — приоритет процесса

На исполнение отправляется процесс с наименьшим vruntime

Регулировка приоритетов

```
#include <unistd.h>

int nice(int inc);
```

Увеличивает niceness (уменьшает приоритет) вызывающего треда.
Возвращает новое значение

Диапазон niceness: [-20, 19]

Процесс без CAP_SYS_NICE не может уменьшить свою niceness.

При ошибке возвращает -1 (перед вызовом сбрасывайте errno 0 и проверяйте)

есть одноименная утилита командной строки

Массовая регулировка приоритетов

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

which:
PRIO_PROCESS —> who = PID
PRIO_PGRP —> who = PGRP (как -PID в kill)
PRIO_USER —> who = UID

Операция применяется ко всем множеству, подходящих под переданные параметры who и which, процессов

get* выдает наименьшее значение niceness в этом множестве

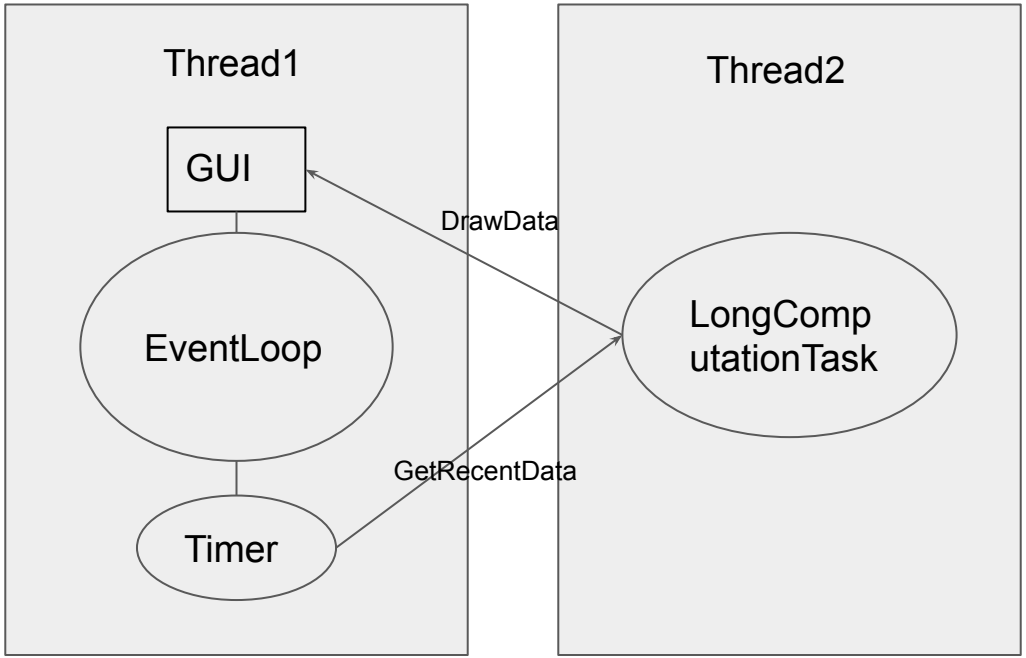
Значение niceness автоматически влияет еще и на планировщик IO. Можно переопределить приоритеты и для него

```
int ioprio_get(int which, int who);
int ioprio_set(int which, int who, int ioprio);

Note: There are no glibc wrappers for these system calls; see NOTES.
```

Зачем настраивать приоритеты?

Пример: планировщик Windows

- На исполнение каруселью (RR) ставятся потоки с наибольшим приоритетом
- Чтобы другие потоки не голодали, им иногда (раз 1-10 сек) повышается приоритет
- У потоков, обрабатывающих события пользовательского IO (мышь, клавиатура), приоритет повышается

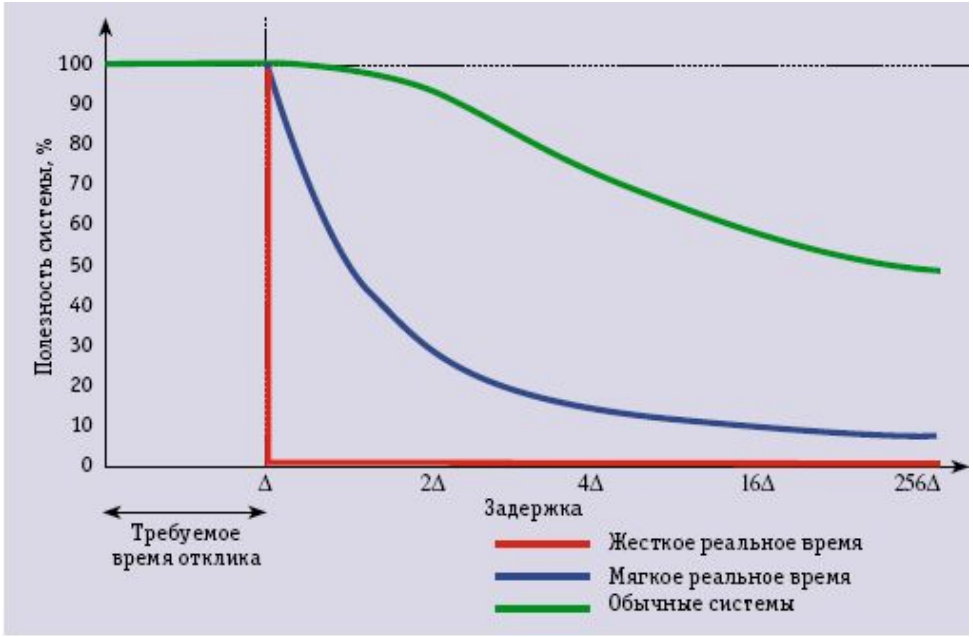


Системы реального времени (RT)

Система реального времени ⇔ система соблюдает ограничения времени реакции на воздействие

Жесткие системы RT — требует точного соблюдения ограничений
Мягкие — превышение не считается критическим сбоем

Задержка — время от момента воздействия до начала реакции
Иногда задержку померить сложно, но можно измерить колебания между реакциями



Поддержка RT в Linux. Политики планирования

У процессов есть статический приоритет (sched_priority) (0-99, по умолчанию 0).

ОС ставит на выполнения процессы с наибольшим приоритетом. Процесс выполняется, пока сам не остановится. Кванты времени не используются*.

подробнее: man sched

Политики планирования для RT систем

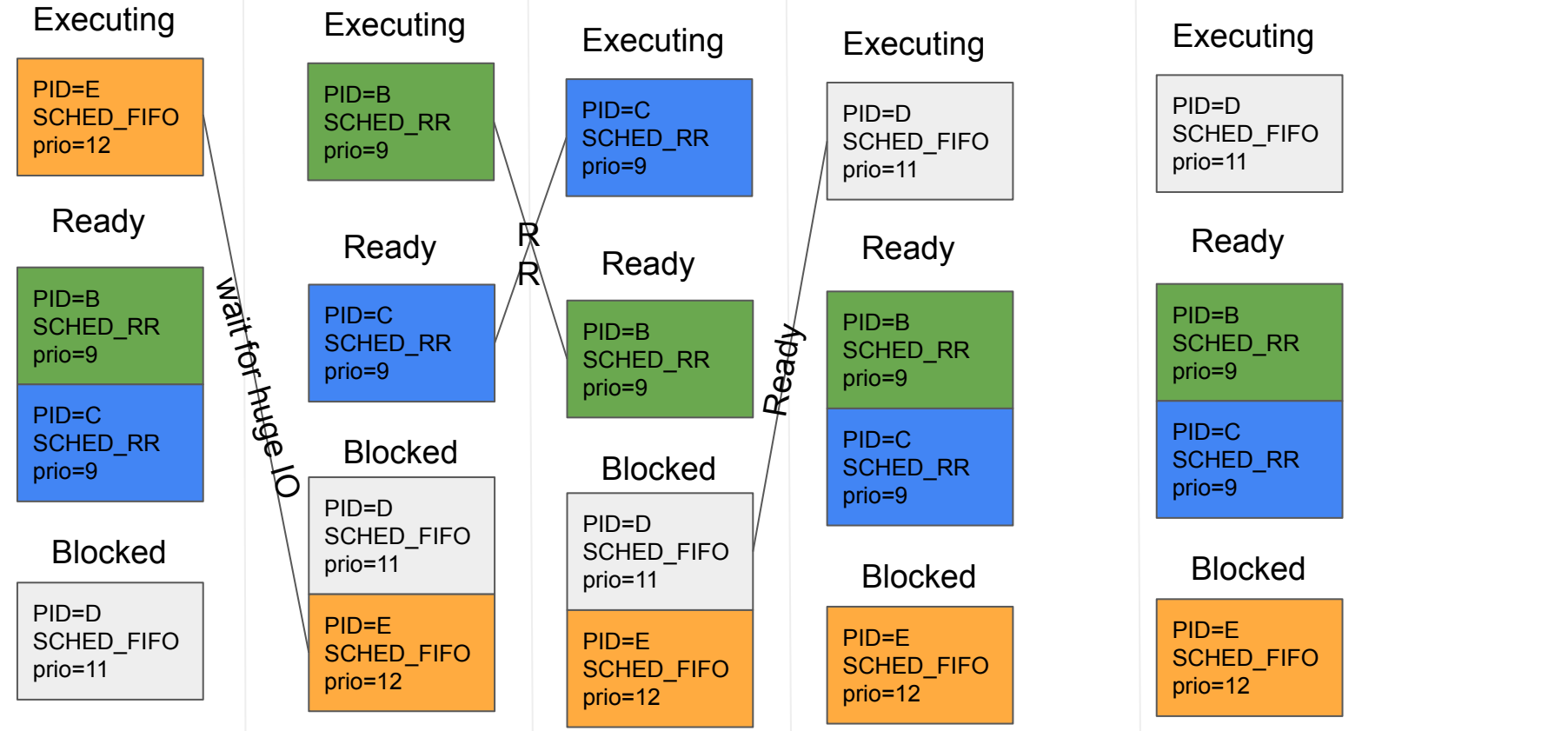
SCHED_FIFO — процессы выполняются по очереди с приоритетом
При блокировке (wait) или передаче управления (sched_yield), отправляется в конец очереди.

SCHED_RR — то же самое, что и SCHED_FIFO, но процессы с одинаковым приоритетом крутятся каруселью (RR) с квантами времени

SCHED_DEADLINE — можно указать дедлайн на выполнение процесса и ожидаемое время выполнения

SCHED_OTHER — дефолтная политика (не RT). Использует niceness

Политики устанавливаются для отдельных процессов. Процессы группируются по установленным политикам.


Executing
PID=E
SCHED_FIFO
prio=12
Ready
PID=B
SCHED_RR
prio=9
PID=C
SCHED_RR
prio=9
Blocked
PID=D
SCHED_FIFO
prio=11
wait for huge IO
Executing
PID=B
SCHED_RR
prio=9
Ready
PID=C
SCHED_RR
prio=9
Blocked
PID=D
SCHED_FIFO
prio=11
PID=E
SCHED_FIFO
prio=12
RR
Executing
PID=C
SCHED_RR
prio=9
Ready
PID=B
SCHED_RR
prio=9
Blocked
PID=D
SCHED_FIFO
prio=11
PID=E
SCHED_FIFO
prio=12
Ready
Executing
PID=D
SCHED_FIFO
prio=11
Ready
PID=B
SCHED_RR
prio=9
PID=C
SCHED_RR
prio=9
Blocked
PID=E
SCHED_FIFO
prio=12
Executing
PID=D
SCHED_FIFO
prio=11
Ready
PID=B
SCHED_RR
prio=9
PID=C
SCHED_RR
prio=9
Blocked
PID=E
SCHED_FIFO
prio=12

Узнать/задать значение кванта времени

```
dmis@dmis-MS-7A15:~$ cat /proc/sys/kernel/sched_rr_timeslice_ms
100
```

```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *tp);
```

На каких-то версиях ядра работает не только для процессов с SCHED_RR

```
dmis@dmis-MS-7A15:~$ cat test_sched.c
#include <stdio.h>
#include <sched.h>

int main() {
   struct timespec tp;
   sched_rr_get_interval(0, &tp);
   printf("time quantum %.2lfms\n", tp.tv_sec * 1000.0f + tp.tv_nsec / 1000000.0f);
   return 0;
}
dmis@dmis-MS-7A15:~$ ./test_sched
time quantum 16.00ms
```

```
dmis@dmis-MS-7A15:~$ sudo chrt -o 0 ./test_sched
time quantum 16.00ms
dmis@dmis-MS-7A15:~$ sudo chrt -b 0 ./test_sched
time quantum 16.00ms
dmis@dmis-MS-7A15:~$ sudo chrt -i 0 ./test_sched
time quantum 16.00ms
dmis@dmis-MS-7A15:~$ sudo chrt -r 10 ./test_sched
time quantum 100.00ms
dmis@dmis-MS-7A15:~$ sudo chrt -f 10 ./test_sched
time quantum 0.00ms
```

```cpp
int N = 100;
uint64_t average_delta = 0;
const int64_t required_usec = 3'500'000;
for (int i = 0; i < N; ++i) {
    std::cout << "Hello!\n";
    auto start = std::chrono::steady_clock::now();
    std::this_thread::sleep_for(std::chrono::nanoseconds(required_usec));
    auto cur = std::chrono::steady_clock::now();
    auto duration =  std::chrono::duration_cast<std::chrono::nanoseconds>(cur - start).count();
    std::cout << duration  << std::endl;
    average_delta += std::abs(duration - required_usec);
}
std::cerr << "average delta= "<< average_delta / N << std::endl;
```

```
dmis@dmis-MS-7A15:~$ ./test_delay > /dev/null
average delta= 114068
dmis@dmis-MS-7A15:~$ ./test_delay > /dev/null
average delta= 121865
dmis@dmis-MS-7A15:~$ ./test_delay > /dev/null
average delta= 114814
dmis@dmis-MS-7A15:~$ ./test_delay > /dev/null
average delta= 108775
dmis@dmis-MS-7A15:~$ sudo chrt -f 10 ./test_delay > /dev/null
average delta= 52920
dmis@dmis-MS-7A15:~$ sudo chrt -f 10 ./test_delay > /dev/null
average delta= 58299
dmis@dmis-MS-7A15:~$ sudo chrt -f 10 ./test_delay > /dev/null
average delta= 57989
dmis@dmis-MS-7A15:~$ sudo chrt -f 10 ./test_delay > /dev/null
average delta= 55253
dmis@dmis-MS-7A15:~$ sudo chrt -f 10 ./test_delay > /dev/null
average delta= 55193
```

Проблемы с RT процессами

- RT процесс забирает себе 100% CPU
- Активное ожидание результата от менее приоритетного процесса может никогда не закончиться
- Пассивное ожидание — может быть инверсия приоритетов
- Недетерминированность: кэши, swap, подгрузка страниц памяти с диска

Привязка процессов/тредов к определенному ядру

Планировщик по умолчанию пытается запускать один и тот же процесс на одном и том же ядре

Из-за несбалансированности нагрузки, планировщик может перепланировать процесс на другое ядро

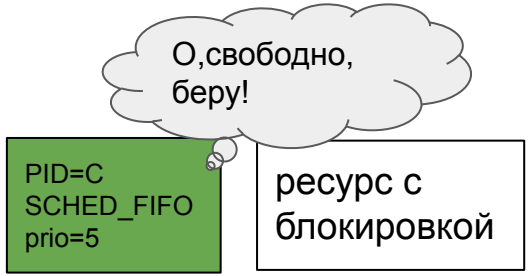
```
#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
void CPU_ZERO (cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize,
                       const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize,
                       cpu_set_t *set);
```

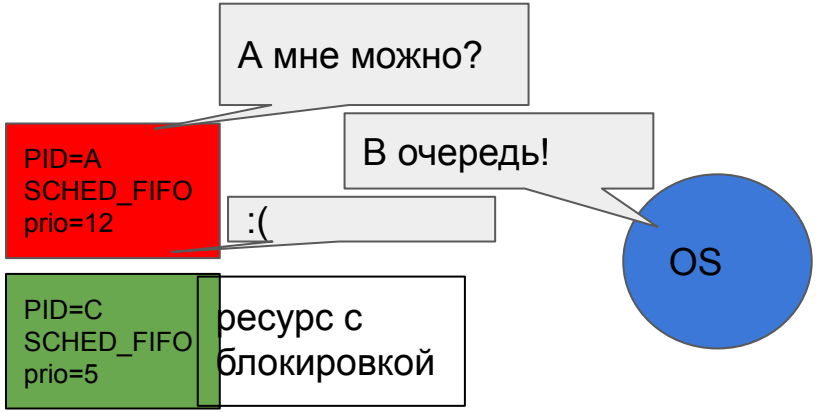
CPU_SETSIZE — максимальное поддерживаемое число ядер.
Корректный вызов:

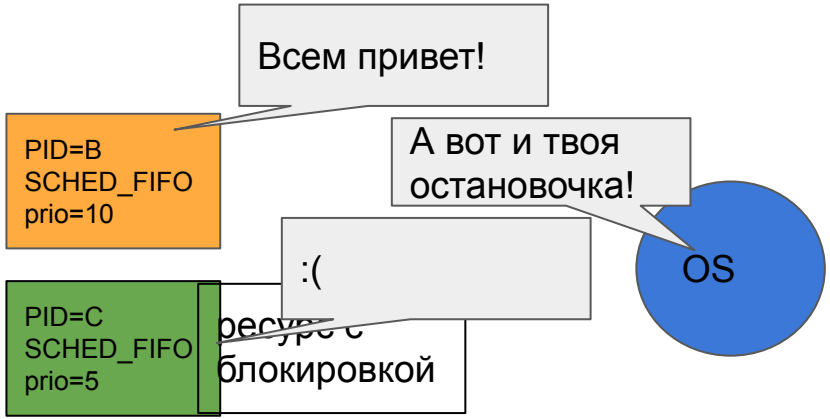
```
if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
    errExit("sched_setaffinity");
```

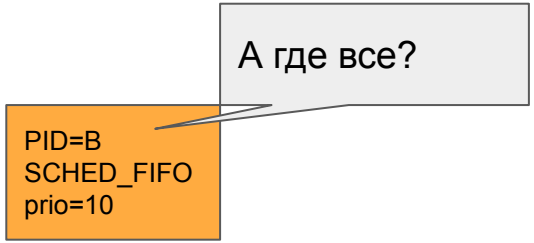
В pthread есть аналогичные функции

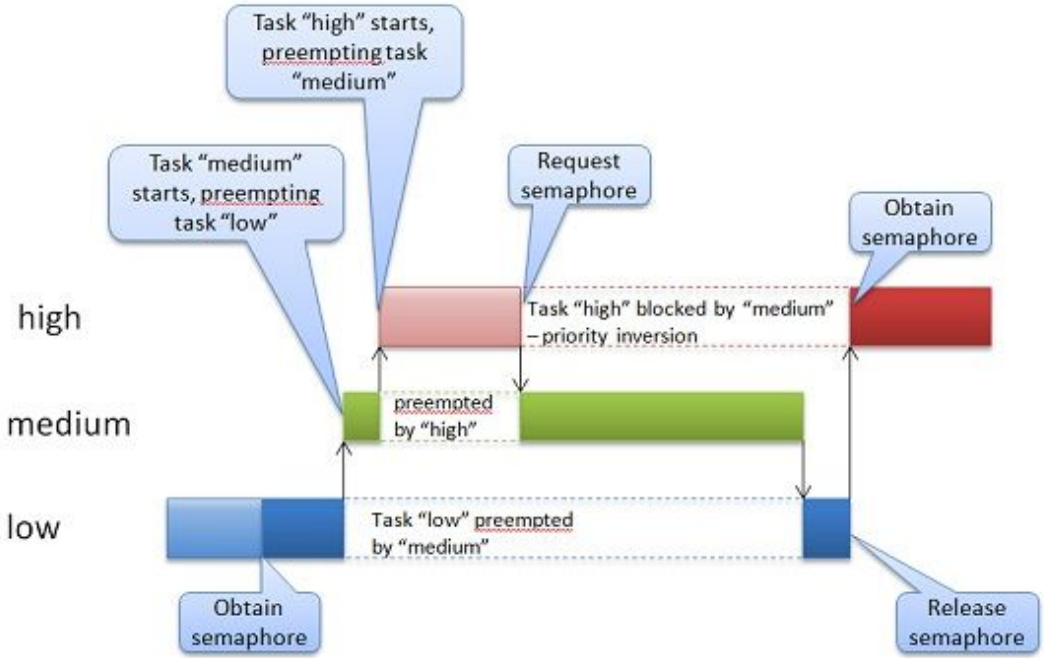
Инверсия приоритетов











“Первый баг на Марсе” https://habr.com/ru/company/pvs-studio/blog/315496/

Решение: наследование приоритетов

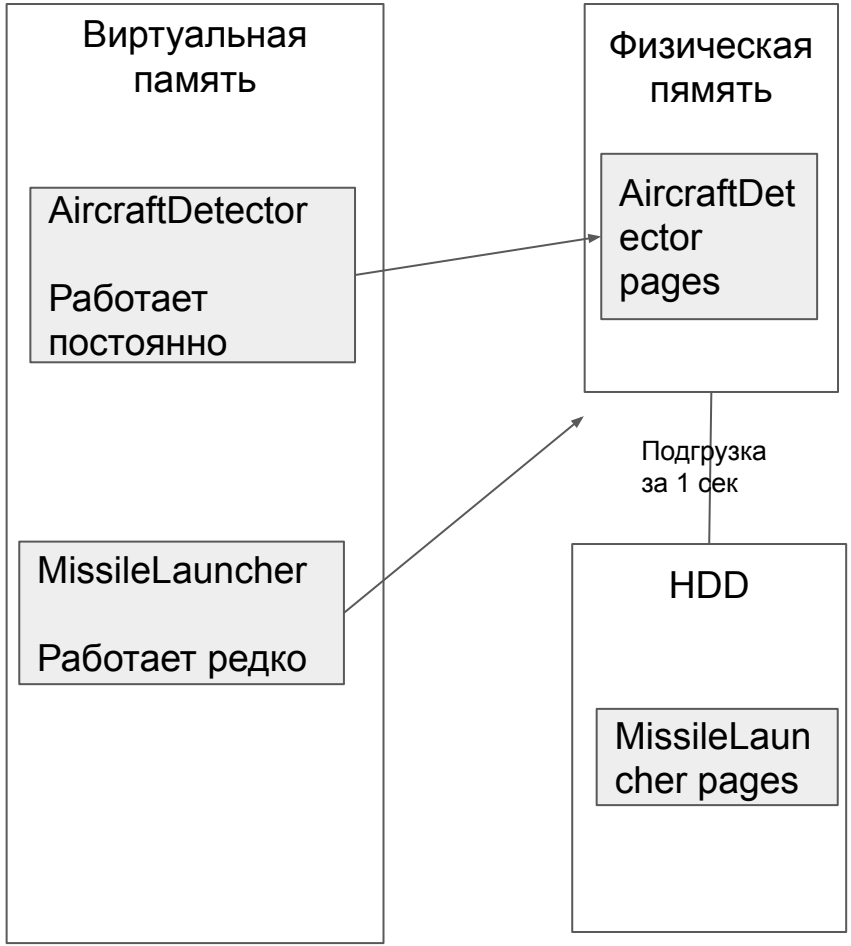
У процесса, захватившего ресурс, повышается приоритет
до максимального из тех, кто ожидает ресурс

```
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *
restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
int protocol);
```

c protocol = PTHREAD_PRIO_INHERIT

c protocol = PTHREAD_PRIO_PROTECT
можно задать блокировке приоритет (процесс,
захвативший блокировку, будет всегда повышать
свой приоритет до уровня блокировки)

Недетерминированность памяти



Требование: старт ракеты должен произойти не позднее чем через 0.5 секунды после обнаружения цели

AircraftDetector обнаруживает цель за 0.1 сек
MissileLauncher стартует ракету за 0.1 сек
Оверхед на передачу сообщений 0.1 сек

Общая задержка: 0.3 сек — OK

Если при передаче управления к MisseleLaucher его страницы будут выгружены на диск, к задержке прибавится время подгрузки:

0.3 + 1 сек — Не OK.

Блокировка страниц памяти

Можно запретить ядру выгружать те или иные страницы адресного пространства

```
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int mlock2(const void *addr, size_t len, int flags);
int munlock(const void *addr, size_t len);

int mlockall(int flags);
int munlockall(void);
```

Можно узнать, находится ли страница в памяти

```
int mincore(void *addr, size_t length, unsigned char *vec);
```