XV6 Analysis - Fork

Course	操作系统
Created	@October 3, 2021 11:17 PM
Status	Completed

小组成员：邱浩辰 魏怡健 杨泽超

主讲：杨泽超

本文档可在线阅读，体验更佳：

https://josephqiu.notion.site/XV6-Analysis-Fork-b6bb7323a8b8411db9eb73acbd0f4d9d

简介

在 `main.c` 的 `main()` 中，系统在初始化后用 `userinit()` 启动第一个用户进程：PCB 初始化、用 `scheduler` 调度、用 `swtch` 切换上下文，最后开始运行 `init.c` 的代码段，利用 `fork()` 函数创建子进程、启动程序。本次实例分析围绕此处调用的 `fork()` 函数：

```
// init.c:24

printf(1, "init: starting sh\n");
pid = fork();
```

我们将分析 `fork()` 函数调用过程中的细节，分析其作用，并关注下列重点问题：

> 💡 用户态程序 main/init.c 调用的 fork 的定义在哪里?和一般的函数定义有什么不同?

> 💡 为什么子进程和父进程一样都会返回 main/init.c？

> 💡 子进程是如何从 RUNNABLE 转换到 RUNNING 状态的?

> 💡 main/init.c 调用 fork 后，是父进程先返回还是子进程先返回?

> 💡 对于父进程和子进程，fork 返回的 pid 相同么?为什么?

> 💡 子进程返回后，加载的程序是什么程序?

> 💡 wait 系统调用的功能?

目录

从用户态 fork() 进入内核态 fork()

C 语言函数声明的 `fork` 符号在链接器链接时绑定到汇编代码中的 `fork` 上，在用户程序中调用 `fork` 即进入汇编代码。

现在**回答第一个问题：**

> 💡 用户态程序 main/init.c 调用的 fork 的定义在哪里?和一般的函数定义有什么不同?

这个定义在 `usys.S` 中，用宏扩展后就是：

```
# usys.S

.globl fork
fork:
movl $SYS_fork, %eax #1, fork的系统调用号
int $T_SYSCALL       #64, 系统调用的中断号
ret
```

其将系统调用号写入 eax 寄存器，然后运行 int 陷入内核。

```
// trapasm.S

# vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

```
// trap.c:35

//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

```
// syscall.c:131

void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
```

```
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

表:

```
// syscall.c:107

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,

...
```

最后 `sysproc.c` 进入内核：

```
// syscall.c:10

int
sys_fork(void)
{
  return fork();
}

int
sys_exit(void)
{
  exit();
  return 0;  // not reached
}

int
sys_wait(void)
{
  return wait();
}
```

内核态 fork() 的执行

用 `myproc()` 从 CPU 读取父进程并拷贝到 `curproc`

```
struct proc *curproc = myproc();
```

分配新 PCB `np`，并用 `allocproc()` 初始化

- 从进程表中找到一个未使用（`UNUSED`）的进程
- 设置状态/pid
- 分配内核栈，设置 trap frame
- 初始化上下文

用 `copyuvm()` 分配页表，拷贝信息

这里注意 eax 赋值为 0。

```
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;

// Clear %eax so that fork returns 0 in the child.
np->tf->eax = 0;
```

从而，在后续调用子进程时恢复 trap frame，fork() 返回值是 0。

如此我们解答了第五个问题：

> 💡 对于父进程和子进程，fork 返回的 pid 相同么?为什么?

不同。子进程返回的 pid 是 0。

以及**第二个：**

> 💡 为什么子进程和父进程一样都会返回 main/init.c？

因为 trap frame 除了 eax 寄存器其余都一致，因而除了 fork() 返回值不同，它们回到的上下文都相同。

继续复制 open file 表：

```
for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
np->cwd = idup(curproc->cwd);

safestrcpy(np->name, curproc->name, sizeof(curproc->name));

pid = np->pid;
```

上锁后更改状态为 `RUNNABLE`，解锁并返回 pid

```
pid = np->pid;

acquire(&ptable.lock);

np->state = RUNNABLE;

release(&ptable.lock);

return pid;
```

下面是 `fork()` 完整代码：

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

fork() 调用完毕后

回到 syscall.c：

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

此处将 `pid` 返回给 `curproc->tf->eax`。

可知第四个问题的答案：

> 💡 main/init.c 调用 fork 后，是父进程先返回还是子进程先返回?

没有中断都情况下是父进程先返回。

虽然是父进程先返回，但是返回后很快可能遇到时钟中断而调用子进程，因此我们先来看一下子进程的启动。

启动子进程

回到父进程，发生时钟中断后，通过 `scheduler()` 调度，可能启动子进程。

从而我们回答了第三个问题：

> 💡 子进程是如何从 RUNNABLE 转换到 RUNNING 状态的?

后续调度时，`scheduler()` 将状态为 `RUNNABLE` 的子进程切换为 `RUNNING`。

恢复上下文后，从 `init` 中 `fork` 的位置接着运行，由于返回的 `pid` 为 `0`，所以进入以下分支：

```
if(pid == 0){
    exec("sh", argv);
    printf(1, "init: exec sh failed\n");
    exit();
}
```

`exec` 加载 `sh.c` 的可执行文件 `sh`。

从而回答了第六个问题：

> 💡 子进程返回后，加载的程序是什么程序?

是 **shell**。

它提供了接受用户输入、解释命令、调用内核处理等功能。

放一点sh.c的代码：

```
argc = 0;
  ret = parseredirs(ret, ps, es);
  while(!peek(ps, es, "|)&;")){
    if((tok=gettoken(ps, es, &q, &eq)) == 0)
      break;
    if(tok != 'a')
      panic("syntax");
    cmd->argv[argc] = q;
    cmd->eargv[argc] = eq;
    argc++;
    if(argc >= MAXARGS)
      panic("too many args");
    ret = parseredirs(ret, ps, es);
  }
  cmd->argv[argc] = 0;
  cmd->eargv[argc] = 0;
  return ret;
}

// NUL-terminate all the counted strings.
struct cmd*
nulterminate(struct cmd *cmd)
{
  int i;
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    return 0;

  switch(cmd->type){
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    for(i=0; ecmd->argv[i]; i++)
      *ecmd->eargv[i] = 0;
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd;
    nulterminate(rcmd->cmd);
    *rcmd->efile = 0;
    break;

  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    nulterminate(pcmd->left);
    nulterminate(pcmd->right);
    break;
```

返回父进程：wait() 系统调用

在 fork 成功情况下，父进程调用 `wait()` 等待子进程运行结束

```
while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
```

`wait()` 遍历 `ptable` 寻找子进程，回收释放处于僵尸态的子进程

找到后释放页表、清空 PCB。

```
// proc.c:283

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent != curproc)
        continue;
      havekids = 1;    // 找到子进程
      if(p->state == ZOMBIE){
        // 找到僵尸态的子进程
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
}
```

在父进程被 killed 中止或者无子进程时返回 `-1`

否则调用 `sleep()` 函数进入睡眠等待活着的子进程结束返回

```
    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

于是我们可以回答最后一个问题：

> 💡 wait 系统调用的功能?

它的基本功能是回收释放处于僵尸态的子进程。而在 `init.c` 中我们可以看到下面的死循环：

```
// init.c:22

for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
```

```
    printf(1, "zombie!\n");
  }
```

其中通过 fork 产生子进程并调用运行 shell，而 shell 返回后又会 fork 出新的子进程，如此循环。

在 shell 返回后，产生它的子进程通过 `exit()` 转为 `ZOMBIE` 态，而 `wait()` 的功能就是回收这些子进程。

关于处于 sleep 状态父进程的后续处理

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

exit() 时利用 wakeup() 唤醒父进程。

```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
```

```
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

进阶题：

> 💡 xv6 和 Linux 中调度器如何选择下一个要执行的进程?可选取一个 Linux 调度算法针对代码详细分析。

下面我们交给杨泽超同学发挥。

```
// kernel/sched/core.c:3103

static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /*
     * do_exit() calls schedule() with preemption disabled as an exception;
     * however we must fix that up, otherwise the next task will see an
     * inconsistent (higher) preempt count.
     *
     * It also avoids the below schedule_debug() test from complaining
     * about this.
     */
    if (unlikely(prev->state == TASK_DEAD))
        preempt_enable_no_resched_notrace();

    schedule_debug(prev);
```

```
if (sched_feat(HRTICK))
    hrtick_clear(rq);

local_irq_disable();

rcu_note_context_switch();

/*
 * Make sure that signal_pending_state()->signal_pending() below
 * can't be reordered with __set_current_state(TASK_INTERRUPTIBLE)
 * done by the caller to avoid the race with signal_wake_up().
 */
smp_mb__before_spinlock();

raw_spin_lock(&rq->lock);
lockdep_pin_lock(&rq->lock);

rq->clock_skip_update <<= 1; /* promote REQ to ACT */

switch_count = &prev->nivcsw;

if (!preempt && prev->state)
{

    if (unlikely(signal_pending_state(prev->state, prev)))
    {
        prev->state = TASK_RUNNING;
    }
    else
    {
        deactivate_task(rq, prev, DEQUEUE_SLEEP);

        prev->on_rq = 0;

        if (prev->flags & PF_WQ_WORKER) {
            struct task_struct *to_wakeup;

            to_wakeup = wq_worker_sleeping(prev);
            if (to_wakeup)
                try_to_wake_up_local(to_wakeup);
        }
    }
    switch_count = &prev->nvcsw;
}

if (task_on_rq_queued(prev))
    update_rq_clock(rq);
next = pick_next_task(rq, prev);

clear_tsk_need_resched(prev);

clear_preempt_need_resched();

rq->clock_skip_update = 0;

if (likely(prev != next))
{
    rq->nr_switches++;
    rq->curr = next;
    ++*switch_count;

    trace_sched_switch(preempt, prev, next);

    rq = context_switch(rq, prev, next); /* unlocks the rq */
}
else
{
    lockdep_unpin_lock(&rq->lock);
    raw_spin_unlock_irq(&rq->lock);
}
```

```
    balance_callback(rq);
}
STACK_FRAME_NON_STANDARD(__schedule); /* switch_to() */
```

```
// kernel/sched/core.c:3068

/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev)
{
    const struct sched_class *class = &fair_sched_class;
    struct task_struct *p;

    /*
     * Optimization: we know that if all tasks are in
     * the fair class we can call that function directly:
     */

    if (likely(prev->sched_class == class &&
           rq->nr_running == rq->cfs.h_nr_running))
    {
        p = fair_sched_class.pick_next_task(rq, prev);

        if (unlikely(p == RETRY_TASK))
            goto again;

        if (unlikely(!p))
            p = idle_sched_class.pick_next_task(rq, prev);

        return p;
    }

again:
    for_each_class(class)
    {
        p = class->pick_next_task(rq, prev);
        if (p)
        {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }

    BUG();
```

```
// SPDX-License-Identifier: GPL-2.0
/*
 * stop-task scheduling class.
 *
 * The stop task is the highest priority task in the system, it preempts
 * everything and will be preempted by nothing.
 *
 * See kernel/stop_machine.c
 */
#include "sched.h"

#ifdef CONFIG_SMP
static int
select_task_rq_stop(struct task_struct *p, int cpu, int flags)
{
```

```
  return task_cpu(p); /* stop tasks as never migrate */
}

static int
balance_stop(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
  return sched_stop_runnable(rq);
}
#endif /* CONFIG_SMP */

static void
check_preempt_curr_stop(struct rq *rq, struct task_struct *p, int flags)
{
  /* we're never preempted */
}

static void set_next_task_stop(struct rq *rq, struct task_struct *stop, bool first)
{
  stop->se.exec_start = rq_clock_task(rq);
}

static struct task_struct *pick_task_stop(struct rq *rq)
{
  if (!sched_stop_runnable(rq))
    return NULL;

  return rq->stop;
}

static struct task_struct *pick_next_task_stop(struct rq *rq)
{
  struct task_struct *p = pick_task_stop(rq);

  if (p)
    set_next_task_stop(rq, p, true);

  return p;
}

static void
enqueue_task_stop(struct rq *rq, struct task_struct *p, int flags)
{
  add_nr_running(rq, 1);
}

static void
dequeue_task_stop(struct rq *rq, struct task_struct *p, int flags)
{
  sub_nr_running(rq, 1);
}

static void yield_task_stop(struct rq *rq)
{
  BUG(); /* the stop task should never yield, its pointless. */
}

static void put_prev_task_stop(struct rq *rq, struct task_struct *prev)
{
  struct task_struct *curr = rq->curr;
  u64 delta_exec;

  delta_exec = rq_clock_task(rq) - curr->se.exec_start;
  if (unlikely((s64)delta_exec < 0))
    delta_exec = 0;

  schedstat_set(curr->se.statistics.exec_max,
      max(curr->se.statistics.exec_max, delta_exec));

  curr->se.sum_exec_runtime += delta_exec;
  account_group_exec_runtime(curr, delta_exec);

  curr->se.exec_start = rq_clock_task(rq);
```

```
  cgroup_account_cputime(curr, delta_exec);
}

/*
 * scheduler tick hitting a task of our scheduling class.
 *
 * NOTE: This function can be called remotely by the tick offload that
 * goes along full dynticks. Therefore no local assumption can be made
 * and everything must be accessed through the @rq and @curr passed in
 * parameters.
 */
static void task_tick_stop(struct rq *rq, struct task_struct *curr, int queued)
{
}

static void switched_to_stop(struct rq *rq, struct task_struct *p)
{
  BUG(); /* its impossible to change to this class */
}

static void
prio_changed_stop(struct rq *rq, struct task_struct *p, int oldprio)
{
  BUG(); /* how!?, what priority? */
}

static void update_curr_stop(struct rq *rq)
{
}

/*
 * Simple, special scheduling class for the per-CPU stop tasks:
 */
DEFINE_SCHED_CLASS(stop) = {

  .enqueue_task   = enqueue_task_stop,
  .dequeue_task   = dequeue_task_stop,
  .yield_task   = yield_task_stop,

  .check_preempt_curr = check_preempt_curr_stop,

  .pick_next_task   = pick_next_task_stop,
  .put_prev_task    = put_prev_task_stop,
  .set_next_task          = set_next_task_stop,
```

```
// kernel/sched/idle.c:436-547

static void set_next_task_idle(struct rq *rq, struct task_struct *next, bool first)
{
  update_idle_core(rq);
  schedstat_inc(rq->sched_goidle);
  queue_core_balance(rq);
}

#ifdef CONFIG_SMP
static struct task_struct *pick_task_idle(struct rq *rq)
{
  return rq->idle;
}
#endif

struct task_struct *pick_next_task_idle(struct rq *rq)
{
  struct task_struct *next = rq->idle;

  set_next_task_idle(rq, next, true);
```

```
  return next;
}
```

```
// kernel/sched/fair.c:7261

struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
  struct cfs_rq *cfs_rq = &rq->cfs;
  struct sched_entity *se;
  struct task_struct *p;
  int new_tasks;

again:
  if (!sched_fair_runnable(rq))
    goto idle;

#ifdef CONFIG_FAIR_GROUP_SCHED
  if (!prev || prev->sched_class != &fair_sched_class)
    goto simple;

  /*
   * Because of the set_next_buddy() in dequeue_task_fair() it is rather
   * likely that a next task is from the same cgroup as the current.
   *
   * Therefore attempt to avoid putting and setting the entire cgroup
   * hierarchy, only change the part that actually changes.
   */

  do {
    struct sched_entity *curr = cfs_rq->curr;

    /*
     * Since we got here without doing put_prev_entity() we also
     * have to consider cfs_rq->curr. If it is still a runnable
     * entity, update_curr() will update its vruntime, otherwise
     * forget we've ever seen it.
     */
    if (curr) {
      if (curr->on_rq)
        update_curr(cfs_rq);
      else
        curr = NULL;

      /*
       * This call to check_cfs_rq_runtime() will do the
       * throttle and dequeue its entity in the parent(s).
       * Therefore the nr_running test will indeed
       * be correct.
       */
      if (unlikely(check_cfs_rq_runtime(cfs_rq))) {
        cfs_rq = &rq->cfs;

        if (!cfs_rq->nr_running)
          goto idle;

        goto simple;
      }
    }

    se = pick_next_entity(cfs_rq, curr);
    cfs_rq = group_cfs_rq(se);
  } while (cfs_rq);

  p = task_of(se);

  /*
```

```
	 * Since we haven't yet done put_prev_entity and if the selected task
	 * is a different task than we started out with, try and touch the
	 * least amount of cfs_rqs.
	 */
	if (prev != p) {
		struct sched_entity *pse = &prev->se;

		while (!(cfs_rq = is_same_group(se, pse))) {
			int se_depth = se->depth;
			int pse_depth = pse->depth;

			if (se_depth <= pse_depth) {
				put_prev_entity(cfs_rq_of(pse), pse);
				pse = parent_entity(pse);
			}
			if (se_depth >= pse_depth) {
				set_next_entity(cfs_rq_of(se), se);
				se = parent_entity(se);
			}
		}

		put_prev_entity(cfs_rq, pse);
		set_next_entity(cfs_rq, se);
	}

	goto done;
simple:
#endif
	if (prev)
		put_prev_task(rq, prev);

	do {
		se = pick_next_entity(cfs_rq, NULL);
		set_next_entity(cfs_rq, se);
		cfs_rq = group_cfs_rq(se);
	} while (cfs_rq);

	p = task_of(se);

done: __maybe_unused;
#ifdef CONFIG_SMP
	/*
	 * Move the next running task to the front of
	 * the list, so our cfs_tasks list becomes MRU
	 * one.
	 */
	list_move(&p->se.group_node, &rq->cfs_tasks);
#endif

	if (hrtick_enabled_fair(rq))
		hrtick_start_fair(rq, p);

	update_misfit_status(p, rq);

	return p;

idle:
	if (!rf)
		return NULL;

	new_tasks = newidle_balance(rq, rf);

	/*
	 * Because newidle_balance() releases (and re-acquires) rq->lock, it is
	 * possible for any higher priority task to appear. In that case we
	 * must re-start the pick_next_entity() loop.
	 */
	if (new_tasks < 0)
		return RETRY_TASK;

	if (new_tasks > 0)
		goto again;
```

```
  /*
   * rq is about to be idle, check if we need to update the
   * lost_idle_time of clock_pelt
   */
  update_idle_rq_clock_pelt(rq);

  return NULL;
}

static struct task_struct *__pick_next_task_fair(struct rq *rq)
{
  return pick_next_task_fair(rq, NULL, NULL);
}


static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
              unsigned long delta_exec)
{
    unsigned long delta_exec_weighted;
    curr->sum_exec_runtime += delta_exec;
    delta_exec_weighted = delta_exec;

    if (unlikely(curr->load.weight != NICE_0_LOAD)) {
        delta_exec_weighted = calc_delta_fair(delta_exec_weighted, &curr->load);
    }
    curr->vruntime += delta_exec_weighted;
}
```

分工

文稿：邱浩辰 魏怡健 杨泽超

演示文档：邱浩辰 魏怡健

主讲：杨泽超