گام ۱: ۱. ابتدا به بررسی داده ساختار semaphore می‌پردازیم.

```
struct semaphore {
        raw_spinlock_t          lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

دادهساختار آن به شکل روبرو است:

- lock - spinlock for a semaphore data protection
- count - amount available resources
- wait_list - list of processes which are waiting to acquire a lock

wait_list

لیستی از افرادی است که در انتظار گرفتن lock هستند. یعنی منتظر اند به ناحیه بحرانی خود وارد شوند. این داده ساختار به صورت یک لیست پیوندی است و اشارهگر به بعدی و قبلی خود را در خود نگه میدارد.

```
struct list_head {
        struct list_head *next, *prev;
};
```

Count

این متغیر مشخص میکند که حداکثر چند نفر میتوانند به طور هم زمان از منبعی که با این lock از آن حفاظت شده استفاده کنند.

lock

قفلی است که برای محافظت از منبع/منابع موجود از آن استفاده میکنیم.

۲-راه حل کلی دو تابع up و down استفاده از قفل ها برای محافظت از ناحیه بحرانی می باشد.در تابع up ماکروهای raw_spin_lock_irqsave و raw_spin_unlock_irqrestore و در تابع down ماکروهای raw_spin_lock_irqsave و raw_spin_lock_irqrestore همانند ماکروهای spin_lock و spin_unlock از counter سمافور محافظت می کنند، با این تفاوت که مقدار فعلی flag های interrupt را ذخیره و بازیابی می کنند. در تابع down، پراسس منتظر دریافت قفل، در یک لوپ بی نهایت قرار می گیرد و تا زمانی که با سیگنال به آن وقفه داده نشده و مهلت زمانی آن تمام نشده و کار پراسس در حال اجرا به پایان نرسیده،در این لوپ قرار می گیرد. وقتی که تابع up صدا زده می شود قفل رها می شود و پراسس منتظر از لوپ خارج می شود و قفل را بدست می آورد.

۳-ساختار تابع up به شکل زیر است:

```
void up(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

ساختار کلی این تابع، تقریبا شبیه تابع down است اما تفاوت هایی بین آن ها وجود دارد.مثلا اگر لیست waiter ها (کسانی که در انتظار ورود به ناحیه ی بحرانی هستند)خالی باشد؛ شمارنده ی سمافور را یک واحد افزایش میدهیم.اگر این لیست خالی نباشد؛ تابع up-- را روی سمافور مربوطه صدا میزنیم.در واقع اگر لیست کسانی که در انتظار ورود به ناحیه ی بحرانی هستند(wait_list) خالی نباشد؛ ما نیاز

داریم که اولین task از لیست قفل را در اختیار بگیرد.ماکروهای raw_spin_lock_irqsave و
raw_spin_unlock_irqrestore از شمارنده (counter)سمافور داده شده حفاظت میکنند.در واقع هر دو این ماکروها مانند
ماکروهای spin_lock و spin_unlock عمل میکنند با این تفاوت که آن ها میتوانند مقدار فعلی flag های intrrupt را ذخیره و
باز یابی کنند.واضح است تابع list_empty برای تشخیص خالی بودن لیست waiterها مورد استفاده قرار میگیرد.حال به سراغ تعریف
تابع up-- میرویم که به شکل زیر است:

```
static noinline void __sched __up(struct semaphore *sem)
{
struct semaphore_waiter *waiter =
list_first_entry(&sem->wait_list,struct,semaphore_waiter, list);
list_del(&waiter->list);
waiter->up = true
wake_up_process(waiter->task)}
```

در این تابع اولین task از لیست waiter ها را میگیریم(list_first_entry).آن را در یک اشاره گر از استراکت
semaphore_waiter ذخیره میکنیم.آن را از لیست حذف میکنیم(list_del)
waitr.upرا نیز true میکنیم.ازاین قسمت به بعد حلقه ی بی نهایت تابع common-down_ متوقف خواهد شد.(حلقه ی داخل این
تابع تا زمانی که up مقدار ۱ نداشته باشد؛ ادامه می یابد.)تابع wake_up_process در انتهای این تابع فراخوانی میشود.تابع
scheduled_timeout که در حلقه ی بی نهایت common_down ـ صدا زده می شود؛ باعث می شود task فعلی تا زمانی که
time out آن داده نشده است؛ به حالت sleep بماند.بنابراین با تابع wake_up_process باید ان را بیدار کنیم.تعریف تابع
wake_up_process و استراکچر semaphore waiter در زیر آمده است:

```
int wake_up_process(struct task_struct *p)
{
WARN_ON(task_is_stopped_or_traced(p));
;(return try_to_wake_up(p, TASK_NORMAL, 0
{

struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

۴- به بررسی ساختار تابع down می‌پردازیم شکل کلی این تابع به صورت زیر است.

```
void down(struct semaphore *sem)
{
         unsigned long flags;
        raw_spin_lock_irqsave(&sem->lock, flags);
         if (likely(sem->count > 0))
        sem->count--;
         else
        __down(sem);
```

```
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

تابع down برای به دست آوردن سمافور داده شده می باشد. تابع down_interruptible سعی می کند که سمافور را به دست آورد.اگر تلاش آن نتیجه بخش بود، متغیر count سمافور داده شده،کاهش می یابد و قفل حاصل می شود، به عبارت دیگر، task به blocked state میرود و پرچم TASK_INTERRUPTIBLE، set می شود.که این به معنای این است که پراسس ممکن است با سیگنال به ruined state بازگردد.تابع down_killable همانند down_interruptible کار می کند ولی پرچم TASK_KILLABLE را برای پراسس کنونی set می کند که به این معنا می باشد که پراسس منتظر، ممکن است توسط سیگنال kill دچار وقفه شود.تابع down_trylock مشابه تابع spin_trylock است.این تابع سعی می کند قفل را بگیرد و درصورت ناموفق بودن عملیات، خارج شود.در این حالت پراسسی که می خواهد قفل را بگیرد منتظر نخواهد شد.تابع آخر down_timeout سعی می کند قفل را بدست آورد.این در حالت انتظار، وقتی مهلت زمانی اختصاص داده شده تمام شود، دچار وقفه می شود.این مهلت زمانی در jiffies است.

```
int  down_interruptible(struct semaphore *sem);
int  down_killable(struct semaphore *sem);
int  down_trylock(struct semaphore *sem);
int  down_timeout(struct semaphore *sem, long jiffies);
```

متغیر flags در اول تابع به ماکرو های raw_spin_lock_irqsave و raw_spin_lock_irqrestore داده می شود که از شمارنده سمافور داده شده محافظت می کنند.درواقع هر دو ماکرو همانند ماکرو های spin_lock و spin_unlock کار می کنند ولی علاوه بر آن، مقدار کنونی وقفه flag را نگهداری کرده و در صورت نیاز آن را برمیگردانند و دیگر وقفه ها را از کار می اندازند.کار اصلی این تابع بین این دو ماکرو صورت می گیرد.مقدار counter سمافور با صفر مقایسه می شود، اگر بزرگتر بود،آن را کاهش می دهیم.این به این معناست که ما قفل را قبلا گرفته بودیم. به بیان دیگر counter صفر است و تمام منابع موجود، تمام شده اند و ما نیاز داریم برای گرفتن قفل صبر کنیم.
تابع __down به صورت زیر تعریف شده است که این تابع،تابع __down_common را با سه متغیر سمافور، پرچم task و مهلت زمانی صدا می زند.

```
static noinline void __sched __down(struct semaphore *sem)
{
        __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

قبل از نشان دادن تابع __down_common به این توجه می کنیم که سه تابع زیر هم این تابع را صدا زده اند.

```
static noinline int __sched __down_interruptible(struct semaphore *sem)
{
        return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
static noinline int __sched __down_killable(struct semaphore *sem)
{
        return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}
static noinline int __sched __down_timeout(struct semaphore *sem, long timeout)
{
        return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
```

حال به توضیح تابع __down_common می پردازیم:

```
struct task_struct *task = current;
```

```
struct semaphore_waiter waiter;
```

این تابع دو متغیر محلی دارد،که اولی نشانگر task کنونی برای پردازنده محلی که می خواهد قفل را بگیرد هست.current یک ماکرو است که در current.h تعریف شده:

```
#define current get_current()
```

تابع get_current،کار(task) کنونی per-cpu را برمیگرداند.

```
DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
        return this_cpu_read_stable(current_task);
}
```

متغیر دومی waiter است که سر لیست semaphore.wait_list را نشان می دهد.

```
struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

سپس ما task کنونی را به wait_list اضافه می کنیم

```
list_add_tail(&waiter.list, &sem->wait_list);
waiter.task = task;
waiter.up = false;
```

سپس وارد حلقه بی نهایت میشویم:

```
for (;;) {
        if (signal_pending_state(state, task))
        goto interrupted;

        if (unlikely(timeout <= 0))
        goto timed_out;

        __set_task_state(task, state);

        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);

        if (waiter.up)
        return 0;
}
```

در قطعه کد قبلی مقدار waiter.up را false کردیم بنابراین task تا وقتی up است می ماند و true نمی شود.این حلقه با بررسی این که task کنونی در وضعیت pending (شامل پرچم TASK_WAKEKILL یا TASK_INTERRUPTIBLE) است یا نه آغاز می شود.یک task ممکن است در حین انتظار دچار وقفه شود.

```
static inline int signal_pending_state(long state, struct task_struct *p)
{
        if (!(state & (TASK_INTERRUPTIBLE | TASK_WAKEKILL)))
        return 0;
```

```
        if (!signal_pending(p))
        return 0;

        return (state & TASK_INTERRUPTIBLE) || __fatal_signal_pending(p);
}
```

وضعیت bitmask شامل بیتهای TASK_INTERRUPTIBLE یا TASK_WAKEKILL است و اگر bitmask شامل این بیت نباشد، ما خارج می شویم.سپس چک می کنیم که آیا task داده شده سیگنال pending دارد یا نه اگر نه خارج می شویم.اگر task سیگنال pending داشت، به interrupted label می رویم.

```
interrupted:
        list_del(&waiter.list);
   return -EINTR;
```

task را از لیست منتظران پاک می کنیم و کد اِرور(-EINTR) را برمیگردانیم.

```
if (unlikely(timeout <= 0))
   goto timed_out;
```

اگرtask سیگنال pending نداشت، timeout داده شده را چک می کنیم و اگر کمتر یا مساوی صفر بود به time_out می رویم.

```
timed_out:
        list_del(&waiter.list);
   return -ETIME;
```

همانند بخش interrupted، از لیست منتظران task را پاک می کنیم ولی اِرور -ETIME را برمیگردانیم. اگر task، سیگنال pending نداشت و timeout هنوز تمام نشده بود،state داده شده به task داده شده set می شود.

```
__set_task_state(task, state);
```

تابعschedule_timeout صدا زده می شود:

```
raw_spin_unlock_irq(&sem->lock);
timeout = schedule_timeout(timeout);
raw_spin_lock_irq(&sem->lock);
```

تابع task، schedule_timeout کنونی را تا timeout داده شده، به خواب می برد.
این ها درمورد __down_common بود.
یک task که می خواهد قفل را بگیرد، (که قفل قبلا توسط یک task دیگر گرفته شده) تا زمانی که با سیگنال به آن وقفه داده نشود یا مهلت زمانی آن تمام نشود یا taskیی که قفل را گرفته آن را رها نکند، در حلقه بی نهایت باقی می ماند.