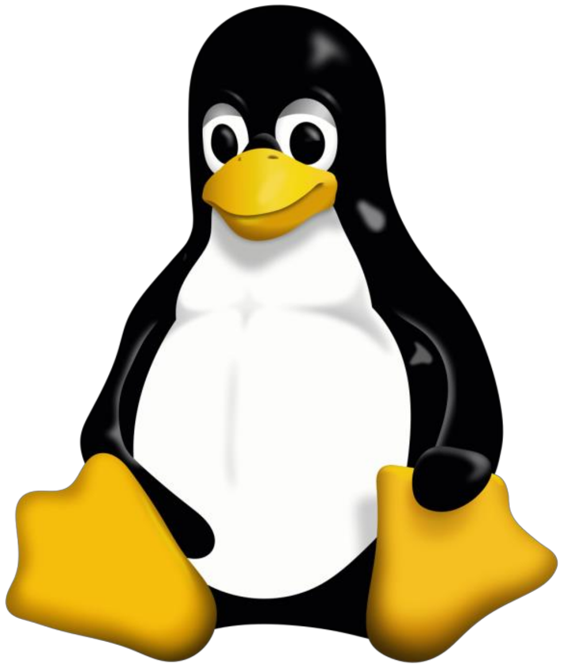
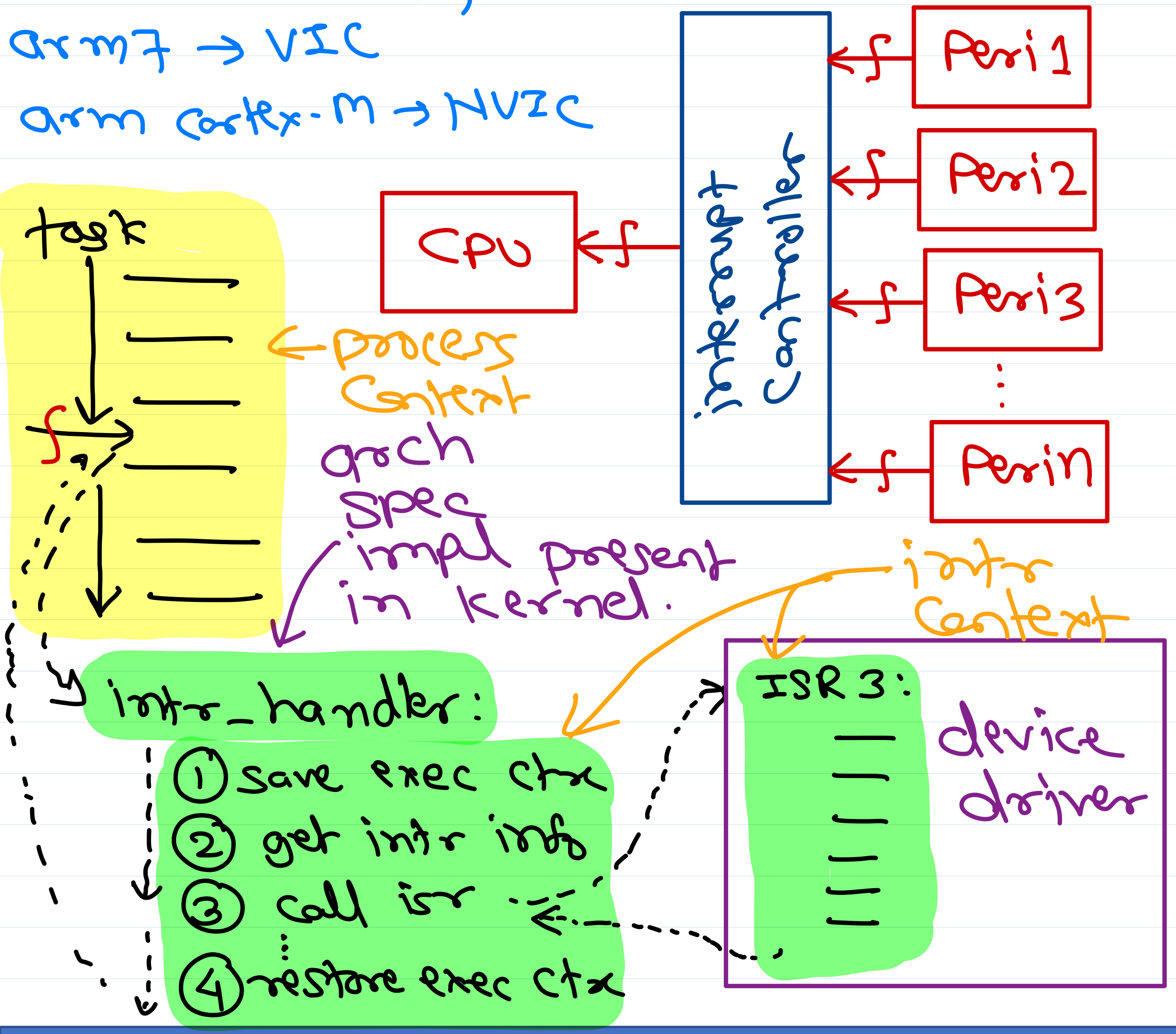


Linux Device Driver

Sunbeam Infotech

Interrupt Handling in Linux

x86 → PIC-8259
arm7 → VIC
arm cortex-M → NVIC



ISR impl:

irqreturn_t isr_fn(int irq, void *param);
- → IRQ_HANDLED → intr handled by this ISR.
- → IRQ_NONE → intr not handled by this ISR.
  - ↳ kernel calls next isr on same line.

Register ISR:

request_irq(irq, isr_fn, flags, name, param);

name ↓ cat /proc/interrupts

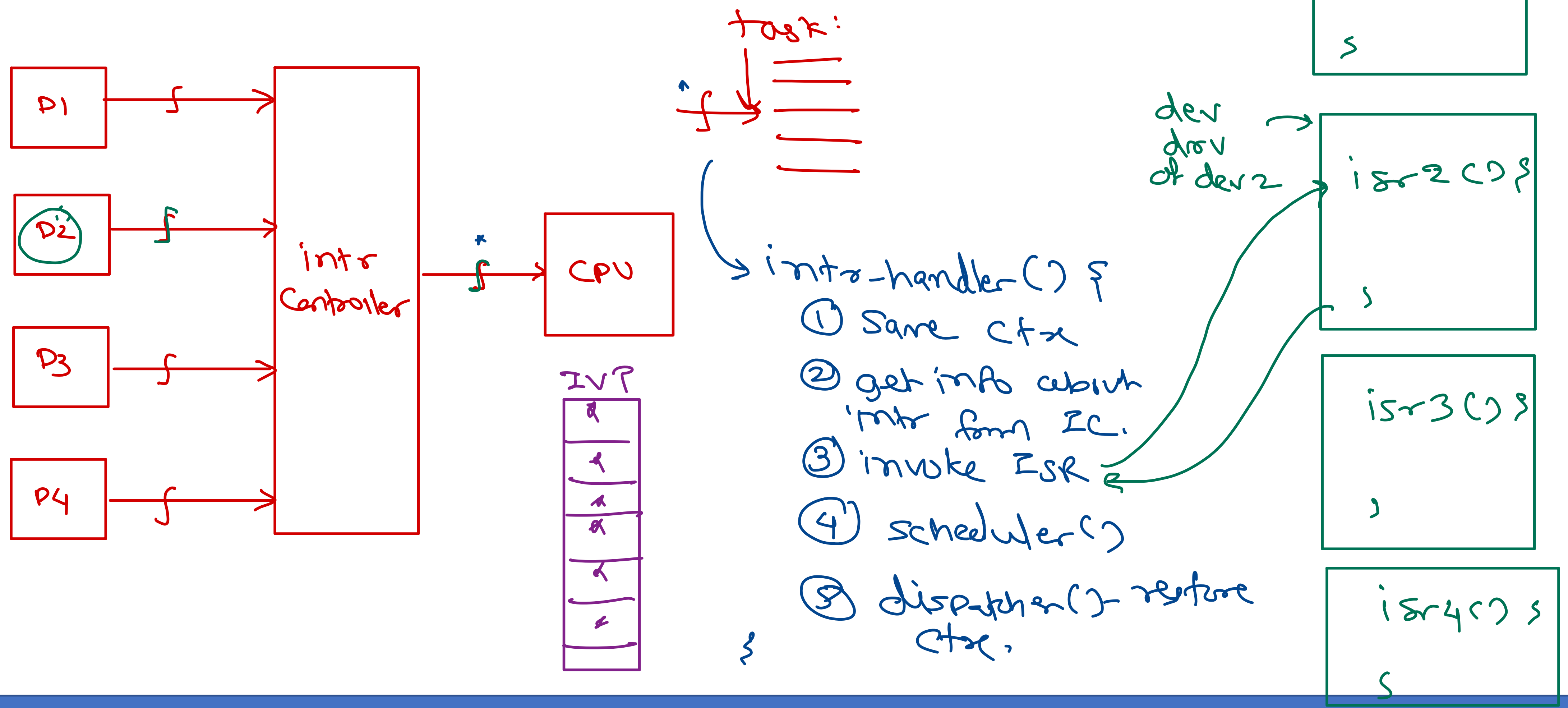
flags:
1. IRQF_DISABLED
2. IRQF_TIMER
3. IRQF_SAMPLE_RANDOM → random entropy pool → true random nums ↳ /dev/random
4. IRQF_SHARED
   - ↳ same irq line is shared for multiple hw devices.
   - e.g. com1, com2, com3, com4 ← serial ports
     - ↓ 4, ↓ 3, ↓ 4, ↓ 3

Unregister ISR:

free_irq(irq, param);

Interrupt

- Interrupts are special signals sent from device to CPU.
- Interrupt handling is architecture specific.



Interrupt handling

- Interrupt is sent from the device to the PIC.
- PIC inform CPU about interrupt through interrupt line.
- CPU pause current task execution and execute interrupt handler.
- Interrupt handler does following
  - Save current task context on stack.
  - Get interrupt details from PIC.
  - Call ISR to handle the interrupt.
  - Invoke scheduler.
  - Restore the task context.
- In Linux there are two execution context.
  - Process context
    - User space process or kernel thread context. May block.
  - Interrupt context
    - Interrupt handler and ISR execution context.
    - Atomic context: cannot block.

Interrupt handling in Linux

- Since interrupt context cannot block, handler/ISR should return immediately.
- Heavy processing and/or blocking tasks should be deferred.
- Linux divides interrupt handling into two parts
  - Top half → ISR → Intr Context
    - Run immediately when interrupt arrives.
    - Do time critical and non-blocking task like interrupt acknowledgement.
    - Cannot be pre-empted by another interrupt from same device.
  - Bottom half → Soft IRQ, Tasklet, Work Queue
    - Variety of bottom half implementations in Linux kernel.
    - Execute later – in interrupt context or process context.
    - Do heavy processing and/or blocking tasks.
    - Can be pre-empted by interrupt (top-half).
- Interrupt handling must be done in corresponding device driver.
  - Driver should implement top-half and/or bottom-half as per requirement.
  - Linux kernel ensure uniform programming model irrespective of architecture.

Implementing top half

- Two step process
  - Implement ISR.
  - Register ISR.
- ISR registration
  - #include <linux/interrupt.h>
  - int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev);
    - irq: interrupt number
    - handler: typedef irqreturn_t (*irq_handler_t)(int, void *);
    - flags:
      - IRQF_DISABLED
      - IRQF_SAMPLE_RANDOM
      - IRQF_TIMER
      - IRQF_SHARED
    - name: device name /proc/irq and /proc/interrupts
    - dev: extra information to be passed to handler.
    - Returns 0 on success or –EBUSY if interrupt line is already in use.
  - request_irq() may block and should not be called from interrupt context. Typically called when opening the device for processing or module initialization.

irq_return_t my_isr(int irq, void *param) {

3

request_irq()

/dev/random

multiple device instances – private obj to keep each device info

e.g. struct serial_info{
int irq;
int ioaddr;
mutex m;
...
};

struct private_struct devices[4];

irq=4, io=0x3F8, ⋮, com1	0
irq=3, io=0x2F8, ⋮, com2	1
irq=4, io=0x3E8, ⋮, com3	2
irq=3, io=0x2E8, ⋮, com4	3

com1
request_irq(4, my_isr, IRQF_SHARED, "com1", &devices[0]);

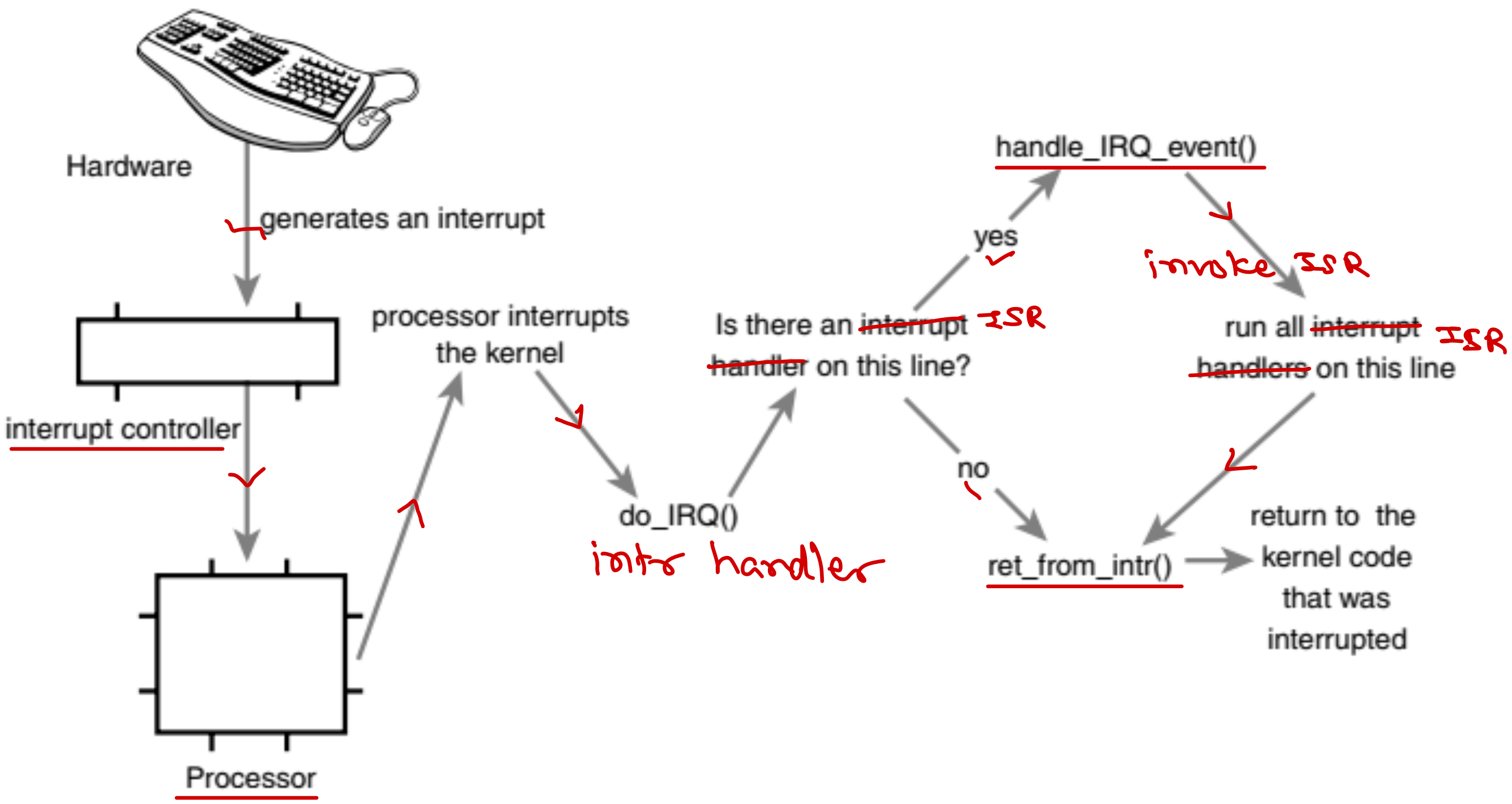
com3
request_irq(4, my_isr, IRQF_SHARED, "com3", &devices[2]);

Implementing top half

- ISR un-registration
  - Interrupt line must be released while unloading module or closing device.
  - void free_irq(unsigned int irq, void *dev);
- Implementing ISR
  - irqreturn_t my_intr_handler(int irq, void *dev);
    - irq: interrupt number
    - dev: extra param passed while request_irq()
    - returns IRQ_HANDLED or IRQ_NONE.
  - Should contain time-critical tasks and interrupt acknowledgement.
  - Also trigger bottom-half if required.
  - Should not sleep/block.
- Linux interrupt handlers are not re-entrant. Current interrupt line is disabled while execution of ISR.
- Shared interrupt handlers
  - Must pass unique dev param – typically device private struct.
  - ISR must check if interrupt is raised from the corresponding device before handling it.
  - Kernel execute all ISR registered on same interrupt line.

Interrupt handling

- Interrupt context
  - Atomic context.
  - One page kernel stack per processor.
- Interrupt execution



Interrupt control

- local_irq_disable(): Disables local interrupt delivery
- local_irq_enable(): Enables local interrupt delivery

current cpu

- local_irq_save(): Saves the current state of local interrupt delivery and then disables it
- local_irq_restore(): Restores local interrupt delivery to the given state
- disable_irq(): Disables the given interrupt line and ensures no handler on the line is executing
- enable_irq(): Enables the given interrupt line
- irqs_disabled(): Returns nonzero if local interrupt delivery is disabled; otherwise returns zero
- in_interrupt(): Returns nonzero if in interrupt context and zero if in process context
- in_irq(): Returns nonzero if currently executing an interrupt handler and zero otherwise

ISR

Bottom half

- If interrupt handling need to do heavy processing or blocking task, then driver must implement it in bottom half.
- General guideline for top and bottom half work division:
  - If the work is time sensitive, perform it in the interrupt handler. ISR
  - If the work is related to the hardware, perform it in the interrupt handler. ISR
  - If the work needs to ensure that another interrupt doesn't interrupt it, perform it in the interrupt handler. ISR
  - For everything else, consider performing the work in the bottom half.

top half

- Bottom half are executed after top half.
  - Immediately after top half in interrupt context.
  - In some specialized process context, when no other another high priority task is running.
- Types of bottom halves
  - BH - Removed in kernel 2.5
  - Task queue - Removed in kernel 2.5
  - Softirq - Added in kernel 2.3
  - Tasklet - Added in kernel 2.3
  - Work queue - Added in kernel 2.3

Softirq

- Softirqs are statically allocated at compile time.

```
struct softirq_action {
        void (*action)(struct softirq_action *);
};
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

32

- Softirqs are implemented for specialized sub-systems.
- Kernel 2.6.34 have 9 Softirqs implemented.

HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
BLOCK_SOFTIRQ	4	Block devices
TASKLET_SOFTIRQ	5	Normal priority tasklets
SCHED_SOFTIRQ	6	Scheduler
HRTIMER_SOFTIRQ	7	High-resolution timers
RCU_SOFTIRQ	8	RCU locking

Softirqs

- Softirqs must be triggered for the execution. This is called as “raising Softirq”. Mostly done from ISR.
- Pending Softirq are checked and executed in one of the following place – do_softirq().
  - In the return from hardware interrupt code path
  - In the ksoftirqd kernel thread (per processor)
  - In any code that explicitly checks for and executes pending softirqs, such as the networking subsystem
- Using Softirq
  - Currently only network and block subsystem is using Softirq directly.
  - It is not advised to use Softirqs directly.
  - Softirq can be registered using open_softirq() and can be triggered using raise_softirq().

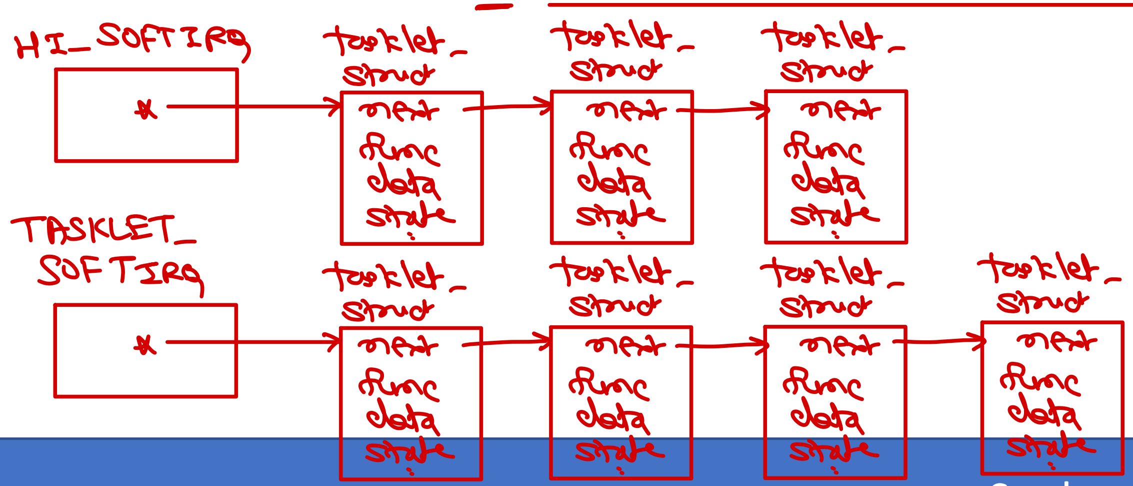
Tasklets

0 5

- Implemented on top of Softirqs i.e. HI_SOFTIRQ and TASKLET_SOFTIRQ.
- Tasklets are dynamic components and much easier to use.

```
struct tasklet_struct {
    struct tasklet_struct *next; /* next tasklet in the list */
    unsigned long state; /* state of the tasklet */
    atomic_t count; /* reference counter */
    void (*func)(unsigned long); /* tasklet handler function */
    unsigned long data; /* argument to the tasklet function */
};
```

- Tasklet state can be: 0, TASKLET_STATE_SCHED or TASKLET_STATE_RUN.



Tasklets

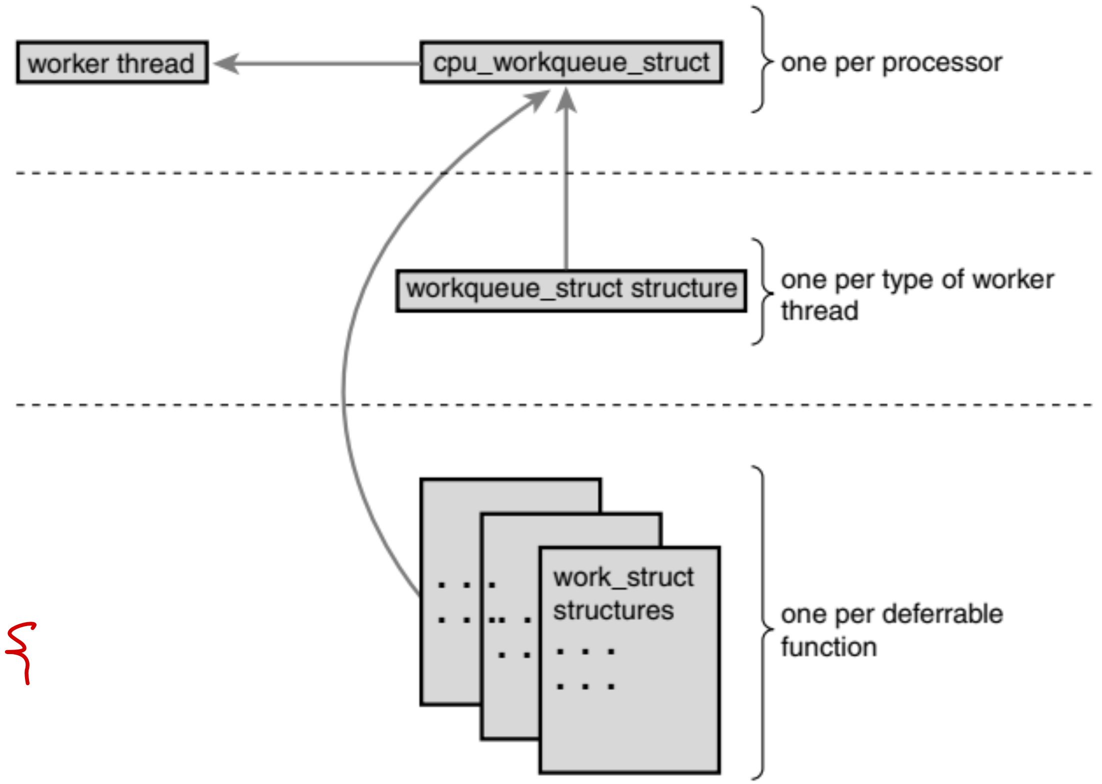
- Declare Tasklet statically.
    - DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);
- Declare & initialize Tasklet dyanmically.
    - struct tasklet_struct my_tasklet;
    - tasklet_init(t, tasklet_handler, dev); arg to tasklet func.
- Tasklet handler implementation
    - void tasklet_handler(unsigned long data) { … }
    - Like softirq, tasklet cannot sleep/block.
    - While executing tasklet, interrupts are enabled.
    - Tasklet are not re-entrant or execute concurrently.
- Trigger Tasklet
    - tasklet_schedule(&my_tasklet); tasklet_hi_schedule(&my_tasklet);
    - Change tasklet state to TASKLET_STATE_SCHED.
- Tasklet can be enabled/disabled explicitly.
    - tasklet_enable(&my_tasklet);
    - tasklet_disable(&my_tasklet);

Work queue

- Work queues defer work into a kernel thread. → kworker
- Always runs in process context – worker threads (per processor).
- Work queues are schedulable and can sleep/block.
- Usual alternative to work queues is kernel threads. However creating new kernel threads isn't advised.

```
struct cpu_workqueue_struct {
        spinlock_t lock; /* lock protecting this structure */
        struct list_head worklist; /* list of work */
        wait_queue_head_t more_work;
        struct work_struct *current_struct;
        struct workqueue_struct *wq; /* associated */
        task_t *thread; /* associated thread */
};
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
};
```

→ void func (data) {
}



Work queue

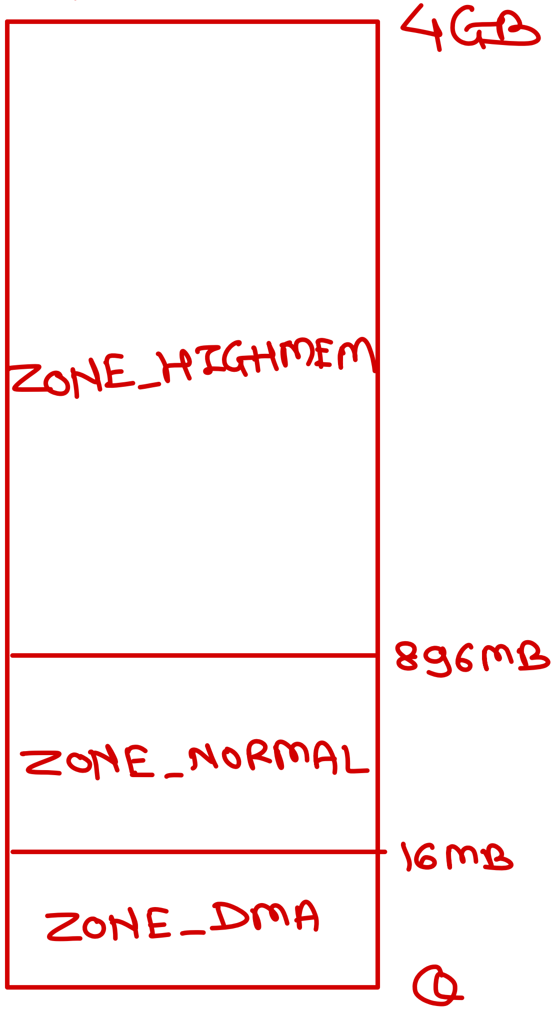
- Creating work
  - DECLARE_WORK(name, void (*func)(void *), void *data); // static
  - INIT_WORK(struct work_struct *work, void (*func)(void *), void *data); // dynamic
- Work handler
  - void work_handler(void *data) { … } → may sleep
- Scheduling work
  - schedule_work(&work);
  - schedule_delayed_work(&work, delay);
- Ensure work completion
  - void flush_scheduled_work(void);
- Cancel scheduled work
  - int cancel_delayed_work(struct work_struct *work);

Control bottom half

- To enable/disable all bottom half (tasklet & work queue) processing
  - void local_bh_enable();
  - void local_bh_disable();
- Choosing bottom half
  - Softirq
    - Concurrent execution and need synchronization
    - Good for fast execution and high frequency use
    - Cannot sleep
  - Tasklet
    - Simplified programming
    - Not executed concurrently
    - Cannot sleep
  - Work queue
    - Can sleep
    - Higher overheads due to kernel thread & context switching

Kernel memory management

RAM (physical memory)

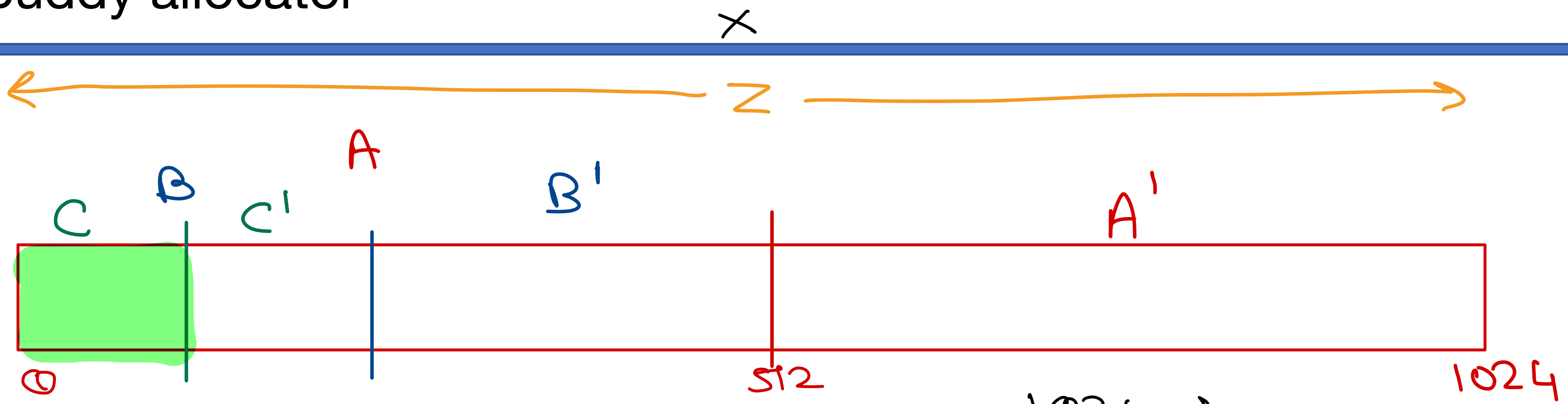


x86_32 architecture

Zonewise buddy allocator is employed to allocate contiguous physical pages as per requirement.

Buddy allocator keeps track of free pages.

Buddy allocator



(1) 128 → C

1024 →

512 → A'

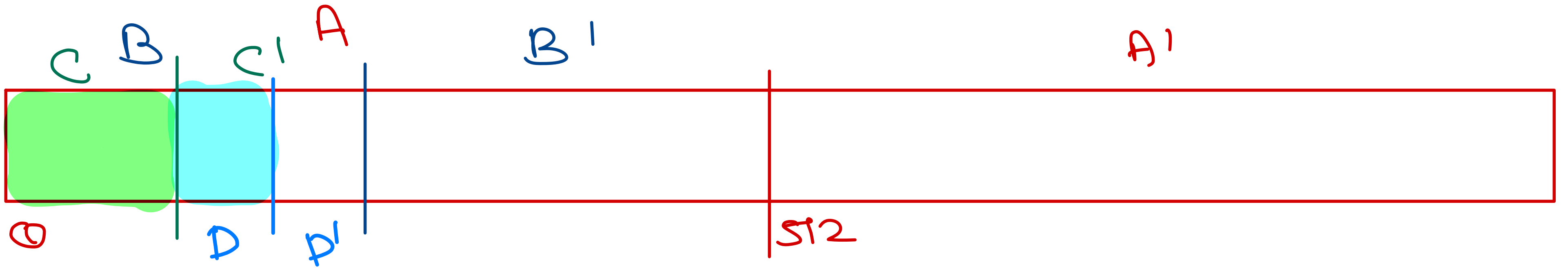
256 → B'

128 → C'

64 →

32 →

Buddy allocator



1. 128 → C
2. 64 → D

1024 →

512 → A'

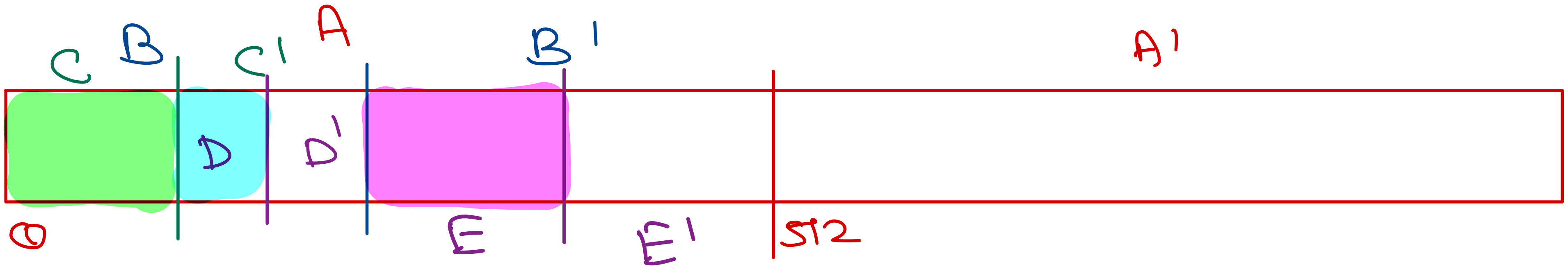
256 → B'

128 →

64 → D'

32 →

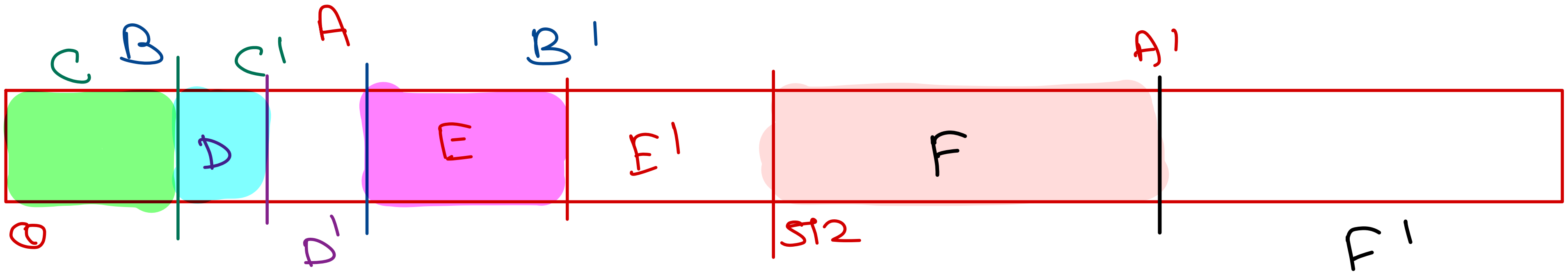
Buddy allocator



1. 128 → C
2. 64 → D
3. 128 → E

1024 →
512 → A'
256 →
128 → E'
64 → D'
32 →

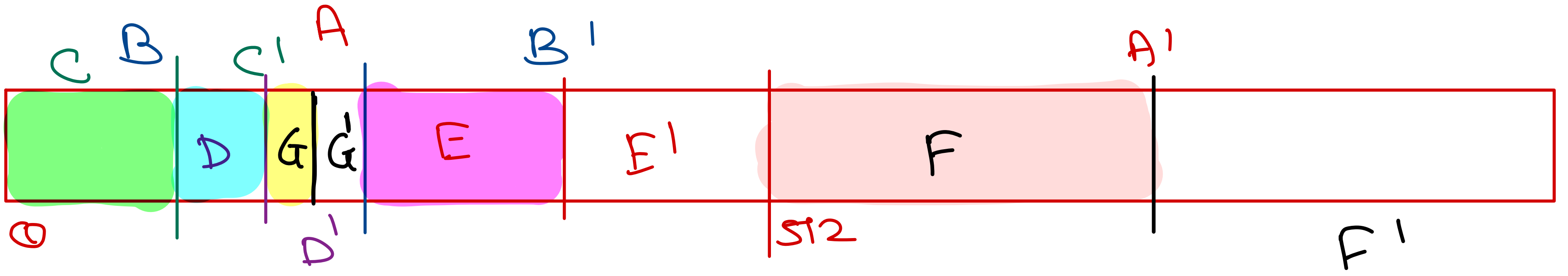
Buddy allocator



1. 128 → C
2. 64 → D
3. 128 → E
4. 256 → F

1024 →
512 →
256 → F'
128 → E'
64 → D'
32 →

Buddy allocator



1. 128 → C
2. 64 → D
3. 128 → E
4. 256 → F
5. 32 → G

1024 →

512 →

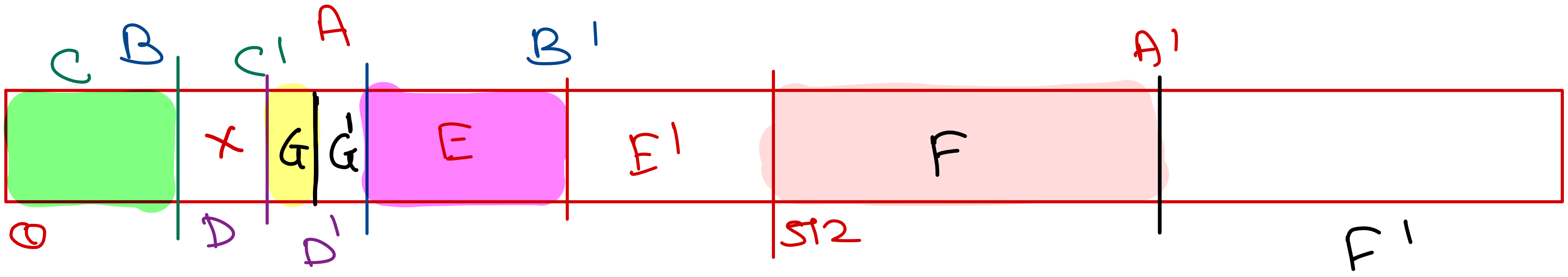
256 → F'

128 → E'

64 →

32 → G'

Buddy allocator



Allocations

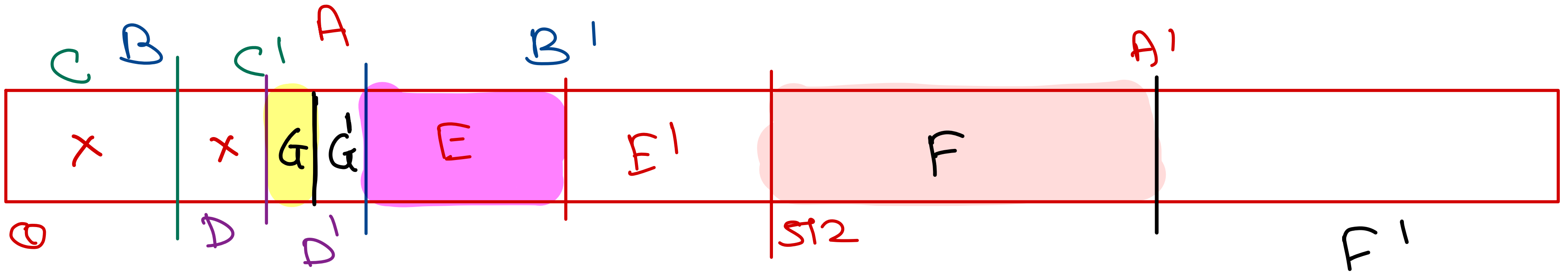
1. 128 → C
2. 64 → D
3. 128 → E
4. 256 → F
5. 32 → G

Deallocation

1. 64 - D

1024 →
512 →
256 → F'
128 → E'
64 → D
32 → G'

Buddy allocator



Allocations

1. 128 → C
2. 64 → D
3. 128 → E
4. 256 → F
5. 32 → G

Deallocation

1. 64 - D
2. 128 - C

1024 →

512 →

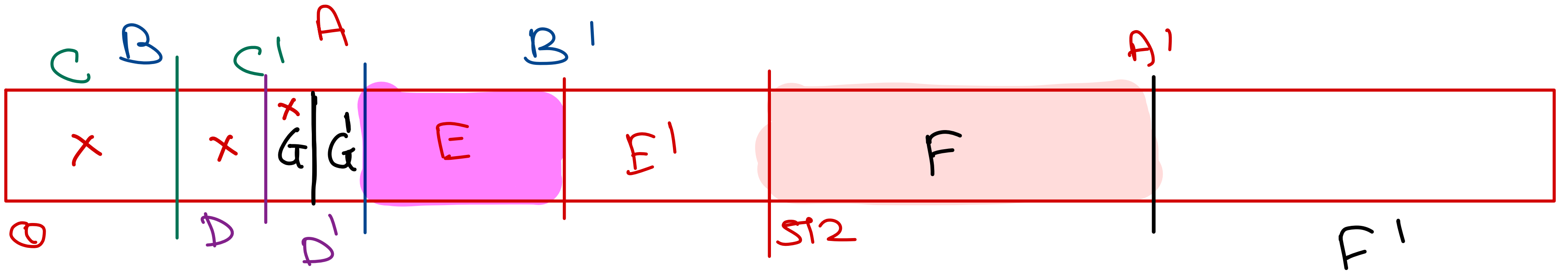
256 → F'

128 → E', C

64 → D

32 → G'

Buddy allocator



Allocations

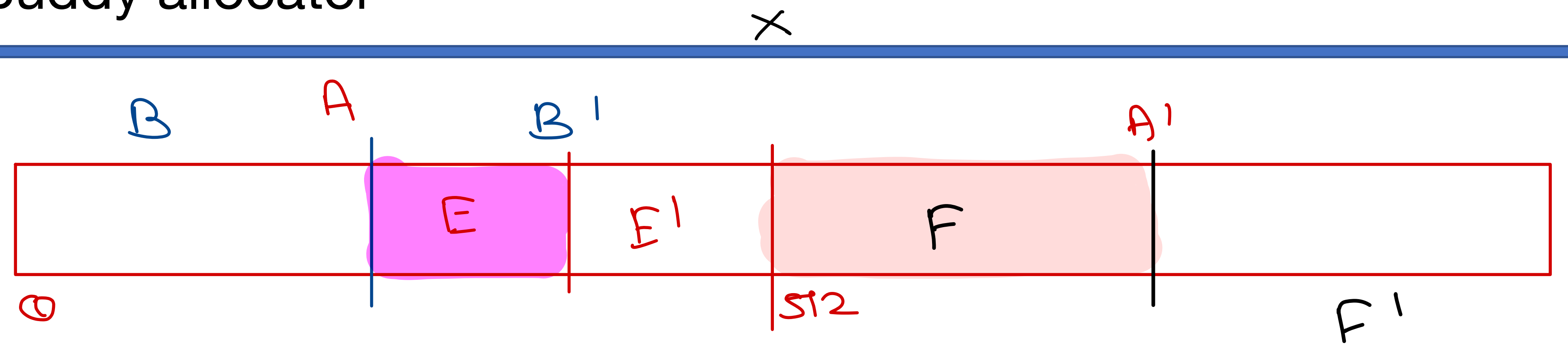
1. 128 → C
2. 64 → D
3. 128 → E
4. 256 → F
5. 32 → G

Deallocation

1. 64 - D
2. 128 - C
3. 32 - G

1024 →
512 →
256 → F', B
128 → E', C, C'
64 → D, D'
32 → G', G

Buddy allocator



Allocations

1. 128 → C
2. 64 → D
3. 128 → E
4. 256 → F
5. 32 → G
6. 32 → H

Deallocation

1. 64 - D
2. 128 - C
3. 32 - G

For this allocation, we need to again break higher order blocks, which were merged in earlier request. This is time-consuming.

1024 →
512 →
256 → F', B
128 → E'
64 →
32 →

To make more efficient, buddy alloc can be implemented with lazy coalescing. Released block will return to its free list, but will not merge with buddy. The merge op of all avail buddies will be done when higher order block is not avail (for alloc).

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>