

# Overview

- This Lecture
  - Hardware and interrupt handler
  - Source: LDD ch9 & ch10, including time and memory allocation from ch7 & ch8

# Time measurement

- jiffies
  - The number of timer interrupts so far
  - HZ is the frequency of the timer interrupt (usually 100)
  - $\text{Num-of-seconds} = \text{num-of-jiffies} / \text{HZ}$
- Functions for comparing time
  - `int time_after(unsigned long a, unsigned long b);`
  - `int time_before(unsigned long a, unsigned long b);`
- Caveat: integer wraps around!
- For 64 bit jiffies, use `u64 get_jiffies_64(void);`
- If you need high precision time, use processor specific registers, e.g. counter of CPU clock cycles

# Other time functions

- `int time_after_eq(unsigned long a, unsigned long b);`
- `int time_before_eq(unsigned long a, unsigned long b);`
- `unsigned long timespec_to_jiffies(struct timespec *value);`
- `void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);`
- `unsigned long timeval_to_jiffies(struct timeval *value);`
- `void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);`
- `void do_gettimeofday(struct timeval *tv);`  
//familiar?:-)

# Delaying execution

- Device drivers need to delay execution while waiting for device being ready
- Busy waiting (not recommended)
  - `while (time_before(jiffies, j1)) cpu_relax();`
- Yielding the CPU
  - `while (time_before(jiffies, j1)) { schedule(); }`
- Timeout
  - `wait_queue_head_t wait;`
  - `init_waitqueue_head (&wait);`
  - `wait_event_interruptible_timeout(wait, 0, delay);`

# Delaying execution (cont.)

- Use scheduler timeout

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```
- For short delays, the following functions can be used

```
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int
    millisecs);
void ssleep(unsigned int seconds)
```

# Kernel timers

- Use kernel timers to schedule a parallel thread at some future time, without blocking the current thread
  - A kernel timer is a data structure that instructs the kernel to execute a programmer-defined function with a programmer-defined argument at a programmer-defined time.
  - The programmer-defined function must be atomic
    - No user space to access
    - No scheduling or sleeping
  - Use the following functions to test if your function can sleep or not
    - `in_interrupt()`: returns nonzero if in interrupt context
    - `in_atomic()`: return nonzero if scheduling is not allowed

# Timer API

```
#include <linux/timer.h>
struct timer_list {      /* ... */
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data; };
void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function,
    _expires, _data);
void add_timer_on(struct timer_list * timer, int cpu);
int del_timer(struct timer_list * timer);
int mod_timer(struct timer_list *timer, unsigned long
    expires);
```

# Timer implementations

- Requirements
  - Timer management must be as lightweight as possible.
  - The design should scale well as the number of active timers increases.
  - Most timers expire within a few seconds or minutes at most, while timers with long delays are pretty rare.
  - A timer should run on the same CPU that registered it.
- Refer to ch7 of LDD for details.



# Tasklet

- Tasklet
  - A tasklet is like kernel timer. It is executed as a soft interrupt and on the same CPU on which it is scheduled
  - Used for non-crucial operations for interrupt handling
- Related data structure and macros
  - `#include <linux/interrupt.h>`
  - `struct tasklet_struct { /* ... */`
  - `void (*func)(unsigned long);`
  - `unsigned long data; };`
  - `void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);`
  - `DECLARE_TASKLET(name, func, data);`

# Tasklet features

- Tasklet has the following features
  - Disabled and re-enabled later;
  - Like timers, a tasklet can reregister itself.
  - Normal priority or high priority.
  - Tasklets may be run immediately if the system is not under heavy load but never later than the next timer tick (?).
  - A tasklet can be concurrent with other tasklets but is strictly serialized with respect to itself, i.e. the same tasklet never runs simultaneously on more than one processor. Also, a tasklet always runs on the same CPU that schedules it.

# Tasklet functions

- `void tasklet_disable(struct tasklet_struct *t);`
- `void tasklet_disable_nosync(struct tasklet_struct *t);`
- `void tasklet_enable(struct tasklet_struct *t);`
- `void tasklet_schedule(struct tasklet_struct *t);`
- `void tasklet_hi_schedule(struct tasklet_struct *t);`
- `void tasklet_kill(struct tasklet_struct *t);`

# Work queues

- Similar to tasklets, but
  - Run in the context of special kernel process, and thus can sleep.
  - Can be delayed at a specified time
- Shared workqueue
  - The default queue provided by the system
  - Cannot sleep for a long time; otherwise it may affect other queue users.

# Memory allocation

- The most convenient function

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```
- Flags are worth paying attention
  - GFP\_ATOMIC: no sleep
  - GFP\_KERNEL: normal, may sleep
  - GFP\_USER: memory for user space, may sleep
  - ...
- Memory zones: normal, DMA, high-mem
  - Flags: \_\_GFP\_DMA, \_\_GFP\_HIGHMEM
- Smallest allocated size: 32 or 64 bytes

# Lookaside caches

- A facility to create a pool of memory blocks with the same size
- This helps with optimal use of memory when the same data structure is used repeatedly in the kernel, e.g. `skb` for networking
- Check the details at LDD ch8. They are very useful when a serious device driver is written
- You may reserve a large block of memory for use of critical situations
  - Memory pool is another option

# Handling free pages

- `get_zeroed_page(unsigned int flags);`
- `__get_free_page(unsigned int flags);`
- `__get_free_pages(unsigned int flags, unsigned int order);`
- `void free_page(unsigned long addr);`
- `void free_pages(unsigned long addr, unsigned long order);`
- `struct page alloc_page(unsigned int flags);`
- Previous functions return physically contiguous pages
- Occasionally you may use **`vmalloc`**, which may return non-continuous pages, but is not recommended for device drivers

# Per-CPU variables

- A feature in 2.6 kernel
- A variable is defined once, but has a copy for each CPU
- Important for SMP or CMT machines
- When the variable is modified, only the local copy is modified.
- There is no need of locking when modifying the variable (high performance)
- The total value of the variable is the sum of all local copies of the CPUs, of course:)



# I/O ports and I/O memory

- Devices have registers which are mapped to I/O address space (I/O ports) or the memory address space (I/O memory)
- The main difference between I/O memory and conventional memory
  - Side effect on I/O memory
  - Can't get optimized as RAM, such as cache and reordering (no cache used for I/O memory)
- To counter against compiler optimization, memory barriers are used
  - `void barrier(void); void rmb(void); void wmb(void); void mb(void);`

# I/O ports

- I/O port allocation
  - `#include <linux/ioport.h>`
  - `struct resource *request_region(unsigned long first, unsigned long n, const char *name);`
  - `void release_region(unsigned long start, unsigned long n);`
  - `int check_region(unsigned long first, unsigned long n);`
- Manipulating I/O ports
  - `unsigned inb(unsigned port);`
  - `void outb(unsigned char byte, unsigned port);`

# I/O ports (cont.)

- Manipulating I/O ports
  - unsigned inw(unsigned port);
  - void outw(unsigned short word, unsigned port);
  - unsigned inl(unsigned port);
  - void outl(unsigned longword, unsigned port);
- I/O port access from user space
  - Use ioperm or iopl to get permission
- String operations
  - Can read/write repeatedly
  - E.g. void insb(unsigned port, void \*addr, unsigned long count);
- You may need to pause when doing I/O

# Parallel port

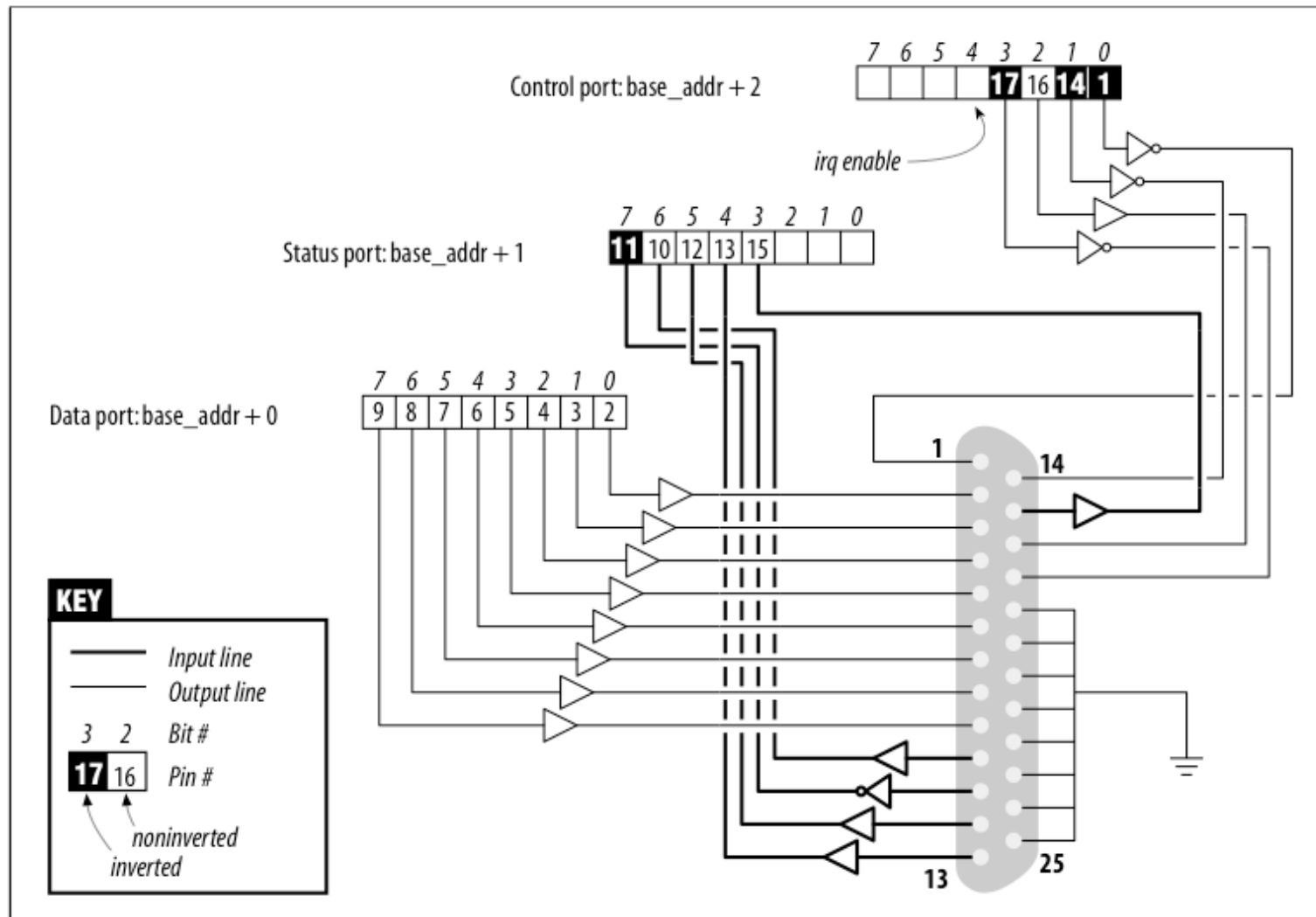


Figure 9-1. The pinout of the parallel port

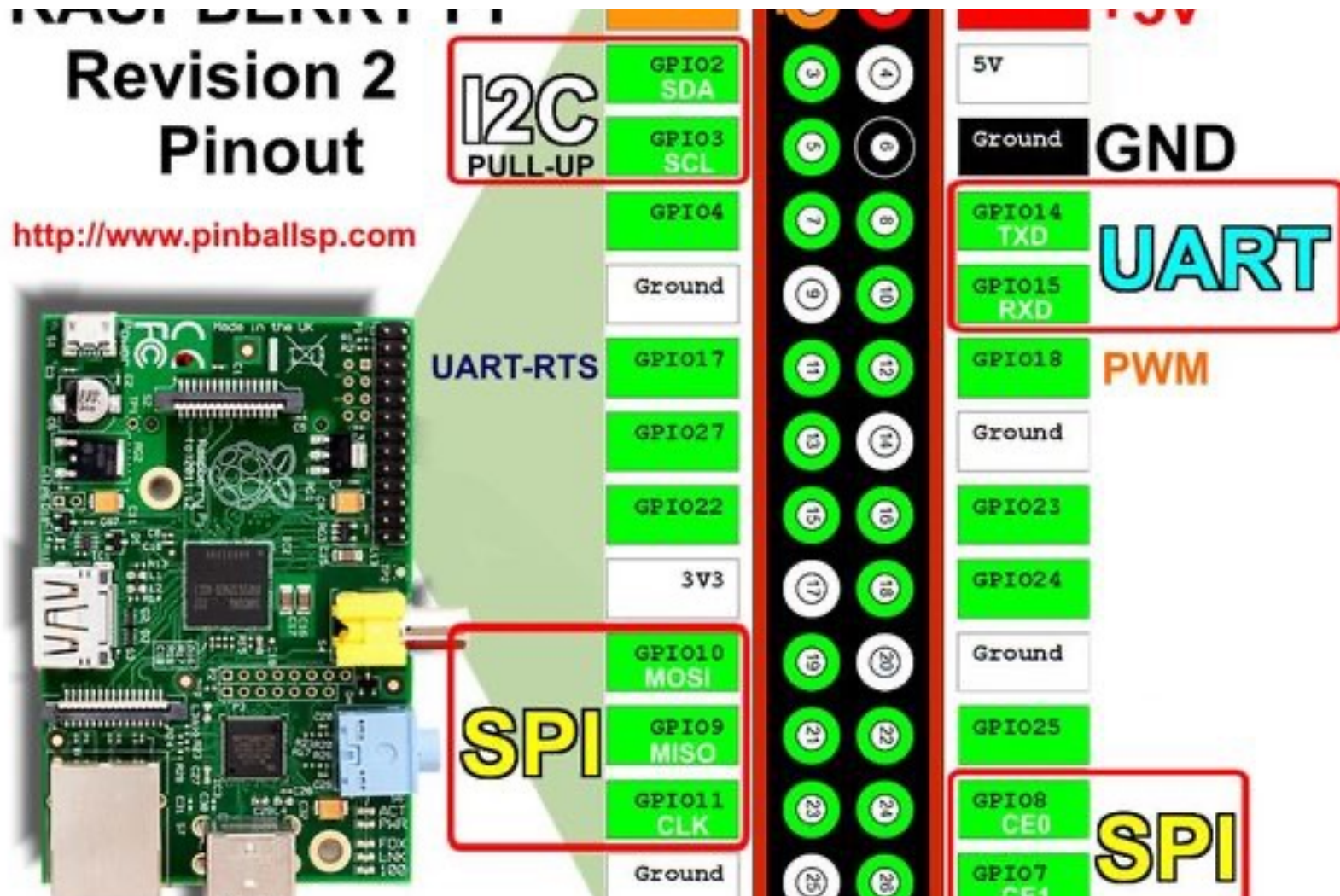
# Parallel port

- Base address
  - 0x378 for the first parallel port, 0x278 for the second
- Port 0
  - Bidirectional data register
- Port 1
  - Read-only status register
  - Bit 7 (0x80) can be used by the device to trigger interrupt when its value is changed from 0 to 1
- Port 2
  - Write-only control register
  - Bit 4 (0x10) is used to enable interrupt

# GPIO

- A programmable circuit
- Behavior can be defined by user
  - E.g. input or output, enable/disable, high/low
- Use by System-on-Chip, embedded applications for reading various sensors
- Our second assignment uses 10 GPIO pins
  - Five pairs, four pairs used for data transmission and one pair used for IRQ
  - You should read half byte at every interrupt

# RPI GPIO pins



# Using I/O memory

- I/O memory allocation (physical)
  - `struct resource *request_mem_region (unsigned long start, unsigned long len, char *name);`
  - `void release_mem_region(unsigned long start, unsigned long len);`
  - `int check_mem_region(unsigned long start, unsigned long len);`
- Mapping to virtual address (linear address)
  - `#include <asm/io.h>`
  - `void *ioremap(unsigned long phys_addr, unsigned long size);`
  - `void iounmap(void * addr);`



# Using I/O memory (cont.)

- Accessing I/O memory
  - `unsigned int ioread8(void *addr);`
  - `unsigned int ioread16(void *addr);`
  - `unsigned int ioread32(void *addr);`
  - `void iowrite8(u8 value, void *addr);`
  - `void iowrite16(u16 value, void *addr);`
  - `void iowrite32(u32 value, void *addr);`
  - `void ioread8_rep(void *addr, void *buf, unsigned long count);`
  - `void ioread16_rep(void *addr, void *buf, unsigned long count);`
  - ...

# Interrupt handling

- Install an interrupt handler
  - `int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);`
  - `void free_irq(unsigned int irq, void *dev_id);`
- Flags
  - `SA_INTERRUPT, SA_SHIRQ, SA_SAMPLE_RANDOM`
- Example
  - `result = request_irq(short_irq, short_interrupt, SA_INTERRUPT, "short", NULL);`

# Autodetecting IRQ number

- Based on knowledge of devices
- Kernel assisted probing
  - unsigned long probe\_irq\_on(void);
  - int probe\_irq\_off(unsigned long);
- Do it yourself (DIY)
  - Install a handler for all possible irq numbers and check which one respond when interrupt occurs
- Dynamically config
- Interesting proc directories
  - /proc/interrupts; /proc/iomem; /proc/stat;

# Write a handler

- Remember a handler is in the context of hard interrupt
- Args for interrupt handler
  - `irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)`
  - IRQ number
  - Device data (for shared IRQ)
  - Saved register values when interrupt occurs (may not need them, architecture dependent)
- Return value
  - `IRQ_HANDLED` or `IRQ_NONE`

# Disable interrupts

- Functions are provided to enable and disable a particular interrupt
  - `void disable_irq(int irq);`
  - `void disable_irq_nosync(int irq);`
  - `void enable_irq(int irq);`
- But when do we need to disable interrupts?

## 2<sup>nd</sup> programming assignment

- Install an interrupt handler for the dummy port device using GPIO pins
- The handler should assemble two half-bytes from the port into one byte and write to a circular buffer and then wake up a sleeping tasklet.
- The tasklet should copy the data from the circular buffer to an infinite buffer
- A reader continuously fetches the data from the infinite buffer; if the buffer is empty, it sleeps.
- The current IRQ is blocked in the handler.

# Microkernel

- Minimize the kernel
  - Implement outside the kernel whatever possible
- Pros
  - More modular system structure
  - Servers run at user level so malfunction is isolated as other user processes
  - Flexible and tailorable
    - Different policies and APIs can be implemented by different servers that coexist.
- Cons: lack of efficiency due to frequent IPC and context switches

# L4 ideology

- Principle
  - A concept is tolerated inside the kernel only if moving it outside the kernel would prevent the implementation of the system's required functionality
- Assumptions
  - Support interactive and not completely trustworthy applications
  - Page-based virtual memory



# L4 ideology (cont.)

- Requirements
  - Principle of independence: a programmer must be able to implement an arbitrary subsystem  $S$  in such a way that it cannot be disturbed or corrupted by other subsystem  $S'$ 
    - $S$  can give guarantees independent of  $S'$
  - Principle of integrity: other subsystems must be able to rely on the guarantees of a subsystem
    - There must be a way for  $S1$  to address  $S2$  and to establish a communication channel which can neither be corrupted nor eavesdropped by  $S'$

# Example

- Use a key server as an example
- A key server can only be realized with mechanisms that
  - Protect its code and data
  - Ensure nobody else reads or modifies the key
  - Enable the demander to check whether the key comes from the key server
    - Finding the key server can be done by a name server and public key based authentication

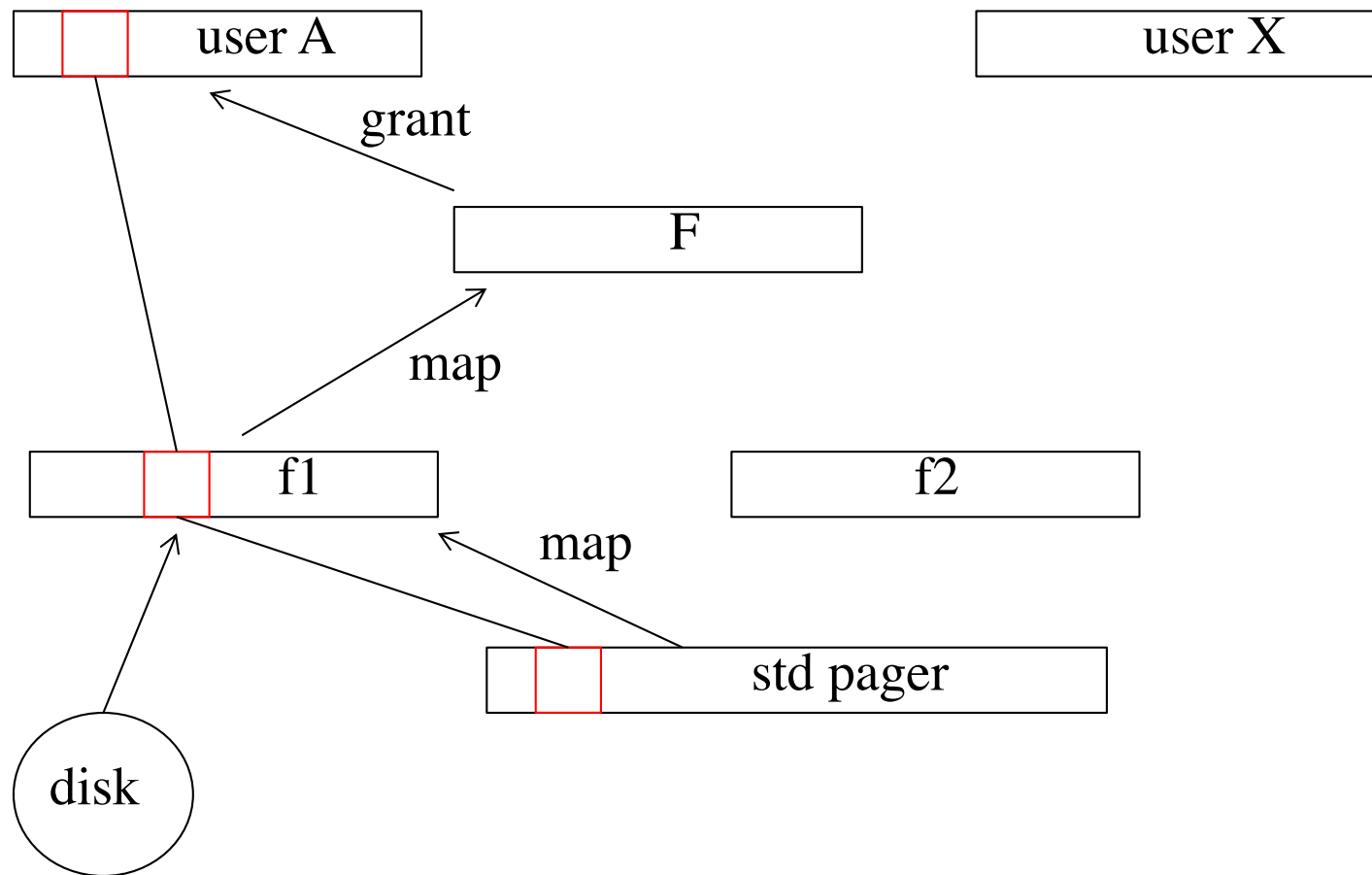
# Address space

- A mapping between virtual pages with physical pages
- Support recursive construction of address spaces outside the kernel
- By magic, there is the initial address space  $\sigma_0$  which maps all physical pages and is controlled by the first subsystem  $S_0$
- Initially all other address spaces are empty

# Address space (cont.)

- For constructing and maintaining further address spaces on top of  $\sigma_0$ , L4 provides the following three operations:
  - Grant
    - Grant a page to another space and the granted page is removed from the granter's space
  - Map
    - Similar to grant but the mapper keeps the page in its space
  - Flush
    - Remove a page from other spaces that receives the page from the flusher

# Address space (cont.)



# Threads

- A thread is an activity executing inside an address space
- It is characterized by a set of registers
  - instruction pointer, stack pointer and state info
  - Address space it is executing on
- Threads are included in the kernel due to its tight association with address spaces
- All changes to thread's address space must be controlled by the kernel

# IPC

- Cross-address-space communication (IPC) must be supported by the kernel
  - The classical method is transferring messages between threads by the kernel
- IPC protocol
  - The sender decides to send information and determines its content
  - The receiver determines if it is willing to receive information and is free to interpret the received message

# IPC (cont.)

- IPC is not only the basic concept for comm. between subsystems, but also, with address spaces, the foundation of independence.
- Interrupts
  - The natural abstraction of hardware interrupts is the IPC message
  - The hardware is regarded as a set of threads which have special IDs and send empty messages to corresponding software threads
- Unique Identifiers



# User space servers

- Memory manager
- Pager
- Multimedia resource allocation
- Device driver
- Second level cache and TLB
- Remote communication
- Unix server

# Performance

- Switching overhead
- IPC overhead
- Non-portability