

Programming assignments

- Five must submit programming assignments (20 Marks)
 - We expect all of you to do these assignments
 - Questions from assignments in the midsem and endsem
 - do them to perform well in exams
- Four optional assignments (16 Points)
 - Optional assignments have prerequisites
 - Demos for optional assignments will be conducted after the endsem
 - You can do them to boost your marks or improve your learning, whichever suits you better
- We will consider the $\min(20, \text{must} + \text{optional} + 2)$ marks in the final grading

Homework

- $\min(10, \text{marks obtained in homework})$ will be considered for final grading
- Questions form homework in midsem and endsem
 - do them perform well in exams

How to learn OS

- Do the programming assignments and homework
- If you are struggling with C
 - take help from the TAs and your friends who know C
 - If you think you need a full semester course for C language, you can talk to HoD or other administrative people
- Good knowledge of the C language is a prerequisite for the OS course

Homework-2

```
int foo () {
    int a, b, c, d, e, f;
    a = 0, b = 1, c = 2, d = 3;
    e = 5, f = 6;
    bar(b, c, d, f);
    return a + b + c;
}
foo: a(%eax), b(%ebx), c(%ecx),
d(%edx), e(%esi), f(%edi)
```

```
movl 8(%esp), %edx
movl 12(%esp), %edi
call bar
addl $16, %esp
popl %ecx
popl %eax
addl %ebx, %eax
addl %ecx, %eax
popl %edi
popl %esi
popl %ebx
ret
```

```
ebx, esi, edi,
ebp

pushl %ebx
pushl %esi
pushl %edi
movl $0, %eax
movl $1, %ebx
movl $2, %ecx
movl $3, %edx
movl $5, %esi
movl $6, %edi
pushl %eax
pushl %ecx
addl $-16, %esp
movl %ebx, (%esp)
movl %ecx, 4(%esp)
```

Homework-2

```
int bar(int a, int b, int c, int d) {  
    int x, y, z, w;  
    x = a, y = b, z = c, w = d;  
    return x + y + z + w;  
}
```

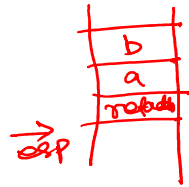
```
bar: x(%eax), y(%ebx), z(%ecx),  
     w(%edx)
```



```
push .l.ebx  
mov 8(.l.esp), .l.eax  
mov 12(.l.esp), .l.ebx  
mov 16(.l.esp), .l.ecx  
mov 20(.l.esp), .l.edx  
add .l.ebx, .l.eax  
add .l.ecx, .l.eax  
add .l.edx, .l.eax  
pop .l.ebx  
ret
```

Homework-2

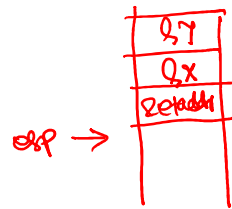
```
void foo (int a, int b) {  
    bar(&a, &b);  
}
```



```
lea 4(-1.esp), .1.eax  
lea 8(-1.esp), .1.ecx  
push .1.ecx  
push .1.eax  
call bar  
add $8, .1.esp  
ret
```

Homework-2

```
int bar(int *x, int *y) {  
    *x = 0; *y = 1;  
}
```



```
mov 4(1.esp), 1.eax  
mov 8(1.esp), 1.ecx  
mov %0, (1.eax)  
mov %1, (1.ecx)  
ret
```


Homework-2

```
int bar(int x, int y);  
int foo (int a, int b) {  
    return bar(a, b);  
}
```



$x + 0x21$

00000000 <foo>:

0:	55	push	%ebp
1:	89 e5	mov	%esp,%ebp
3:	53	push	%ebx
4:	83 ec 04	sub	\$0x4,%esp
7:	e8 fc ff ff ff	call	8 <foo+0x8>
c:	05 01 00 00 00	add	\$0x1,%eax
11:	83 ec 08	sub	\$0x8,%esp
14:	ff 75 0c	pushl	0xc(%ebp)
17:	ff 75 08	pushl	0x8(%ebp)
1a:	89 c3	mov	%eax,%ebx
1c:	e8 fc ff ff ff	call	1d <foo+0x1d> # bar
21:	83 c4 10	add	\$0x10,%esp
24:	8b 5d fc	mov	-0x4(%ebp),%ebx
27:	c9	leave	
28:	c3	ret	

Homework-2

```
int x;  
void foo() {  
    x = 0;  
}
```

4 
foo:
5  call get_eip
 mov \$0, -2(1.eax);
 ret

get_eip
 mov (1.esp), 1.eax
 ret

Homework-2

`foo(1, 4)`

`foo(1, 0, 6)`

`foo(2, 5, 9, 0)`

Homework-3

```
int schedule_disabled = 0; // global
struct list *ready_list; // global
struct thread *cur_thread; // global
void schedule1 () {
    if (schedule_disabled) {
        return;
    }
    schedule_disabled = 1;
    push_list(ready_list, cur_thread);
    schedule();
    schedule_disabled = 0;
}
```

Homework-3

- Emulation of interrupt disable

```
int interrupt_disabled = 0;
```

In every interrupt handler:

```
cmp $0, interrupt_disabled  
je 1f  
iret  
1: // original interrupt code
```

Homework-3

bar:

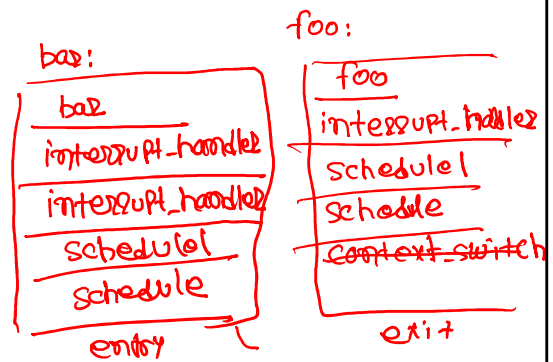
1. push %ebp
2. mov %esp, %ebp
3. mov \$100, %eax
4. mov %ebp, %esp
5. pop %ebp
6. ret

foo:

1. push %ebp
2. mov %esp, %ebp
3. mov \$101, %eax
4. mov %ebp, %esp
5. pop %ebp
6. ret

interrupt_handler:

1. push %eax
2. push %edx
3. push %ecx
4. call schedule1
5. pop %ecx
6. pop %edx
7. pop %eax
8. iret



foo was preempted before.
bar is the current thread.

context_switch(struct thread *prev, struct thread *next)

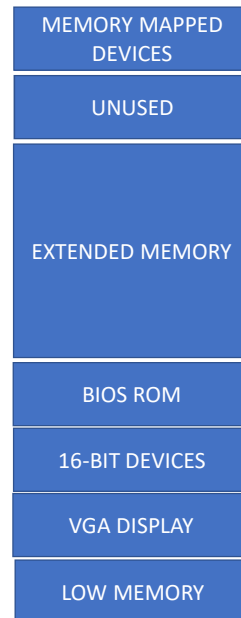
```
push %ebp
push %esi
push %edi
push %ebx
mov 20(%esp), %eax
mov %esp, (%eax)      // prev->esp = %esp
mov 24(%esp), %eax
mov (%eax), %esp      // %esp = next->esp
pop %ebx
pop %edi
pop %esi
pop %ebp
ret
```

```
struct thread {
    void *esp;
};
```

How to say hello to a device?

Port I/O

- Use dedicated I/O space
 - Only 1024 I/O address
 - also called ports
 - accessed using in/out instructions



Apart from the physical address space, the hardware has a dedicated I/O space. The size of I/O space is very small (only 1024 addresses). These addresses are called ports. These ports are connected to devices. Software can talk to devices by reading/writing from/to these ports. Reading and writing to these ports are done through special instructions “in” and “out”. “in” instructions takes the address of the port as an argument and reads a (1,2,4) byte value from the port in the EAX register. Similarly, out instruction writes the value of the EAX register to a given port.

Port I/O

- Each device comes with a manual
 - The manual talks about how to communicate with the device
 - e.g., a printer device manual says, “if you want to say hello, please write 1 to port 100.”
 - `outl $100` instruction writes the content of the `eax` register to port 100

```
mov $1, %eax  
outl $100
```

To write the four-byte one to port 100, the x86 CPU needs to execute “`mov $1, %eax`” and “`outl $100`” in that order.

Port I/O

- Each device comes with a manual
 - On the next page, the manual says, “if you want to check if I have received your message, please read my status from **port 101**. I will set it to 1 after receiving your message.”
 - **inl \$101** instruction reads a 4-byte value from **port 101** in the register **eax**

```
inl $101
cmp $1, %eax
je if
je 1:
```

The instruction “inl \$101” reads a 4-byte value from port 101 into register %eax.

Line printer (Port I/O)

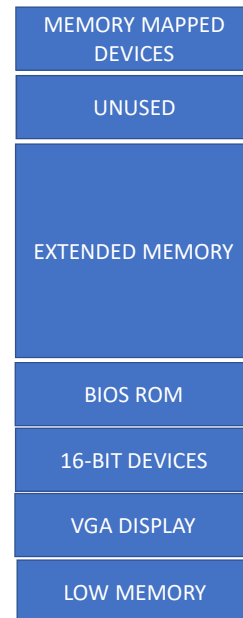
```
#define DATA_PORT 0x378
#define STATUS_PORT 0x379
#define BUSY 0x80
#define CONTROL_PORT 0x37A
#define STROBE 0x01

void lpt_putc(int c) {
    while((inb(STATUS_PORT) & BUSY) == 0) ; /* wait for printer to consume previous byte */
    outb(DATA_PORT, 'c'); /* put the byte on the parallel lines */
    outb(CONTROL_PORT, STROBE); /* tell the printer to look at the data */
    outb(CONTROL_PORT, 0);
}
```

inb routine reads a one-byte value from an input port (using in instruction). outb routine writes a one-byte value to a given port (using out instruction). This particular device requires OS to write 1 followed by 0 to the CONTROL_PORT after putting the data on the DATA_PORT.

Memory mapped I/O

- Some addresses in the physical address space are reserved for devices
- Software talks to the devices by reading/writing to these addresses



Apart from the port I/O, the devices can talk to CPU through memory-mapped I/O. Some part of the physical address space is reserved for devices (marked as MEMORY MAPPED DEVICES in this diagram). Software can talk to devices by reading/writing to these addresses.

Memory mapped I/O

- Each device comes with a manual
 - The manual talks how to communicate with the device
 - e.g., a printer device manual says, “if you want to say hello, please write 1 to address **0x100100**.”

```
int *ptr = (int *) 0x100100;  
ptr[0] = 1;
```

To access memory-mapped devices, we don't need special instructions. We can do it in the C language itself.

Memory mapped I/O

- Each device comes with a manual
 - On the next page, the manual says, “if you want to check if I have received your message, please read my status from address **0x100108**. I will set it to 1 after receiving your message.”

```
int *ptr = (int *) 0x100108;  
while (ptr[0] != 1);
```

Line printer (MMIO)

```
volatile int *DATA_REG = 0xfffff00;  
volatile int *STATUS_REG = 0xfffff04;  
volatile int *CONTROL_REG = 0xfffff0C;  
#define BUSY 0x80  
#define STROBE 0x01  
  
void lpt_putc(int c) {  
    while ((STATUS_REG[0] & BUSY) == 0) ; /* wait for printer to consume previous byte */  
    DATA_REG[0] = 'c'; /* put the byte on the parallel lines */  
    CONTROL_REG[0] = STROBE; /* tell the printer to look at the data */  
    CONTROL_REG[0] = 0;  
}
```

This is the same device code that we discussed before, but this time it is using MMIO.

Compiler optimizations

- We know that the compiler can use registers to save temporary computations
- This results into further optimizations
 - e.g., `int *STATUS_REG = 0xfffff04; while ((STATUS_REG[0] & BUSY) == 0);`
 - can be rewritten to
 - `%eax = STATUS[0] & BUSY; while (%eax == 0);`
 - eliminating the load in a busy loop
 - which results into an infinite loop

Compiler optimizations

```
int *STATUS_REG = 0xffffffff04;  
while ((STATUS_REG[0] & BUSY) == 0);
```

```
%eax = STATUS[0] & BUSY;  
while (%eax == 0);      // infinite loop
```

The compiler can save temporary computations in registers to save loads/stores. This may result into an infinite loop.

What went wrong?

- This program works perfectly if compiler always loads the value in the loop
- The compiler does not know that a device can change the value in the memory location
- How to prevent the compiler from doing this optimization

Optimization barrier

- **asm volatile ("":: "memory")** prevents the compiler from performing this optimization.
 - Read A.3.5 Optimization Barriers
 - https://faculty.iiitd.ac.in/~piyus/pintos/doc/pintos_6.html#SEC98

All the values, which are cached in registers, are reloaded across the optimization barrier.

Correct code

```
while ((STATUS_REG[0] & BUSY) == 0) {  
    asm volatile("" ::: "memory");  
}
```

volatile

- the **volatile** keyword can also be used to prevent the compiler from caching the value of a variable in a register

```
volatile int *STATUS_REG = 0xffffffff04;  
while ((STATUS_REG[0] & BUSY) == 0) ;
```

Here, **volatile** makes sure that the value of the memory location will be loaded inside the loop.

acquire

```
acquire(struct lock *l) {  
    status = interrupt_disable();  
    while (l->value == 0) {  
        list_push(l->wait_list, cur_thread);  
        schedule();  
    }  
    l->value = 0;  
    set_interrupt_status(status);  
}
```

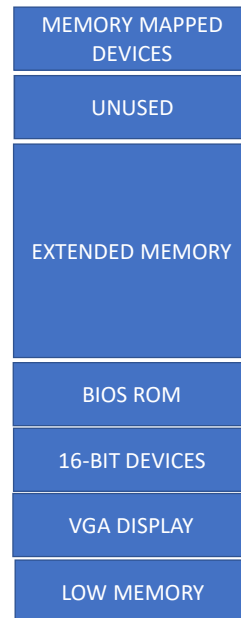
```
struct lock {  
    int value;  
};
```

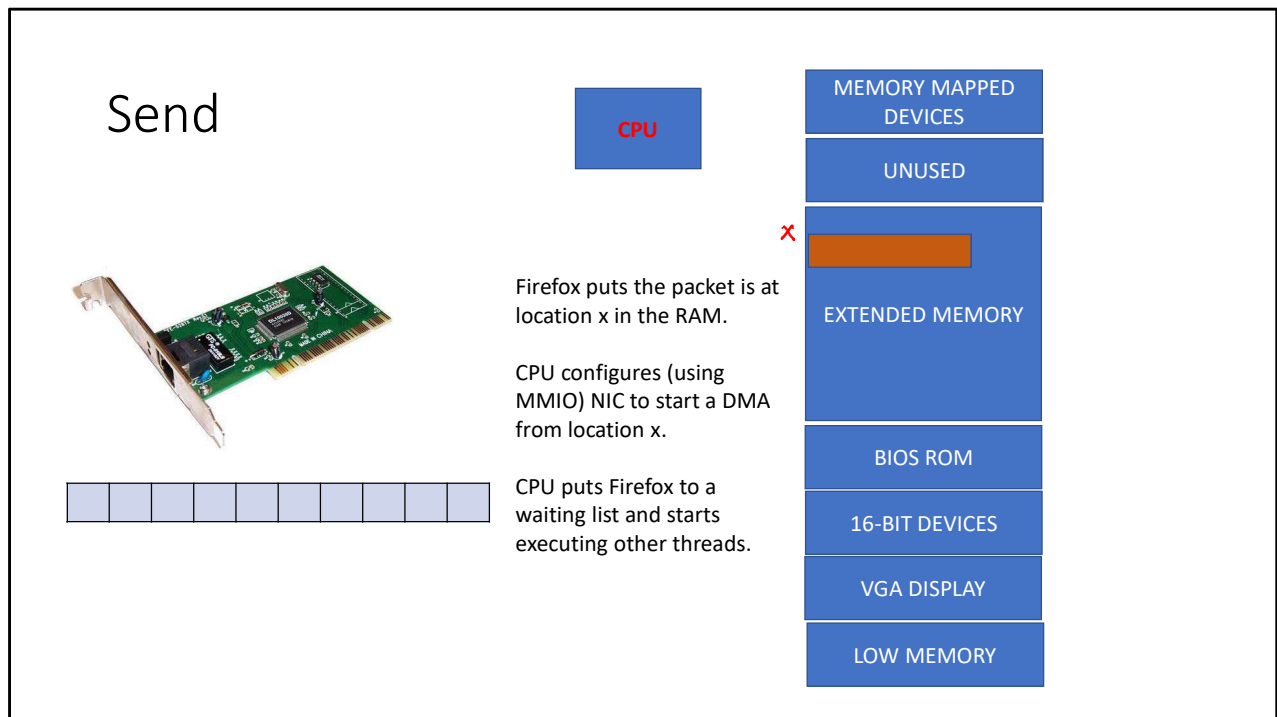
Is it possible that the compiler can transform this while loop into an infinite loop by loading the "l->value" in a register outside the loop?

No, because the compiler doesn't know for sure that "l->value" can't be modified in the "list_push" or "schedule" routines.

Direct memory access (DMA)

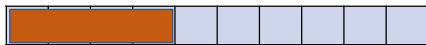
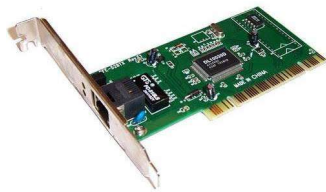
- Similar to CPU(s), devices can also directly access the RAM
 - e.g., a network device can directly copy a packet from RAM to device memory and vice versa
- the device manual says how to configure the device to write an incoming packet to RAM or read a packet that needs to be sent from RAM
 - the device needs to know the RAM addresses





NIC: network interface card. NIC has its internal memory. During send, the packet needs to be copied from RAM to NIC's internal memory.

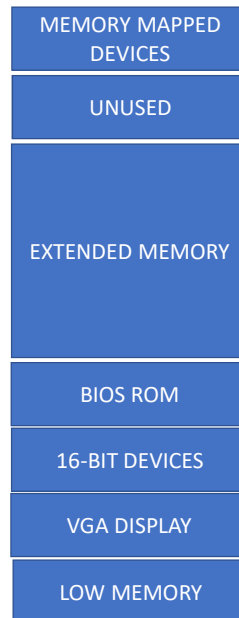
Send



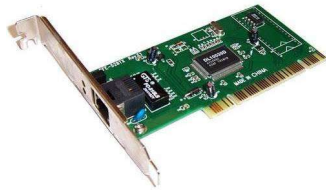
CPU is executing other threads.

NIC is copying packet from memory location x to its internal buffer.

Both things are happening in parallel.



Interrupt mode



CPU

NIC has copied the packet to its internal buffer. NIC sends interrupt to CPU.



MEMORY MAPPED
DEVICES

UNUSED

EXTENDED MEMORY

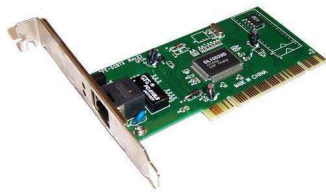
BIOS ROM

16-BIT DEVICES

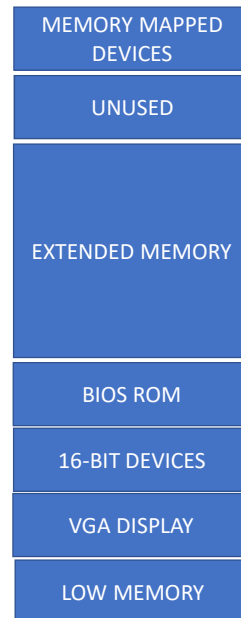
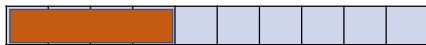
VGA DISPLAY

LOW MEMORY

Interrupt mode



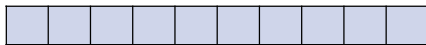
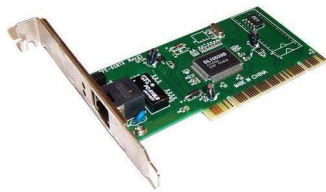
The interrupt handler is called. CPU sets the packet's status to sent, puts Firefox to the ready list, and goes back to user mode.



Polling mode

- In polling mode, CPU performs MMIO to check if the packet has been copied to the device memory

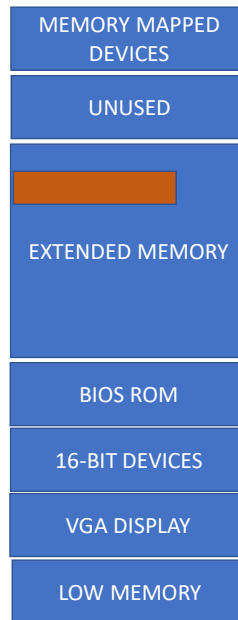
Send



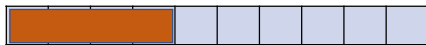
Firefox puts the packet is at location x in the RAM.

CPU configures (using MMIO) NIC to start a DMA from location x.

CPU puts Firefox to a waiting list and starts executing other threads.



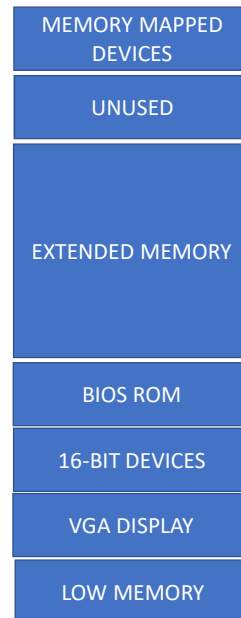
Send



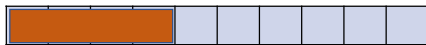
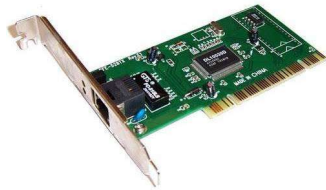
CPU is executing other threads.

NIC is copying packet from memory location x to its internal buffer.

Both things are happening in parallel.

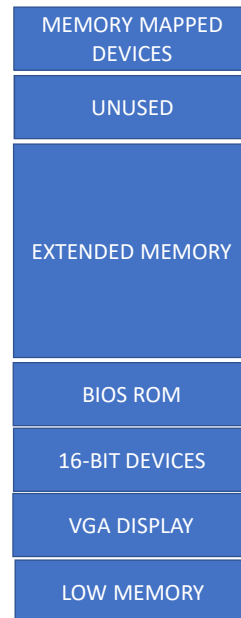


Polling

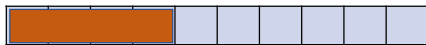


CPU

NIC has copied the packet to its internal buffer. NIC writes something in the memory-mapped area to let the CPU know that copying has been done.



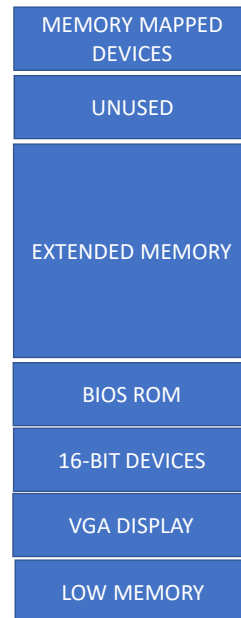
Polling



CPU

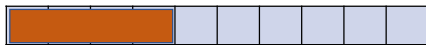
NIC has copied the packet to its internal buffer. NIC writes something in the memory-mapped area to let the CPU know that copying has been done.

CPU doesn't know that the packet has been copied. CPU still executing other threads.



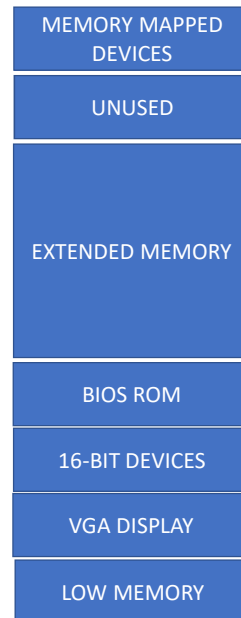
Polling

TIMER
INTERRUPT



The timer interrupt handler is called. CPU reads from memory-mapped area to check if the packet has been copied.

If the answer is yes, CPU sets the packet's status to sent and puts the Firefox to the ready queue.



Polling

- Periodically checking for the occurrence of an event
 - e.g., after every timer interrupt CPU checks the status of devices
- Not good if events are rare
 - e.g., keyboard, slow NIC
 - Unnecessary wasting CPU in polling
 - In these cases, interrupts are useful

Interrupts

- Automatically invoked after a packet is sent/received
- If the events are very frequent, the CPU spends most of its time in interrupt handling
 - e.g., very fast NIC
 - Other applications (not using NIC) will be unnecessarily interrupted a lot
 - In this case, polling is better

Example

- Firefox invokes `receive(buf)` system call to receive a packet in the `buf`
 - `receive(buf)` waits until a packet is received in `buf`

receive

```
#define RX_STATUS_PORT 0x402
#define RX_ADDR_PORT   0x403
struct list *wait_list;

void receive (char *buf) {
    /* configure device to copy packet
     * to buf when it is received
     */
    outl (RX_ADDR_PORT, buf); ✓
    outl (RX_STATUS_PORT, 0); ✓
    thread_block(wait_list); // put the current thread to wait_list and schedule a new thread
}
```

Here the device driver (OS code which communicates with the device) is communicating with the network device to tell the RAM address where an incoming packet needs to be written. The copying may take some time. The driver puts the current thread to a waitlist and schedules a new thread.

Interrupt

- `rx_intr` is invoked after the device has copied an incoming packet to `buf`

```
void rx_intr () {  
    thread_unblock(wait_list); // remove a thread from wait_list  
                                // and put it to the ready list  
}
```

The interrupt handler is called when the packet has been written to the buf. It's time to move the waiting thread to the ready list.

Polling

- `rx_poll` is invoked after every timer interrupt

```
#define RX_STATUS_PORT    0x402
void rx_poll() {
    if (inl(RX_STATUS_PORT) == 1) // if the device has copied a packet to buf
    {
        thread_unblock(wait_list); // remove a thread from wait_list
                                   // and put it to the ready list
    }
}
```

In `rx_poll`, CPU checks if a packet has been copied to buf (using port I/O in this example). If yes, then it puts the waiting thread to the ready list.