Assignment 8

12011702 张镇涛

1

(1)

Polling

For Polling, we can find these related descriptions in book:

Let us now look at a canonical device (not a real one), and use this device to drive our understanding of some of the machinery required to make device interaction efficient. From Figure 36.3 (page 4), we can see

This basic protocol has the positive aspect of being simple and working. However, there are some inefficiencies and inconveniences involved. The first problem you might notice in the protocol is that polling seems inefficient; specifically, it wastes a great deal of CPU time just waiting for the (potentially slow) device to complete its activity, instead of switching to another ready process and thus better utilizing the CPU.

Therefore, we can conclude that for polling:

Pros:

1. Being simple and working
2. Make interaction of device efficiently

Cons:

1. Polling is inefficient
2. Wastes too much CPU time, just for waiting for the device to complete activity.

Interrupt-based I/O

We can find these related paragraphs in book:

Interrupts thus allow for **overlap** of computation and I/O, which is key for improved utilization. This timeline shows the problem:

Note that using interrupts is not *always* the best solution. For example, imagine a device that performs its tasks very quickly: the first poll usually finds the device to be done with task. Using an interrupt in this case will actually *slow down* the system: switching to another process, handling the interrupt, and switching back to the issuing process is expensive. Thus, if a device is fast, it may be best to poll; if it is slow, interrupts, which allow

overlap, are best. If the speed of the device is not known, or sometimes fast and sometimes slow, it may be best to use a **hybrid** that polls for a little while and then, if the device is not yet finished, uses interrupts. This **two-phased** approach may achieve the best of both worlds.

Another reason not to use interrupts arises in networks [MR96]. When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to **livelock**, that is, find itself only processing interrupts and never allowing a user-level process to run and actually service the requests. For example, imagine a web server that experiences a load burst because it became the top-ranked entry on hacker news [H18]. In this case, it is better to occasionally use polling to better control what is happening in the system and allow the web server to service some requests before going back to the device to check for more packet arrivals.

Therefore, we can conclude that for interrupt-based I/O:

Pros:

1. Allow for overlap of computation and I/O
2. Better efficiency and utilization of CPU

Cons:

1. Cause system to slow down if it performs its task very quickly.
2. May lead to livelock in OS when a huge stream of incoming packets in network each generate interrupt, never allowing user-level process to run.

(2)

PIO

We can find the description of PIO in book as this:

take place to transfer a disk block (say 4KB) to the device. When the main CPU is involved with the data movement (as in this example protocol), we refer to it as **programmed I/O (PIO)**. Third, the OS writes a command to the command register; doing so implicitly lets the device know that

Therefore, PIO is the abbrevation of programmed I/O. It means that the main CPU is involved with the data movement from disk block to device.

DMA

We can find the description of DMA in book as this:

The solution to this problem is something we refer to as **Direct Memory Access (DMA)**. A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention.

Therefore, DMA is the abbrevation of Direct Memory Access. This refers to data movement from the device to the main memory, without much CPU intervention.

Difference between PIO and DMA:

1. As for data movement, PIO requires much CPU involvement, but DMA not.
2. The data transfer speed of DMA is faster compared to PIO because it doesn't involve the CPU.
3. DMA is more complex compared to PIO because it requires specialised hardware.
4. DMA is suitable for transferring large amounts of data between devices and main memory while PIO suit for small amounts of data transfer.

(3)

memory-mapped I/O

According to the content in book below:

> The second method to interact with devices is known as **memory-mapped I/O**. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

For memory-mapped I/O, it uses I/O protection with address translation just like main memory.

The hardware needs to make device registers available. And in order to access the register, the OS needs to issue load/store instruction to the address, then routes the load/store to device.

So it must be controlled by OS and can be protected from being abused by user process.

explicit I/O instruction

According to the content in book below:

> years) is to have explicit **I/O instructions**. These instructions specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols described above.
>
> Such instructions are usually **privileged**. The OS controls devices, and the OS thus is the only entity allowed to directly communicate with them. Imagine if any program could read or write the disk, for example: total chaos (as always), as any user program could use such a loophole to gain complete control over the machine.

Making instructions become privileged and the OS thus is the only entity allowed to directly communicate with them, which protects it from being abused.

2

Design idea:

For the struct `condvar_t`, we choose to encapsulate semaphore.

Therefore, for `cond_signal()`, we just need to activate the semaphore inside the `condvar_t`.

And for `cond_wait()`, first we release mutex, and then deactivate semaphore, finally we deactivate mutex.

Code:

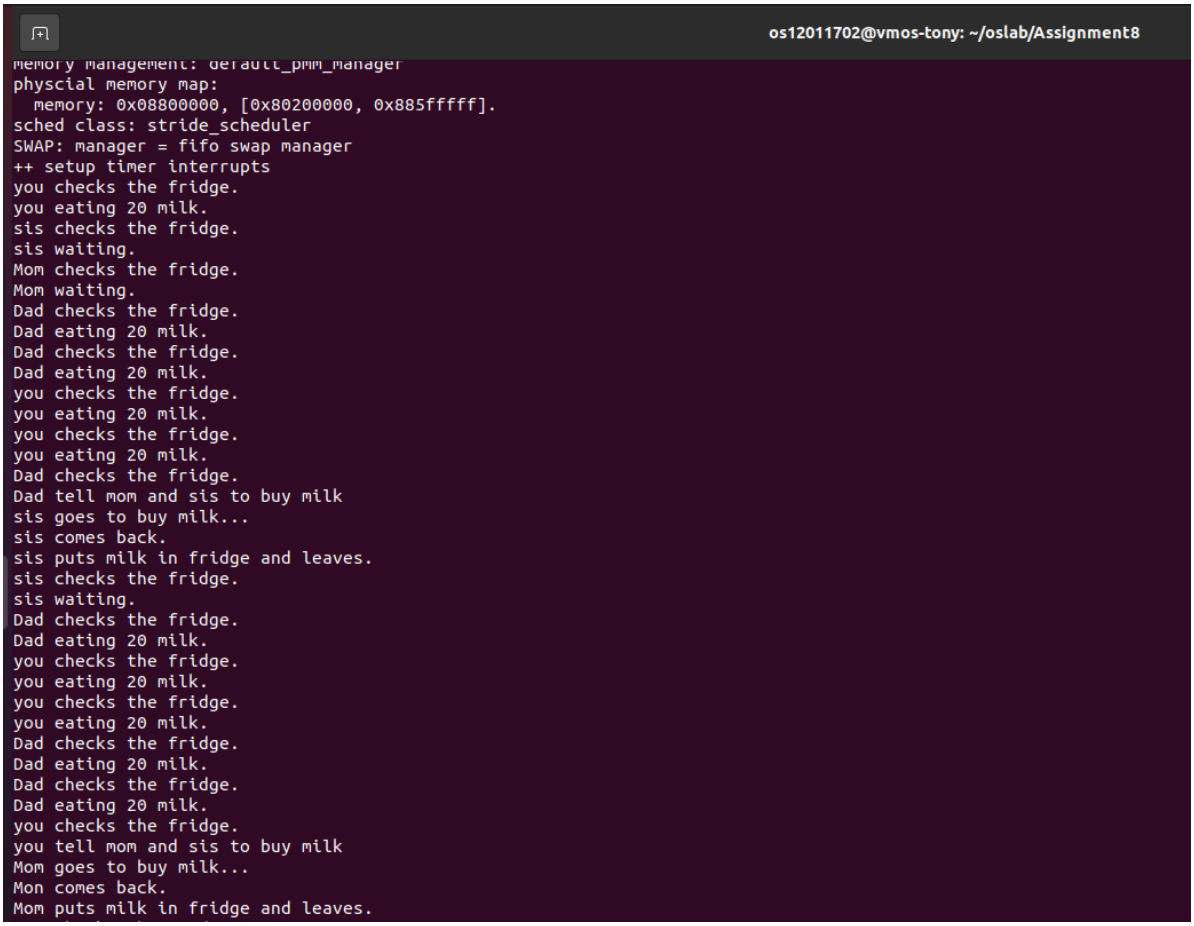
`condvar.h`:

```
#ifndef __KERN_SYNC_MONITOR_CONDVAR_H__
#define __KERN_SYNC_MOINTOR_CONDVAR_H__

#include <sem.h>

typedef struct condvar{
//================your code=====================
   semaphore_t sem;
} condvar_t;


void     cond_init (condvar_t *cvp);

void     cond_signal (condvar_t *cvp);

void     cond_wait (condvar_t *cvp, semaphore_t *mutex);

#endif /* !__KERN_SYNC_MONITOR_CONDVAR_H__ */
```

`condvar.c`:

```
#include <stdio.h>
#include <condvar.h>
#include <kmalloc.h>
#include <assert.h>

void
cond_init (condvar_t *cvp) {
//================your code=====================
    sem_init(&(cvp->sem),0);

}

// Unlock one of threads waiting on the condition variable.
void
cond_signal (condvar_t *cvp) {
//================your code=====================
    up(&(cvp->sem));

}

void
cond_wait (condvar_t *cvp,semaphore_t *mutex) {
//================your code=====================
    up(mutex);
    down(&(cvp->sem));
    down(mutex);

}
```

Running result:


os12011702@vmos-tony: ~/oslab/Assignment8
Memory Management: default_pmm_manager
physcial memory map:
  memory: 0x08800000, [0x80200000, 0x885fffff].
sched class: stride_scheduler
SWAP: manager = fifo swap manager
++ setup timer interrupts
you checks the fridge.
you eating 20 milk.
sis checks the fridge.
sis waiting.
Mom checks the fridge.
Mom waiting.
Dad checks the fridge.
Dad eating 20 milk.
Dad checks the fridge.
Dad eating 20 milk.
you checks the fridge.
you eating 20 milk.
you checks the fridge.
you eating 20 milk.
Dad checks the fridge.
Dad tell mom and sis to buy milk
sis goes to buy milk...
sis comes back.
sis puts milk in fridge and leaves.
sis checks the fridge.
sis waiting.
Dad checks the fridge.
Dad eating 20 milk.
you checks the fridge.
you eating 20 milk.
you checks the fridge.
you eating 20 milk.
Dad checks the fridge.
Dad eating 20 milk.
Dad checks the fridge.
Dad eating 20 milk.
you checks the fridge.
you tell mom and sis to buy milk
Mom goes to buy milk...
Mon comes back.
Mom puts milk in fridge and leaves.