# Assignment8 Report

姓名：Yitong WANG(王奕童) 11910104@mail.sustech.edu.cn

学号：11910104

实验课时段：周五5-6节

实验课教师：Yun SHEN(沈昀) sheny@mail.sustech.edu.cn

实验课SA:

- Yining TANG(汤怡宁) 11811237@mail.sustech.edu.cn
- Yushan WANG(王宇杉) 11813002@mail.sustech.edu.cn

# 1. I/O

## (1) What are the pros and cons of polling and interrupt-based I/O?

For the polling I/O:

- Pros: It can make interaction of device efficiently. This can be seen in the 3rd page in the book chapter:

  > 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

- Cons: Consider the following in the 4th page in the book chapter:

> 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.

2 aspects:

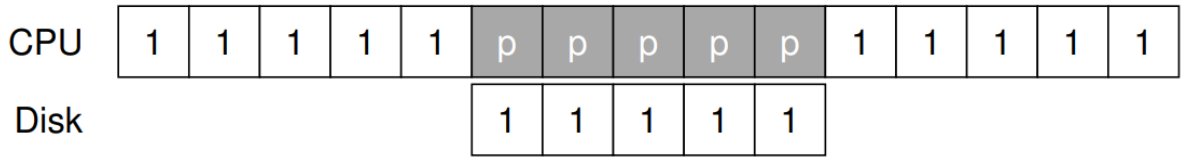
1. polling is inefficient

2. polling wastes too much CPU time, just for waiting for the device to finish the tasks.

For the interrpt-based I/O:

- Pros: Interrupts can allow the overlap for computation and I/O, thus can improve the utilization. It can be seen in the 5th page in the book chapter:

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




- Cons: Consider the following in the 6th page in the book chapter

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

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.

2 aspects:

1. it slows down the system which performs the tasks quickly.
2. in network, it does not allow the user-level process to run and service the requests when it is handling the interrupts.

## (2) What are the differences between PIO and DMA?

PIO: Programming I/O. This refers to the data movement from the disk device to the device with the main CPU involved.

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

DMA: Direct Memory Access. This refers to the progress of data movement from the device to the main memory, without much CPU involvation involvement.

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.

Differences:

1. DMA can have better performance than PIO in the usual case. Since PIO may use CPU while DMA not, CPU is slower than DMA.
2. PIO is much cheaper than DMA, in the area of circuitry design. Thus in the devices not necessary to have great performance, PIO has better cost performance than DMA.

> Reference: http://www.differencebetween.net/technology/difference-between-dma-and-pio/

## (3) How to protect memory-mapped I/O and explicit I/O instructions from being abused by malicious user process?

For the memory-mapped I/O, consider the following statements in the page 8 in the book chapter:

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.

There is not some great advantage to one approach or the other. The memory-mapped approach is nice in that no new instructions are needed to support it, but both approaches are still in use today.

To use memory-mapped I/O, the hardware needs to make the reigsters available. And in order to access the register, the OS needs to issue load/store instruction to the address. Thus it must be controlled by OS and can be protected from being abused by user process.

Foe the explicit I/O, Consider the following statements in the page 8 in the book chapter:

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.

we let these instructions become privileged thus only OS can invoke these instructions.

# 2. Condition variable

- Design idea: Just encapsulate one semaphore in the condvar_t. For the signal function, just activate the semaphore inside the condvar_t.
  For the wait function, first release the mutex, then deactivate the semaphore, finally deactivate the mutex.
- Code screeshots:

condvar.h:

```
1  #ifndef __KERN_SYNC_MONITOR_CONDVAR_H__
2  #define __KERN_SYNC_MOINTOR_CONDVAR_H__
3
4  #include <sem.h>
5
6  typedef struct condvar{
7  //================your code=====================
8      semaphore_t sem;
9  } condvar_t;
10
11
12
13 void     cond_init (condvar_t *cvp);
14
15 void     cond_signal (condvar_t *cvp);
16
17 void     cond_wait (condvar_t *cvp, semaphore_t *mutex);
18
19 #endif /* !__KERN_SYNC_MONITOR_CONDVAR_H__ */
20
```

condvar.c:

home > wyt11910104 > Desktop > Assignment > A8 > Week13 > kern > sync > C condvar.c > cond_init(condvar_t *)

```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);

}

// reference1: https://github.com/Trinkle23897/os2019/blob/0274c5919a8356081e3c838fa1277394ff78cd54/
// reference2: https://chyyuu.gitbooks.io/ucore_os_docs/content/lab7/lab7_3_4_monitors.html
```

- Running result:

```
wyt11910104@wyt11910104-virtual-machine:~/Desktop/Assignment/A8/Week13$ make qemu

OpenSBI v0.6
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 120 KB
Runtime SBI Version    : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
OS is loading ...

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.
```

# 3. Bike

- Design idea: Use 1 overall mutex to represent the lock, and 3 condition variables to represent the lock for the 3 workers.
- Code screenshots:

check_exercise.c:

```c
// kern/sync/check_exercise.c
#include <stdio.h>
#include <proc.h>
#include <sem.h>
#include <assert.h>
#include <condvar.h>

struct proc_struct *pworker1,*pworker2,*pworker3;
semaphore_t overallMutex;
condvar_t ws[3];

int index = 0;

void worker1(int i)
{

    while (1)
    {
        cond_wait(&(ws[0]), &overallMutex);
        cprintf("make a bike rack\n");
        index++;
        index %=3;
        cond_signal(&(ws[index]));

    }
}

```

```
28  void worker2(int i)
29  {
30      while (1)
31      {
32          cond_wait(&(ws[1]), &overallMutex);
33          cprintf("make two wheels\n");
34          index++;
35          index %=3;
36          cond_signal(&(ws[index]));
37      }
38  }
39
40  void worker3(int i){
41      while (1)
42      {
43          cond_wait(&ws[2], &overallMutex);
44          cprintf("assemble a bike\n");
45          index++;
46          index %=3;
47          cond_signal(&(ws[index]));
48      }
49  }
50
51
```

```
52  void check_exercise(void){
53
54      //initial
55      sem_init(&(overallMutex), 0);
56      cond_init(&(ws[0]));
57      cond_init(&(ws[1]));
58      cond_init(&(ws[2]));
59      cond_signal(&(ws[index]));
60      int pids[3];
61      int i =0;
62      pids[0]=kernel_thread(worker1, (void *)i, 0);
63      pids[1]=kernel_thread(worker2, (void *)i, 0);
64      pids[2]=kernel_thread(worker3, (void *)i, 0);
65      pworker1 = find_proc(pids[0]);
66      set_proc_name(pworker1, "worker1");
67      pworker2 = find_proc(pids[1]);
68      set_proc_name(pworker2, "worker2");
69      pworker3 = find_proc(pids[2]);
70      set_proc_name(pworker3, "worker3");
71  }
72
```

- Running result: