CS302 OS Week12 Assignment - Report

Name: 刘仁杰
SID: 11911808

1. I/O

1. What are the pros and cons of polling and interrupt-based I/O?
    - polling:
        - Pros: Simple and relatively lower overhead when the device is fast.
        - Cons: Inefficient, may waste many cycles on polling when the device is slow.
    - interrupt-based I/O:
        - Pros: Fully utilize CPU during I/O operations when the device is slow.
        - Cons: interrupts cost relatively high overhead when the device is fast.
2. What are the differences between PIO and DMA?

    In PIO, each byte is transferred via processor load/store. So the disk controller is simple and easy to program. However, this consumes processor cycles proportional to data size. In DMA, disk controller can directly access memory bus, so data blocks can be directly transferred to/from memory in parallel with the processor.
3. How to protect memory-mapped I/O and explicit I/O instructions from being abused by malicious user process?

    When the use processes launch memory-mapped I/O and explicit I/O instructions, OS is the only entity that can communicate with the device. Therefore, OS will guarantee that no harm will be caused by user processes.

2. Condition variable

- Design Idea:

We just need to use a semaphore to implement our conditional variable. When initiating the conditional variable, we just initiate the semaphore with value 0. Then in `cond_signal`, we just release a semaphore signal by `up`. In `cond_wait`, we first release the mutex semaphore and grab a conditional variable's semaphore. After we grab the semaphore by `down`, we just re-hold the mutex semaphore.

- Code:

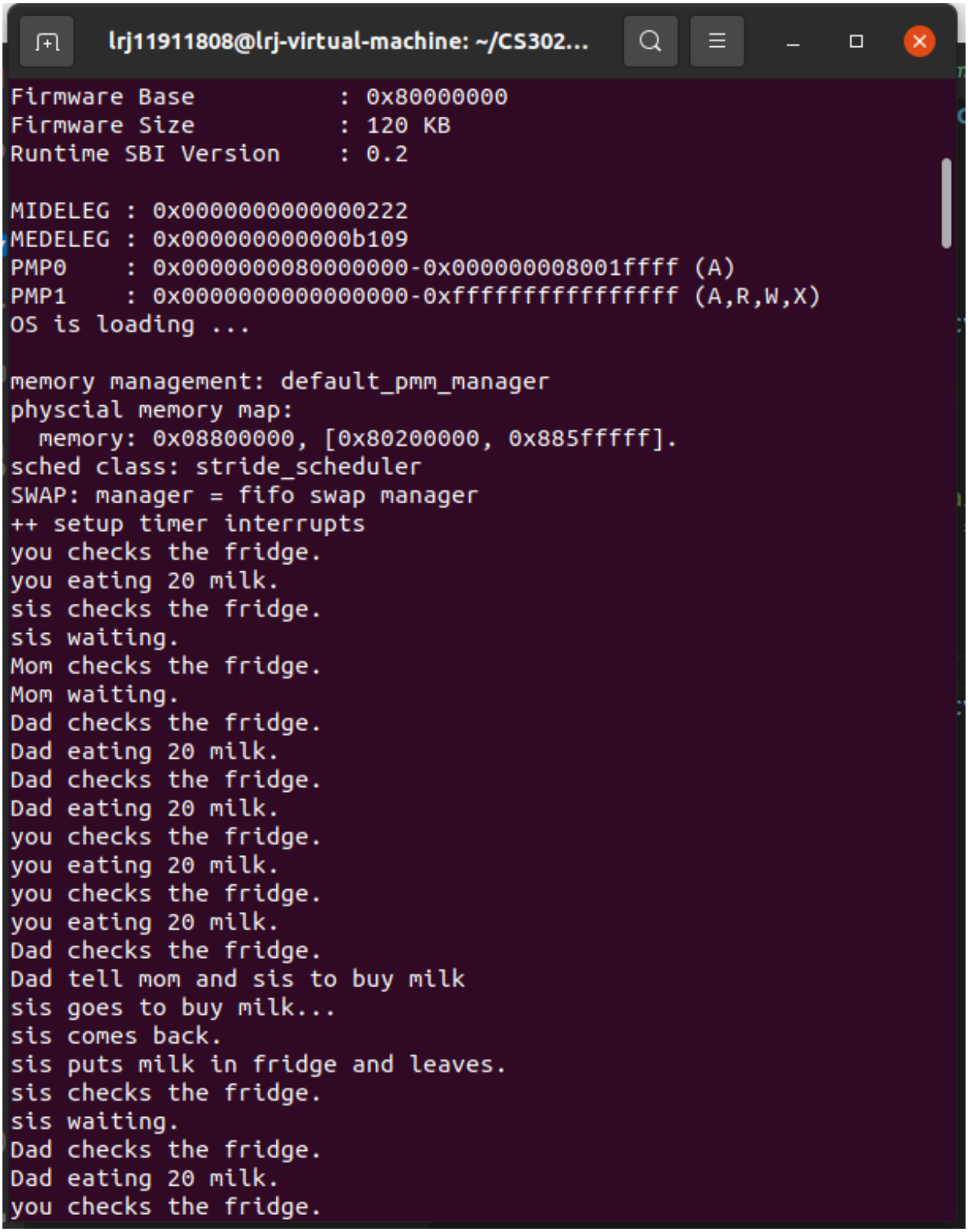
assignment > week13ass > Week13 > kern > sync > C condvar.h > condvar > sem

```h
#ifndef __KERN_SYNC_MONITOR_CONDVAR_H__
#define __KERN_SYNC_MOINTOR_CONDVAR_H__

#include <sem.h>

typedef struct condvar
{
    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__ */

```

assignment > week13ass > Week13 > kern > sync > C condvar.c > ...

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

void cond_init(condvar_t *cvp)
{
    sem_init(&cvp->sem, 0);
}

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

void cond_wait(condvar_t *cvp, semaphore_t *mutex)
{
    up(mutex);
    down(&cvp->sem);
    down(mutex);
}

```

- Running Results:



3. Bike

- Design Idea:

We can use three conditional variables and one mutex to complete the implementation. First worker1 makes a bike rack, then he release a conditional signal. Then the woker2 wait and receive this conditional signal. starts making two bike wheels and then releases the second conditional signal. Then the worker3 wait and receive the second conditional signal and starts assembling the bike. After this cycle, the worker3 release a third conditional signal to be taken by worker1 for the next cycle. Whenever a worker starts this work, he will first grab the mutex and release it when he finishes.

- Code:

```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;

condvar_t cond1, cond2, cond3;
semaphore_t mutex;

void worker1(int i)
{
    do_sleep(2);
    down(&mutex);
    cprintf("make a bike rack\n");
    cond_signal(&cond1);
    up(&mutex);
    while (1)
    {
        do_sleep(2);
        down(&mutex);
        cond_wait(&cond3, &mutex);
        cprintf("make a bike rack\n");
        cond_signal(&cond1);
        up(&mutex);
    }
}
```

```c
void worker2(int i)
{
    while (1)
    {
        do_sleep(2);
        down(&mutex);
        cond_wait(&cond1, &mutex);
        cprintf("make two wheels\n");
        cond_signal(&cond2);
        up(&mutex);
    }
}

void worker3(int i)
{
    while (1)
    {
        do_sleep(2);
        down(&mutex);
        cond_wait(&cond2, &mutex);
        cprintf("assemble a bike\n");
        cond_signal(&cond3);
        up(&mutex);
    }
}
```

```c
void check_exercise(void)
{

    // initial
    cond_init(&cond1);
    cond_init(&cond2);
    cond_init(&cond3);
    sem_init(&mutex, 1);

    int pids[3];
    int i = 0;
    pids[0] = kernel_thread(worker1, (void *)i, 0);
    pids[1] = kernel_thread(worker2, (void *)i, 0);
    pids[2] = kernel_thread(worker3, (void *)i, 0);
    pworker1 = find_proc(pids[0]);
    set_proc_name(pworker1, "worker1");
    pworker2 = find_proc(pids[1]);
    set_proc_name(pworker2, "worker2");
    pworker3 = find_proc(pids[2]);
    set_proc_name(pworker3, "worker3");
}
```

- Running Result: