Week 13

1

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

polling

pros

Simple and efficient enough

cons

However, it is inevitable that there will be some inefficiency and inconvenience. The first problem we note about this protocol is that the polling process is inefficient, wasting a lot of CPU time while waiting for the device to execute the completion command

interrupt

pros

Increase CPU utilization

cons

Although interrupts can overlap with I/O, this only makes sense on slow devices. Otherwise, the cost of additional interrupt handling and top-down switching will outweigh the benefits. In addition, if there are many interruptions in a short period of time, it may overload the system and cause a livelock . In this case, the polling method can provide more control over the operating system's scheduling, but it is more efficient.

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

The CPU is in charge of transporting data from one location to another via PIO. The CPU grows busier as the transfer speed increases, resulting in a substantial bottleneck in the computer's performance. DMA is not the same as PIO in terms of how it works. The CPU does not support data transport, leaving it free to perform other activities regardless of the rate of data transfer. This means that for determining the maximum rate of transmission, the CPU is irrelevant.

PIO controller is much simpler, therefore cheaper

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

memory-mapped I/O

When access to device registers is required, the operating system loads (reads) or deposits (writes) to that memory address; the hardware then transfers the load/deposit to the device instead of physical memory

explicit I/O

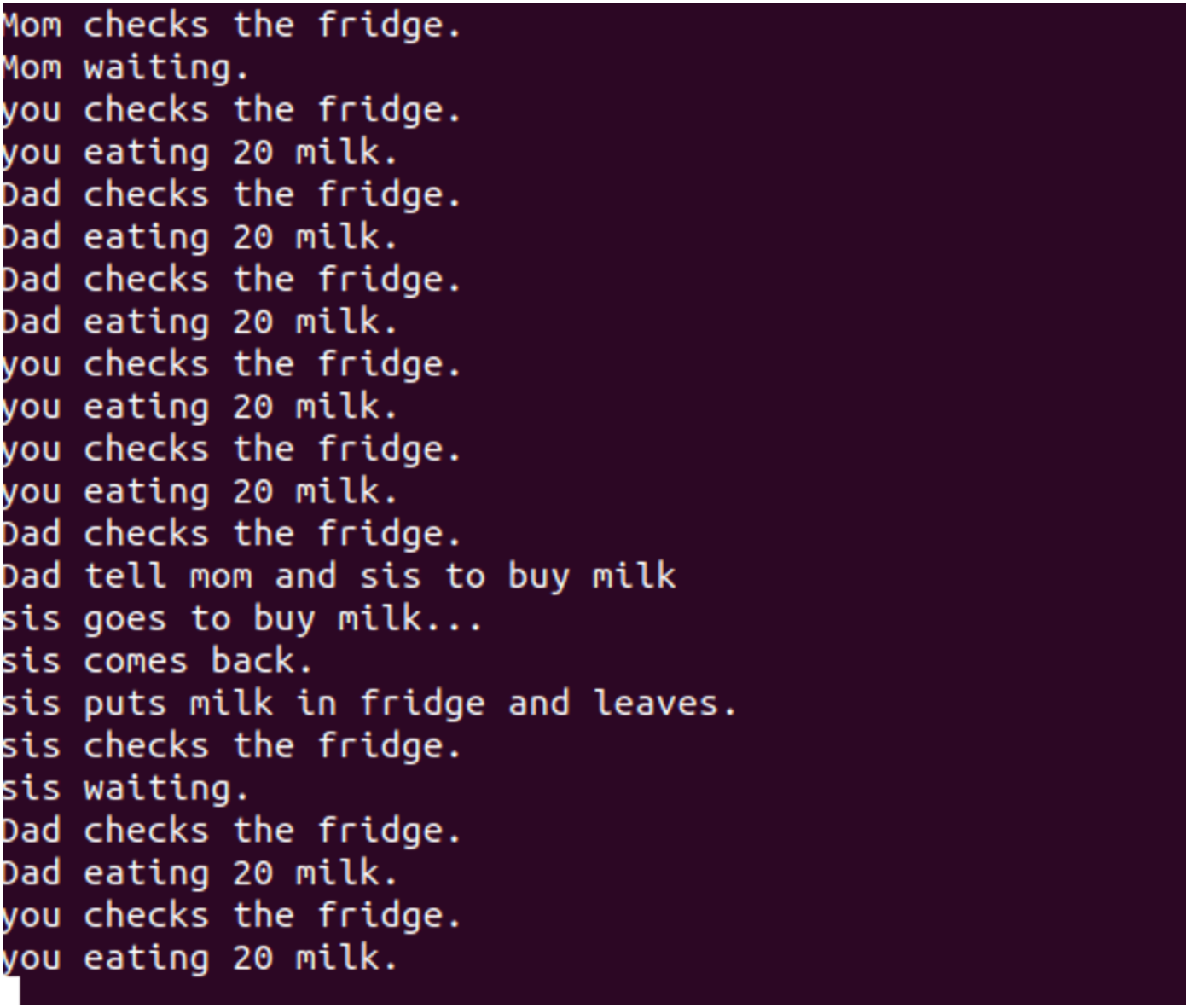
When access to device registers is required, the operating system loads (reads) or deposits (writes) to that memory address; the hardware then transfers the load/deposit to the device instead of physical memory.These directives are usually privileged directives. The operating system is the only entity that can interact directly with the device

2.

rt.

```c
void
cond_init (condvar_t *cvp) {
//================your code====================
    cvp->count = 0;
    sem_init( sem: &(cvp->sem), value: 0);
}

// Unlock one of threads waiting on the condition variable.
void
cond_signal (condvar_t *cvp) {
//================your code====================


        up( sem: &(cvp->sem));
//        down(&(cvp->owner->next));
//        cvp->owner->next_count--;

}

void
cond_wait (condvar_t *cvp,semaphore_t *mutex) {
//================your code====================



        cvp->count++;
        up( sem: mutex);
        down( sem: &(cvp->sem));
        cvp->count--;

}
```

```c
#include <sem.h>

typedef struct condvar{
//================your code====================
    semaphore_t sem;
    int count;

} 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__ */
```



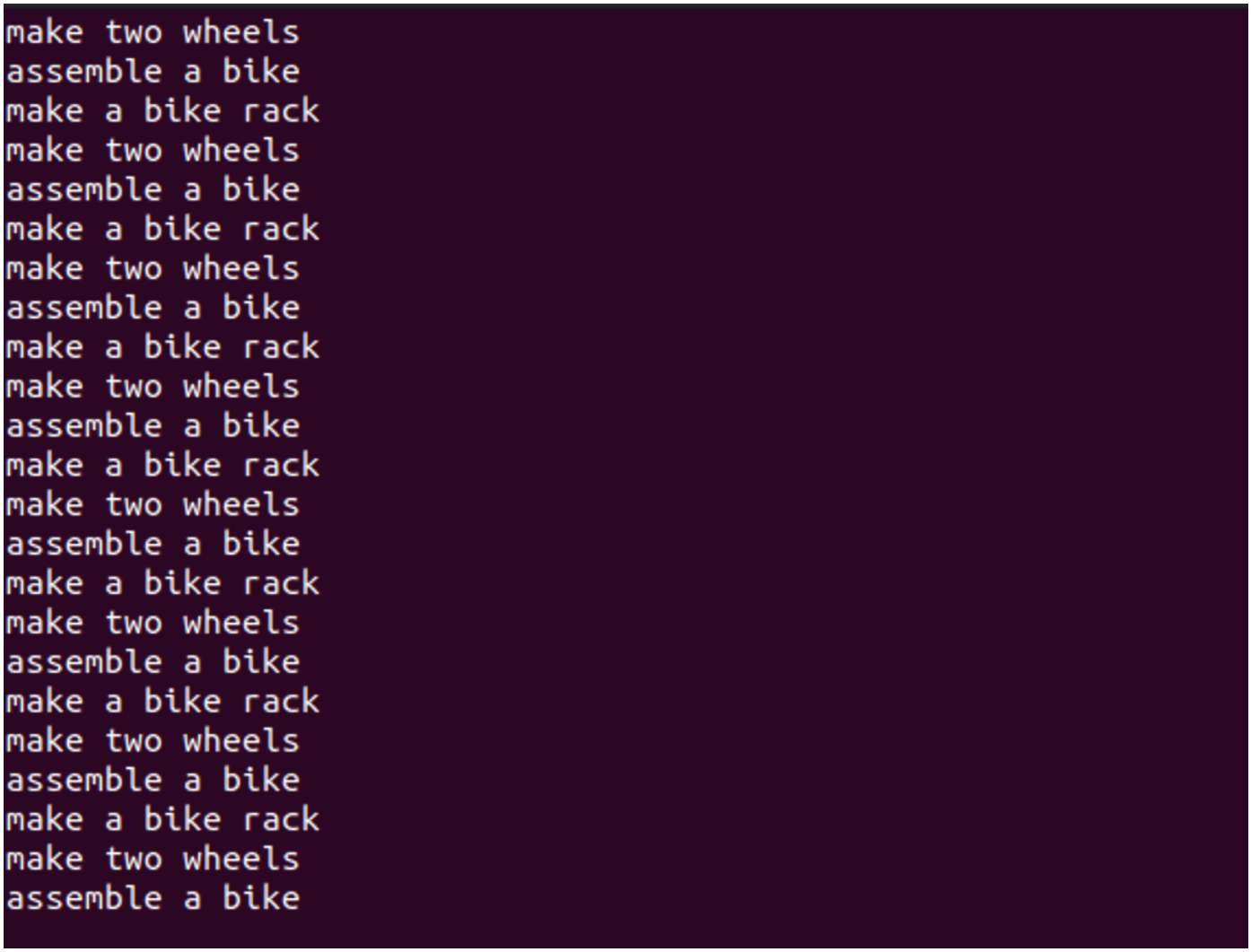
Easy to solve the problem. Very trivial. When wait. Free the mtex and wait for signal. When signal , announce the waited thread.

3

```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 mutex;
 condvar_t p1,p2,p3;

 void worker1(int i)
{
    int flag = 0;
    while (1)
    {   down( sem: &mutex);
        if(flag){
            cond_wait( cvp: &p3,  mutex: &mutex);
        }
        flag=1;

        cprintf("make a bike rack\n");
        cond_signal( cvp: &p1);


        up( sem: &mutex);
        do_sleep(100);
    }
}


void worker2(int i)
```

```
void worker2(int i)
{
    while (1)
    {
        down( sem: &mutex);
        cond_wait( cvp: &p1,  mutex: &mutex);

        cprintf("make two wheels\n");
        cond_signal( cvp: &p2);
        up( sem: &mutex);
        do_sleep(100);
    }
}




void worker3(int i){
    while (1)
    {
        cond_wait( cvp: &p2,  mutex: &mutex);
        down( sem: &mutex);
        cprintf("assemble a bike\n");
        cond_signal( cvp: &p3);
        up( sem: &mutex);
        do_sleep(100);
    }
}
```

```
void check_exercise(void){

    //initial
    sem_init( sem: &(mutex), value: 1);
    cond_init( cvp: &p1);
    cond_init( cvp: &p2);
    cond_init( cvp: &p3);
    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");
}
```



Use three conditional variable, w1 wait for w3 w2 wait w1 w3 wait w2 and first let w1 access.