Lab12 POSIX

1、OVERVIEW

In this lab, we will learn some POSIX APIs.

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines both the system- and user-level application programming interfaces (API), along with command line shells and utility interfaces, for software compatibility (portability) with variants of Unix and other operating systems. POSIX is also a trademark of the IEEE. POSIX is intended to be used by both application and system developers.

2、OBJECTIVE

1. POSIX Pthread
2. spin_lock vs mutex
3. semaphore
4. condition variables
5. shared memory
6. message queue

3、STEP

Read & run all the sample codes:

Step 1. POSIX Pthread

Function	Description
pthread_create	create a new thread
pthread_exit	terminate calling thread
pthread_cancel	send a cancellation request to a thread
pthread_join	wait for the specified thread
pthread_detach	detach a thread

Sample code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

int a = 0;

void *add()
```

```
{
    for (int i = 0; i < 3; i++)
    {
        int b = a;
        b++;
        a = b;
        printf("%d\n", a);
    }
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;

    // Create two threads (both run func)
    pthread_create(&p1, NULL, add, NULL);
    pthread_create(&p2, NULL, add, NULL);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
}
```

> **Race Condition —— The too much milk problem**
>
> Mom and Dad are all used to checking the fridge when they arrive home. If milk run out, he or she will leave home to buy milk. The fridge is small in your home that only one bottle of milk can be put in it at a time. Mom and Dad always arrive home at different time.
>
> | Time | Dad | Mom |
> |---|---|---|
> | 3:00 | Arrive Home | |
> | 3:05 | Look in fridge, no milk | |
> | 3:10 | Leave for supermarket | |
> | 3:15 | | Arrive Home |
> | 3:20 | Arrive at Supermarket | Look in fridge, no milk |
> | 3:25 | Buy Milk | Leave for supermarket |
> | 3:30 | Arrive home, put milk into fridge | |
> | 3:35 | | Arrive at Supermarket |
> | 3:40 | | Buy milk |
> | 3:45 | | Arrive home, put milk in fridge |
> | 3:50 | | Oh ..... |
>
> **Compile and run `milk.c`**
>
> ```
> gcc milk.c -o milk -pthread
> ./milk
> ```

You may found the outcome of an execution depends on a particular order in which the shared resource is accessed.

Step 2. spin lock vs mutex

Function	Description
pthread_spin_init	initialize a spin lock
pthread_spin_lock	lock a spin lock
pthread_spin_unlock	unlock a spin lock
pthread_mutex_lock	lock a mutex
pthread_mutex_unlock	unlock a mutex
pthread_mutex_trylock	try to lock a mutex
pthread_spin_trylock	try to lock a spin lock

Sample code——spin lock:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
int a = 0;
pthread_spinlock_t spinlock;

void *add()
{
    for (int i = 0; i < 3; i++)
    {
        pthread_spin_lock(&spinlock);
        int b = a;
        b++;
        a = b;
        printf("%d\n", a);
        pthread_spin_unlock(&spinlock);
    }
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    pthread_spin_init(&spinlock, 0);

    // Create two threads (both run func)
    pthread_create(&p1, NULL, add, NULL);
    pthread_create(&p2, NULL, add, NULL);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    pthread_spin_destroy(&spinlock);
}
```

> **The too much milk problem**
>
> Try to fix the too much milk problem by spin_lock(busy waiting). When Dad go to buy milk, he will lock the fridge. If Mom found the fridge locked, she will spin before the fridge until it unlocked. Vice versa.

Sample code——mutex lock:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
int a = 0;
pthread_mutex_t mutex;

void *add()
{
    for (int i = 0; i < 3; i++)
    {
        pthread_mutex_lock(&mutex);
        int b = a;
        b++;
        a = b;
        printf("%d\n", a);
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    pthread_mutex_init(&mutex, NULL);

    // Create two threads (both run func)
    pthread_create(&p1, NULL, add, NULL);
    pthread_create(&p2, NULL, add, NULL);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    pthread_mutex_destroy(&mutex);
}
```

> **The too much milk problem**
>
> Try to fix the too much milk problem by mutex. When Dad go to buy milk, he will lock the fridge. If Mom found the fridge locked, she will sleep until it unlocked. Vice versa.
>
> **Better way**
>
> Try trylock to realize:
>
> - Mom see the fridge is empty.
> - Mom leave a note and then go buy milk.
> - Dad open the fridge and see the note.
> - Dad just go away.

Step 3. semphores——POSIX IPC(Inter-Process Communication)

How to realize `semaphore`:

```
struct semaphore {
       int count;
       queueType queue;
};
void semWait(semaphore s)
{
       s.count--;
       if (s.count < 0) {
         /* place this process in s.queue */;
         /* block this process */;
       }
}
void semSignal(semaphore s)
{
       s.count++;
       if (s.count<= 0) {
         /* remove a process P from s.queue */;
         /* place process P on ready list */;
       }
}
```

APIs:

Function	Description
sem_open	opens/creates a named semaphore for use by a process
sem_wait	lock a semaphore
sem_post	unlock a semaphore
sem_close	deallocates the specified named semaphore
sem_unlink	removes a specified named semaphore
sem_getvalue	get semaphore value

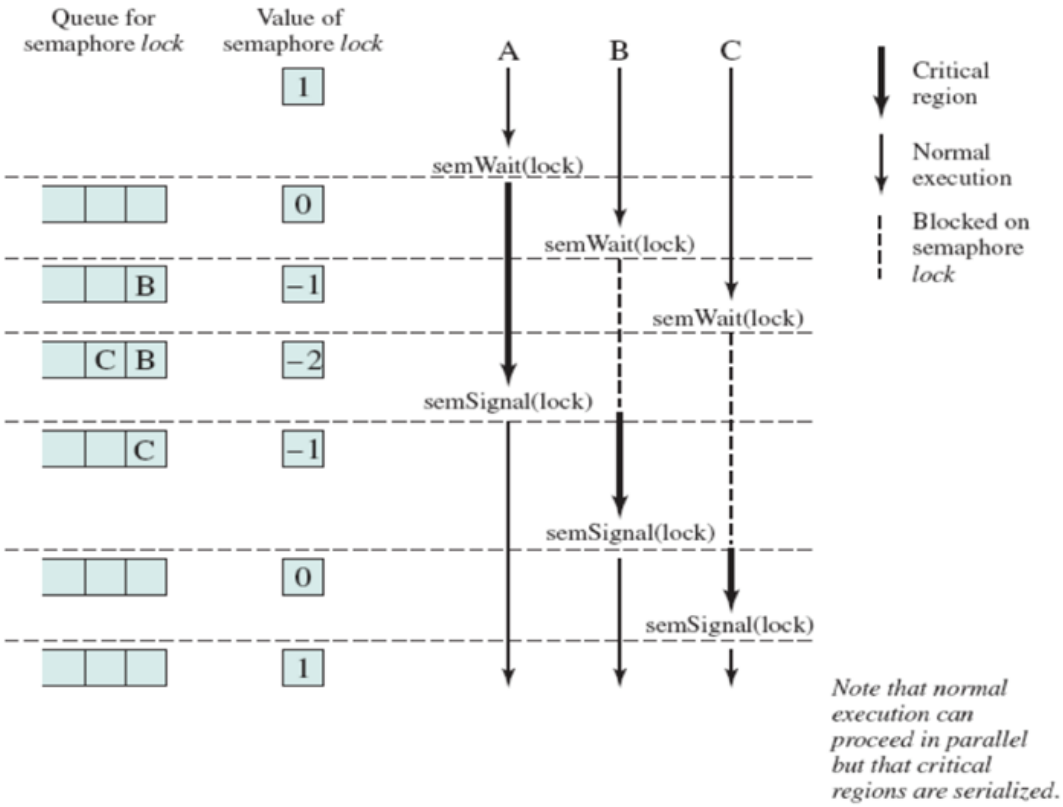
Sample code:

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>
#include <stdio.h>

int main(int argc, char * argv[]){
    char * name = "my_semaphore";
    int VALUE = 2;
    sem_t * sema;
    //If semaphore with name does not exist, then create it with VALUE
    printf("Open or Create a named semaphore, %s, its init value is %d\n",
name,VALUE);
    sema = sem_open(name, O_CREAT, 0666, VALUE);
```

```
    //wait on semaphore sema and decrease it by 1
    sem_wait(sema);
    sem_getvalue(sema, &VALUE);
    printf("Decrease semaphore by 1, now the value is %d\n", VALUE);
    //add semaphore sema by 1
    sem_post(sema);
    sem_getvalue(sema, &VALUE);
    printf("Add semaphore by 1, now the value is %d\n", VALUE);
    //Before exit, you need to close semaphore and unlink it, when all
processes have
    //finished using the semaphore, it can be removed from the system using
sem_unlink
    sem_close(sema);
    sem_unlink(name);
    return 0;
}
```

How to use:



> **The too much milk problem**
>
> Now your family bought a new fridge, it has space to store 2 bottles of milk. And you began to buy milk, there are three people in your family who buy milk. A person buys only one bottle of milk at a time.
>
> Try to use semaphore to solve this problem.

Step 4. condition variable

Function	Description
pthread_cond_wait	release lock, put thread to sleep until condition is signaled; when thread wakes up again, re-acquire lock before returning.
pthread_cond_signal	Wake up at least one of the threads that are blocked on the specified condition variable; If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked.

Sample code:

```
//Producers and Consumers.
//Two producers vs two consumers
//At any time, only one person can access count

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
void *producer(void *arg)
{
    int i = 5;
    while (i--)
    {
        pthread_mutex_lock(&mutex);
        printf("producer add lock\n");
        count++;
        printf("in producer count is %d\n", count);
        if (count > 0)
        {
            pthread_cond_signal(&cond);
        }
        printf("producer release lock\n");
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *consumer(void *arg)
{
    int i = 5;
    while (i--)
    {
        pthread_mutex_lock(&mutex);
        printf("consumer add lock\n");
        if (count <= 0)
        {
            printf("begin wait\n");
            pthread_cond_wait(&cond, &mutex);
            printf("end wait\n");
```

```
        }
        count--;
        printf("in consumer count is %d\n", count);
        pthread_mutex_unlock(&mutex);
        printf("consumer release lock\n");
    }
    return NULL;
}

int main()
{
    pthread_t producethread1, producethread2, consumethread1, consumethread2;
    pthread_create(&consumethread1, NULL, consumer, NULL);
    pthread_create(&consumethread2, NULL, consumer, NULL);
    pthread_create(&producethread1, NULL, producer, NULL);
    pthread_create(&producethread2, NULL, producer, NULL);
    pthread_join(producethread1, NULL);
    pthread_join(consumethread1, NULL);
    pthread_join(producethread2, NULL);
    pthread_join(consumethread2, NULL);
    return 0;
}
```

The too much milk problem——new problem

- Your family buy a new big fridge, which can put in 100 bottles of milk.
- Dad and you always take milk but never buy.
- Mom and sister is very frequently checking the fridge is empty or not.
- If fridge is empty, she will go buy milk. Otherwise do nothing.

The problem can be solved like:

Dad and you(two threads)

```
while (1){
 lock
 int num=check_fridge()
 if(num>0)
     take milk
 else  cond_signal
 unlock
}
```

Mom and sister(two threads)

```
while (1){
 lock//lock mutex
 while(check_fridge()>0)//question 2
     cond_wait  //wait for cond_signal and unlock mutex
 go buy milk
 unlock
}
```

Try to realize the above solution by condition variable, and think about:

1. How to wake up a thread which entered cond_wait?

> 2. What will happen if change while to if?
> 3. Why cond_signal first rather than unlock mutex first?

Step 5. shared memory——POSIX IPC

Function	Description
shm_open	create/open POSIX shared memory objects
mmap	map files or devices into memory
munmap	unmap files or devices into memory
shm_unlink	unlink POSIX shared memory objects
ftruncate	truncate a file to a specified length

Sample code:

```
/***** headers.h ******/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

/***** Makefile ********/
all:
    gcc -o producer producer.c -lrt
    gcc -o consumer consumer.c -lrt

/******* producer.c *******/
#include "headers.h"

int main()
{
    const char *name = "OS";
    const char *message = "Learning operating system is fun!";
    int shm_fd;
    void *ptr;
    /* create the shared memory segment */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory segment */
    ftruncate(shm_fd, 4096);

    /* now map the shared memory segment in the address space of the process */
    ptr = mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED)
    {
        printf("Map failed\n");
```

```
        return -1;
    }
    sprintf(ptr, "%s", message);
}

/********** consumer.c ***********/
#include "headers.h"
int main()
{
    const char *name = "OS";
    int shm_fd; // file descriptor, from shm_open()
    char *ptr;  // base address, from mmap()
    /* open the shared memory segment as if it was a file */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    if (shm_fd == -1)
    {
        printf("Shared memory failed\n");
        exit(1);
    }
    /* map the shared memory segment to the address space of the process */
    ptr = mmap(0, 4096, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED)
    {
        printf("Map failed\n");
        exit(1);
    }
    /* Read data */
    printf("%s", ptr);
    /* remove the named shared memory object*/
    shm_unlink(name);
}
```

Step 6. message queue——POSIX IPC

Function	Description
mq_open	open a message queue
mq_close	close a message queue descriptor
mq_getattr	get message queue attributes
mq_setattr	set message queue attributes
mq_send	send a message to a message queue
mq_receive	receive a message from a message queue

Sample code:

```
//gcc -lrt
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
```

```c
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>


int main()
{
    mqd_t mqID;
    mqID = mq_open("/mq", O_RDWR | O_CREAT | O_EXCL, 0666, NULL);

    if (fork() == 0)
    {
        struct mq_attr mqAttr;
        mq_getattr(mqID, &mqAttr);

        char buf[mqAttr.mq_msgsize];

        for (int i = 1; i <= 5; ++i)
        {
            if (mq_receive(mqID, buf, mqAttr.mq_msgsize, NULL) < 0)
            {
                printf("receive message failed. \n");
                continue;
            }

            printf("receive message %d:%s\n",i,buf);
        }
        exit(0);
    }

    char msg[] = "haha";
    for (int i = 1; i <= 5; ++i)
    {
        if (mq_send(mqID, msg, sizeof(msg), i) < 0)
        {
            printf("send message %d failed.\n",i);
            continue;
        }
        printf("send message %d success.\n",i);

        sleep(1);
    }
    mq_close(mqID);
}
```

4、Review

In this experiment, you should learn to use POSIX APIs.

5、Advance notice

In the next experiment, we will learn file system in ucore.