《操作系统》

实验七报告

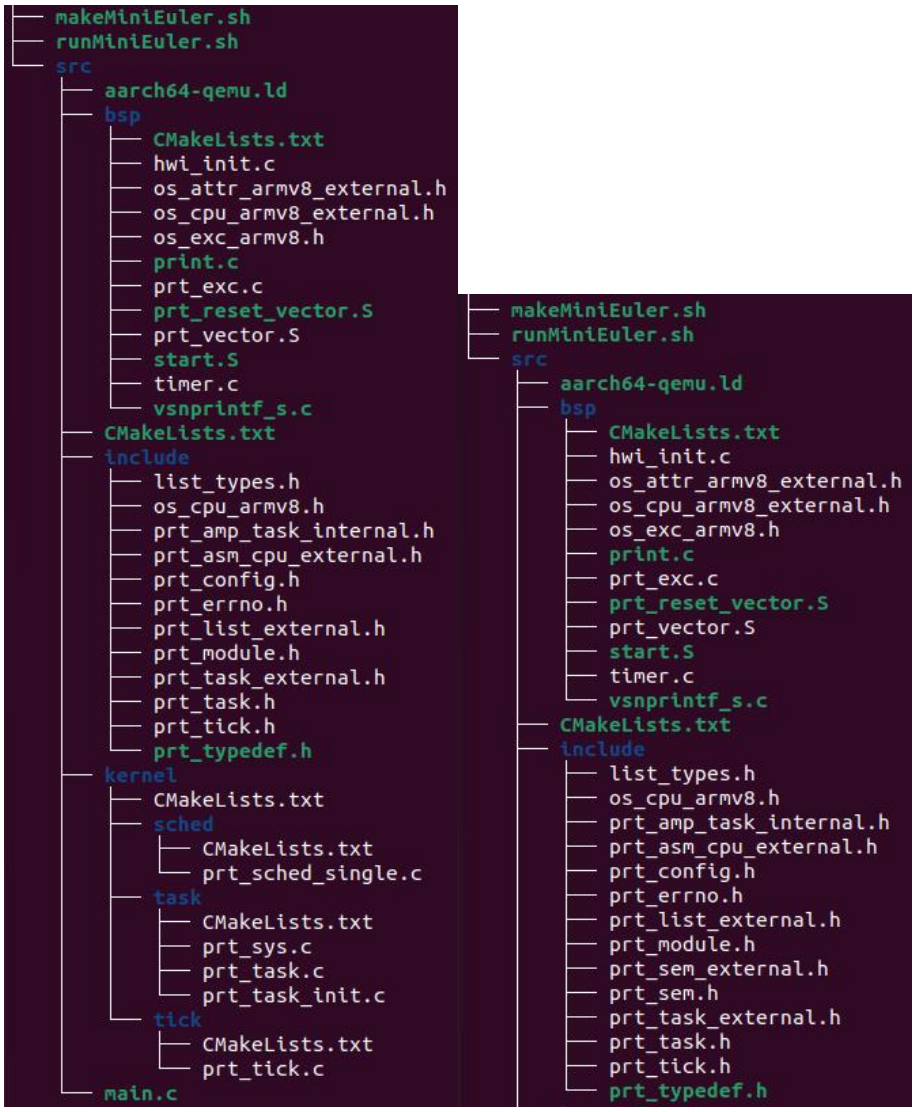
目录

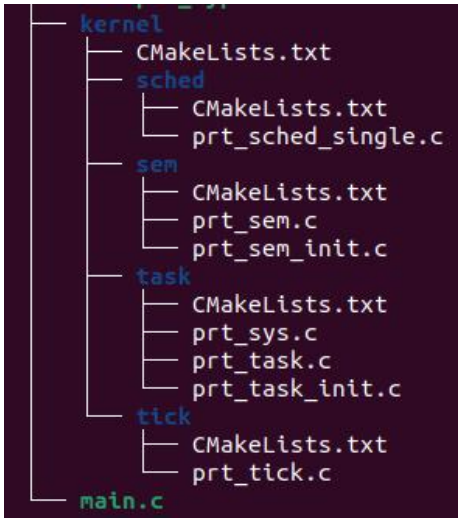
1 实验代码分析

1.1 实验目的

实现信号量与同步。

1.2 代码结构





对应 lab6 的结构来看，新建了 include/prt_sem_external.h 头文件和 include/prt_sem.h 头文件，新建了 sem 文件夹，在其中新建了 prt_sem_init.c 和 prt_sem.c 文件。对一些文件如 src/bsp/os_cpu_armv8_external.h ， prt_task_external.h ， src/kernel/task/prt_task.c ， src/bsp/os_cpu_armv8_external.h 中添加了个别函数。

1.3 信号量实现

新建 lab7/src/include/prt_sem_external.h 头文件，进行了一些处理信号量状态的相关宏的定义，对信号量结构体控制块进行了定义，声明了全局变量 g_maxSem 指针，OsSemCreate 和 OsSemBusy 两个相关函数。

新建 src/kernel/sem/prt_sem_init.c 文件。提供了 OsMemAllocAlign 信号量初始化函数和 OsSemCreate 和 PRT_SemCreate 两个创建信号量函数。

在 src/bsp/os_cpu_armv8_external.h 加入了信号量的定义。

新建 src/kernel/sem/prt_sem.c 文件。OsSemPostErrorCheck 实现了错误检查，OsSemPendListPut 把当前运行任务挂接到信号量链表上，OsSemPendListGet 从非空信号量链表上摘首个任务放入到 ready 队列，PRT_SemPend 指定信号量的 P（wait）操作，OsSemPostIsInvalid 判断信号量 post 是否有效， PRT_SemPost 指定信号量的 V（signal）操作。

src/include/prt_task_external.h 加入 OsTskReadyAddBgd()函数，将任务加入链表。

src/kernel/task/prt_task.c 加入 OsTskScheduleFastPs()函数。如果快速切换后只有中断恢复，使用该接口。在中断恢复时检查是否有更高优先级的任务等待，调用 OsTaskTrapFastPs()。

src/bsp/os_cpu_armv8_external.h 加入 OsTaskTrapFastPs()，被上个函数调用，执行调度。

新建 src/include/prt_sem.h，该头文件主要是信号量相关的函数声明和宏定义。

1.4 信号量测试

修改，将新建文件加入构建系统。

src/CMakeLists.txt:

```
list(APPEND OBJS $<TARGET_OBJECTS:bsp> $<TARGET_OBJECTS:tick> $<TARGET_OBJECTS:task> $<TARGET_OBJECTS:sched> $<TARGET_OBJECTS:sem>) # 增加 $<TARGET_OBJECTS:tick> 目标
add_executable(${APP} main.c ${OBJS})
```

src/kernel/CMakeLists.txt:

```
add_subdirectory(tick)
add_subdirectory(task)
add_subdirectory(sched)
add_subdirectory(sem)
```

src/kernel/sem/CMakeLists.txt:

```
set(SRCS prt_sem_init.c prt_sem.c )
add_library(sem OBJECT ${SRCS})  # OBJECT类型只编译生成.o目标文件，但不实际链接成库
```

修改 main 函数，进行任务调度测试。lab7 的 main 函数与 lab6 基本相同，仅在主题部分增加了启用信号量、初始化部分，修改了具体的调用函数，增加了使用信号量部分。

```
void Test1TaskEntry()
{
    PRT_Printf("task 1 run ...\n");
    PRT_SemPost(sem_sync);
    U32 cnt = 5;
    while (cnt > 0) {
        // PRT_TaskDelay(200);
        PRT_Printf("task 1 run ...\n");
        cnt--;
    }
}
```

```
void Test2TaskEntry()
{
    PRT_Printf("task 2 run ...\n");

    PRT_SemPend(sem_sync, OS_WAIT_FOREVER);
    U32 cnt = 5;
    while (cnt > 0) {
        // PRT_TaskDelay(100);
        PRT_Printf("task 2 run ...\n");
        cnt--;
    }
}
```

task2 中是 PRT_SemPend 函数，进行 P（wait）操作；task1 中是 PRT_SemPost 函数，进行 V（signal）操作。这种信号量的使用方式是条件变量，一个线程暂停执行，等待另一条件成立。**进入函数时，先执行 task2，执行到 PRT_SemPend 时，它会休眠，等待 task1 执行，执行到 signal 时唤醒 task2，再次进行调度时，调度程序选择执行 task2。因为我们这里是优先级调度，task2 的优先级高，所以一直到它执行完毕，再次执行 task1，结束。**

```
task 2 run ...
task 1 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
```

1.5 构建项目与执行结果

本实验逻辑为：

新建 lab7/src/include/prt_sem_external.h 头文件，对信号量进行定义。

新建 src/kernel/sem/prt_sem_init.c 文件，初始化和创建函数。

在 src/bsp/os_cpu_armv8_external.h 补充了信号量的定义。

新建 src/kernel/sem/prt_sem.c 文件，实现了信号量的具体操作。

src/include/prt_task_external.h 加入 OsTskReadyAddBgd()函数。

src/kernel/task/prt_task.c 加入 OsTskScheduleFastPs()函数，在中断恢复时检查是否有更高优先级的任务等待执行。src/bsp/os_cpu_armv8_external.h 加入 OsTaskTrapFastPs()函数，被上个函数调用。

新建 src/include/prt_sem.h，该头文件主要是信号量相关的函数声明和宏定义。

```
guoruiling@guoruiling-virtual-machine:~/lab7$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/
bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarc
h64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab7/build
[  5%] Building C object kernel/sem/CMakeFiles/sem.dir/prt_sem_init.c.obj
[ 11%] Building C object kernel/sem/CMakeFiles/sem.dir/prt_sem.c.obj
[ 11%] Built target sem
Scanning dependencies of target bsp
[ 17%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 23%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 29%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 35%] Building C object bsp/CMakeFiles/bsp.dir/vsnprintf_s.c.obj
[ 41%] Building C object bsp/CMakeFiles/bsp.dir/prt_exc.c.obj
[ 47%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_vector.S.obj
[ 52%] Building C object bsp/CMakeFiles/bsp.dir/hwi_init.c.obj
[ 58%] Building C object bsp/CMakeFiles/bsp.dir/timer.c.obj
[ 58%] Built target bsp
[ 64%] Building C object kernel/tick/CMakeFiles/tick.dir/prt_tick.c.obj
[ 64%] Built target tick
[ 70%] Building C object kernel/task/CMakeFiles/task.dir/prt_task.c.obj
[ 76%] Building C object kernel/task/CMakeFiles/task.dir/prt_sys.c.obj
[ 82%] Building C object kernel/task/CMakeFiles/task.dir/prt_task_init.c.obj
[ 82%] Built target task
[ 88%] Building C object kernel/sched/CMakeFiles/sched.dir/prt_sched_single.c.obj
[ 88%] Built target sched
[ 94%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
```

```
guoruiling@guoruiling-virtual-machine:~/lab7$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/m
iniEuler -s
------------ ------------    ------    -----------
 ************ ************   ********   ***********
----          ------------  ----------  ----    ---
************      ****     ****    **** *********
 ------------      ----     ------------ ---------
       *****     ****     ************ ****  ****
 ------------      ----     ----    ---- ----    ----
************      ****     ****    **** ****    ****
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

task 2 run ...
task 1 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
QEMU: Terminated
```

2 实验任务

2.1 作业 1

题目：各种并发问题模拟，至少 3 种。

在 Python 中，我们可以使用内置的 threading 模块来模拟并发问题。

2.1.1 数据竞争

数据竞争是当两个或更多的线程并发访问同一个内存位置，且至少有一个线程是写入操作，并且线程之间没有使用适当的同步来协调这些访问时发生的情况。

```python
import threading

counter = 0

def increment():
    global counter
    for _ in range(100000):
        counter += 1  # 没有同步，可能导致数据竞争

threads = []
for _ in range(2):
    t = threading.Thread(target=increment)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("Final counter value:", counter)  # 可能不是200000，因为存在数据竞争
```

```
guoruiling@guoruiling-virtual-machine:~/os$ python3 ./test1
Final counter value: 195807
```

出现了竞争问题。

2.1.2 死锁

死锁发生在多线程或多进程环境中，当两个或更多的线程/进程互相等待对方释放资源时，就会形成死锁。这种情况下，线程/进程会陷入无限等待的状态，从而导致程序无法正常执行。

```
guoruiling@guoruiling-virtual-machine:~/os$ python3 ./test2
Thread 1: Holding lock 1... Waiting for lock 2...
Thread 2: Holding lock 2... Waiting for lock 1...
```

互相等待以至于无法正常执行。

```python
import threading

class DeadlockDemo:
    def __init__(self):
        self.lock1 = threading.Lock()
        self.lock2 = threading.Lock()

    def thread1(self):
        self.lock1.acquire()
        print("Thread 1: Holding lock 1... Waiting for lock 2...")
        # 模拟一些工作
        threading.Event().wait(1)
        self.lock2.acquire()  # 这可能会导致死锁，如果线程2已经持有了lock2
        print("Thread 1: Holding lock 1 and 2...")
        self.lock2.release()
        self.lock1.release()

    def thread2(self):
        self.lock2.acquire()
        print("Thread 2: Holding lock 2... Waiting for lock 1...")
        # 模拟一些工作
        threading.Event().wait(1)
        self.lock1.acquire()  # 这会导致死锁，如果线程1已经持有了lock1
        print("Thread 2: Holding lock 1 and 2...")
        self.lock1.release()
        self.lock2.release()

demo = DeadlockDemo()
t1 = threading.Thread(target=demo.thread1)
t2 = threading.Thread(target=demo.thread2)

t1.start()
t2.start()

t1.join()
t2.join()  # 这两个线程可能会陷入死锁
```

2.1.3 活锁

活锁是并发系统中的一种状态，其中线程不断地更改状态，导致它们无法继续执行其任务，但并没有发生死锁（即没有线程/进程被阻塞）。这通常是由于不恰当的同步机制或优先级调度导致的。

```
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
Thread 1: Waiting for flag to be 0...
^Z
[2]+  Stopped                 python3 ./test3
```

无限循环，需要手动停止。

```
1   import threading
  1 import time
  2
  3 class LivelockDemo:
  4     def __init__(self):
  5         self.flag = 0
  6
  7     def thread1(self):
  8         while True:
  9             if self.flag == 0:
 10                 print("Thread 1: Trying to do work...")
 11                 # 模拟工作
 12                 time.sleep(0.1)
 13                 self.flag = 1
 14                 print("Thread 1: Changed flag to 1")
 15             else:
 16                 print("Thread 1: Waiting for flag to be 0...")
 17                 # 线程没有阻塞，但无法完成其任务
 18
 19     def thread2(self):
 20         while True:
 21             if self.flag == 1:
 22                 print("Thread 2: Trying to do work...")
 23                 # 模拟工作
 24                 time.sleep(0.1)
 25                 self.flag = 0
 26                 print("Thread 2: Changed flag to 0")
 27             else:
 28                 print("Thread 2: Waiting for flag to be 1...")
 29                 # 线程没有阻塞，但无法完成其任务
 30
 31 demo = LivelockDemo()
 32 t1 = threading.Thread(target=demo.thread1)
 33 t2 = threading.Thread(target=demo.thread2)
 34
 35 t1.start()
 36 t2.start()
```

2.1.4 饥饿

饥饿是指一个或多个线程因为其他线程不断占用资源而无法获得足够的 CPU 时间或其他资源来执行其任务。

```
Greedy thread: Using up CPU time...
Starving thread: Trying to get some CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
Greedy thread: Using up CPU time...
^Z
[3]+  Stopped                 python3 ./test4
```

饥饿线程无法获取资源，被饿死。

```
  import threading
 1 import time
 2
 3 def starving_thread():
 4     while True:
 5         print("Starving thread: Trying to get some CPU time...")
 6         # 模拟工作
 7         time.sleep(1)
 8
 9 def greedy_thread():
10     while True:
11         # 贪婪线程占用大量CPU时间
12         for _ in range(1000000):
13             pass
14         print("Greedy thread: Using up CPU time...")
15
16 t_starving = threading.Thread(target=starving_thread)
17 t_greedy = threading.Thread(target=greedy_thread)
18
19 t_starving.start()
20 t_greedy.start()
```

2.1.5 优先级反转

优先级反转是一个在多线程或多进程环境中可能出现的问题，它指的是高优先级的任务被低优先级的任务阻塞，导致高优先级任务迟迟得不到调度，而中等优先级的任务却能抢到 CPU 资源。

```
guoruiling@guoruiling-virtual-machine:~/os/test$ python3 ./test5
LowPriority: 低优先级线程获取资源
LowPriority: 低优先级线程持有资源并开始工作
MediumPriority: 中优先级线程开始执行，占用CPU资源
HighPriority: 需要资源来执行高优先级任务
LowPriority: 低优先级线程完成工作，释放资源
HighPriority: 获取到资源，开始执行高优先级任务
HighPriority: 高优先级任务完成
MediumPriority: 中优先级线程完成，释放CPU资源
所有线程执行完毕
```

中等优先级的任务早于高优先级开始执行。

```
import threading
import time

# 假设的“资源”
resource_lock = threading.Lock()

# 线程类
class PriorityThread(threading.Thread):
    def __init__(self, name, priority):
        super().__init__(name=name)
        self.priority = priority

    def run(self):
        if self.priority == 'high':
            self.high_priority_work()
        elif self.priority == 'medium':
            self.medium_priority_work()
        elif self.priority == 'low':
            self.low_priority_work()

    def high_priority_work(self):
        print(f"{self.name}: 需要资源来执行高优先级任务")
        with resource_lock:
            print(f"{self.name}: 获取到资源，开始执行高优先级任务")
            time.sleep(1)  # 模拟任务执行时间
            print(f"{self.name}: 高优先级任务完成")

    def medium_priority_work(self):
        # 模拟中优先级线程占用CPU资源
        print(f"{self.name}: 中优先级线程开始执行，占用CPU资源")
        time.sleep(3)  # 模拟长时间运行
        print(f"{self.name}: 中优先级线程完成，释放CPU资源")

    def low_priority_work(self):
        # 低优先级线程先获取资源并持有，模拟长时间占用
        print(f"{self.name}: 低优先级线程获取资源")
        with resource_lock:
            print(f"{self.name}: 低优先级线程持有资源并开始工作")
            time.sleep(2)  # 模拟任务执行时间
            print(f"{self.name}: 低优先级线程完成工作，释放资源")

# 创建线程
```

```
# 创建线程
low_priority_thread = PriorityThread('LowPriority', 'low')
medium_priority_thread = PriorityThread('MediumPriority', 'medium')
high_priority_thread = PriorityThread('HighPriority', 'high')

# 启动线程
low_priority_thread.start()
time.sleep(0.1)  # 确保低优先级线程先启动
medium_priority_thread.start()
time.sleep(0.1)  # 确保中优先级线程在高优先级线程之前启动
high_priority_thread.start()

# 等待所有线程完成
low_priority_thread.join()
medium_priority_thread.join()
high_priority_thread.join()

print("所有线程执行完毕")
```