CSE 312
OPERATING SYSTEMS

HOMEWORK 2 REPORT

Name: Muharrem Ozan
Surname: Yeşiller
Student ID: 17044033

Problem Definition

SpimOS is an operating system that represents the MIPS emulator. This operating system simulates code as if running an assembly file on MIPS. Since there is no scheduling in this system, it does not support mechanisms such as multithreading or multiprocessing. There is no interrupt mechanism. Our problem is to bring multi-threading support to this operating system. For this, we need to implement some POSIX system calls.

System Design

Two microkernels have been implemented for this system. Assembly language is used. These two microkernels launch a process called "init" and give their id as a parameter. The init process, on the other hand, goes to the relevant label, whichever microkernel has raised itself, and the system starts working.

If kernel1 raises the init process, a multi-threading array of 20 elements is sorted with a merge sort operation. Implemented assembly program: https://www.geeksforgeeks.org/merge-sort-using-multi-threading/ was inspired by the source code.

If the init process is raised by kernel2, a producer consumer simulation runs. Here, mutexes are used to block each other because they access the same source. (There is no buffer in this part, only the producer and consumer write their transactions to the terminal).

There will be a race condition between them. If anyone tries to get the locked mutex, a note will be left on the terminal. In this note, the resource is locked, the current thread is put on hold, if it could enter, a race condition between them would be printed.

Representing System Actors

The system has 4 actors.

1) Threads: It is a class that keeps the running status in the thread, the program counter, its name, its name, and the stack to it for memory.

```
class Thread
{
  private:
     Thread_State_t state;
     mem_addr program_counter;
     std::string name;
     int ID;

  public:
     mem_word R[R_LENGTH];
     Thread();
     Thread(std::string thread_name);
     Thread(const Thread & copy);
     const Thread & operator =(const Thread & rvalue);
     void setState(const Thread_State_t stat);
     void setPC(const mem_addr& program_counter);
     void setName(const std::string name);
     void setStack(const mem_word R[R_LENGTH]);
     void setID(const int ID);
     Thread_State_t getState() const;
     const mem_addr& getPC() const;
     std::string getName() const;
     int getID() const;
    void toString() const;
};
```

2) Thread State: This is an enum. Threading represents its current state. Is it working or ready to work or is it dead?

```
enum Thread_State_t
{
  READY,
  RUN,
  DEAD
};
```

3) Mutex: The Mutex class, on the other hand, is a mechanism designed to provide synchronization for threads that want to access the same resource. It contains the locked status and the ID of the unlocked area. Details will be announced.

```
class PThread_Mutex
{
  private:
     Lock_State_t has_own;
     int who;
  public:
     PThread_Mutex();
     bool lock(const Thread & who);
     bool unlock(const Thread& who);
     int getOwner() const;};
```

4) Lock State: lock status represents whether the mutex is locked.

```
enum Lock_State_t
{
  LOCKED,
  UNLOCKED
};
```

Scheduler and Context Switch

Scheduler

In the scheduling part, the round robin scheduling algorithm is used. A SPIM_timerHandler() method will run every 100ms.

If the kernel has not stood up before, it will return.

If there is only one thread running in the system, which means only the main thread, then there will be no context switch again.

If there are many threads in the system but all of them are dead (joined), if only the main thread is up, there will be no context switch again.

If the method does not enter these states, the current threading program counter and state in the current data structure is updated and a context_switch() method is called, where all backups are made.

Note: Deque is used as the data structure. It is both dynamic and has access from the front and back. In constant time (O(1)), all add-subtract switch operations occur.

Context Switch

In this section, the relevant messages are printed on the screen and the switch operation is performed. First, every member of the current thread is backed up. And the next thread is selected. If the new thread state is not DEAD, it is placed at the beginning of the data structure, and the old thread moves to the end of the data structure. And a new thread is assigned to the current_thread object that I have processed.

And all the backed up register and program counter of the system are updated as the register and program counter of the new thread.

POSIX System Call Implementations

_pthread_create(): When this system call wrapper is called by assembly, a new thread object is created. It writes its own id to REG_V0 as a return. Because the main thread should register this id. It will use this id as a reference to join this thread.

_pthread_join(int thread_id): If this system call wrapper works, some checks are made.
If the kernel is not up, the system shuts down. If the main thread tries to join itself, I evaluate it as undefined and print a kernel panic message and shut down the system. If it passes these checks, it will return if the thread_id matches the status of the thread DEAD, otherwise it will check it again by doing PC = PC - BYTES_PER_WORD (it does this for 100 ms).

Note: PC -= BYTES_PER_WORD could be made to have a context switch, but this would be a move against the round robin scheduling philosophy.

PROCESS_EXIT(int param): It outputs according to the parameter it receives. If it gets -1, a message is written to STDERR and an unsuccessful termination occurs. If it does not get -1, the status of all threads is DEAD and the system shuts down.

_pthread_exit(int thread_id): If this system call wrapper is called and the kernel is not up, the system shuts down. (Because only our kernels serve this operating system as multi-threading). If it has passed these checks, the status of the relevant thread is updated as DEAD.

_pthread_mutex_lock(): First of all, PThread_Mutex::lock(const Thread& who) this system call fires and returns true if the mutex is not locked, giving the thread the lock. But if the mutex has an owner, it returns false, so the mutex cannot be locked.

If the value we get from lock(const Thread& who) in _pthread_mutex_lock() is false, a message is suppressed. This message:
It's like "Mutex has already locked..." + "Thread " << current_thread.getID() + " waiting to unlock mutex...".

And likewise here is the message:
"Microkernel 2 Race condition:
Note that: if mutex was not used it would be race condition. There would be corruption in the buffer before the process is finished or incomplete."

The point is this: If he tried to enter a locked mutex (IF MUTEX HAS NOT BEEN USED IT WOULD HAVE ENTERED), it would corrupt the entire system of a resource that had previously entered the same critical area but was interrupted before it was finished, and there would be a race condition between them.

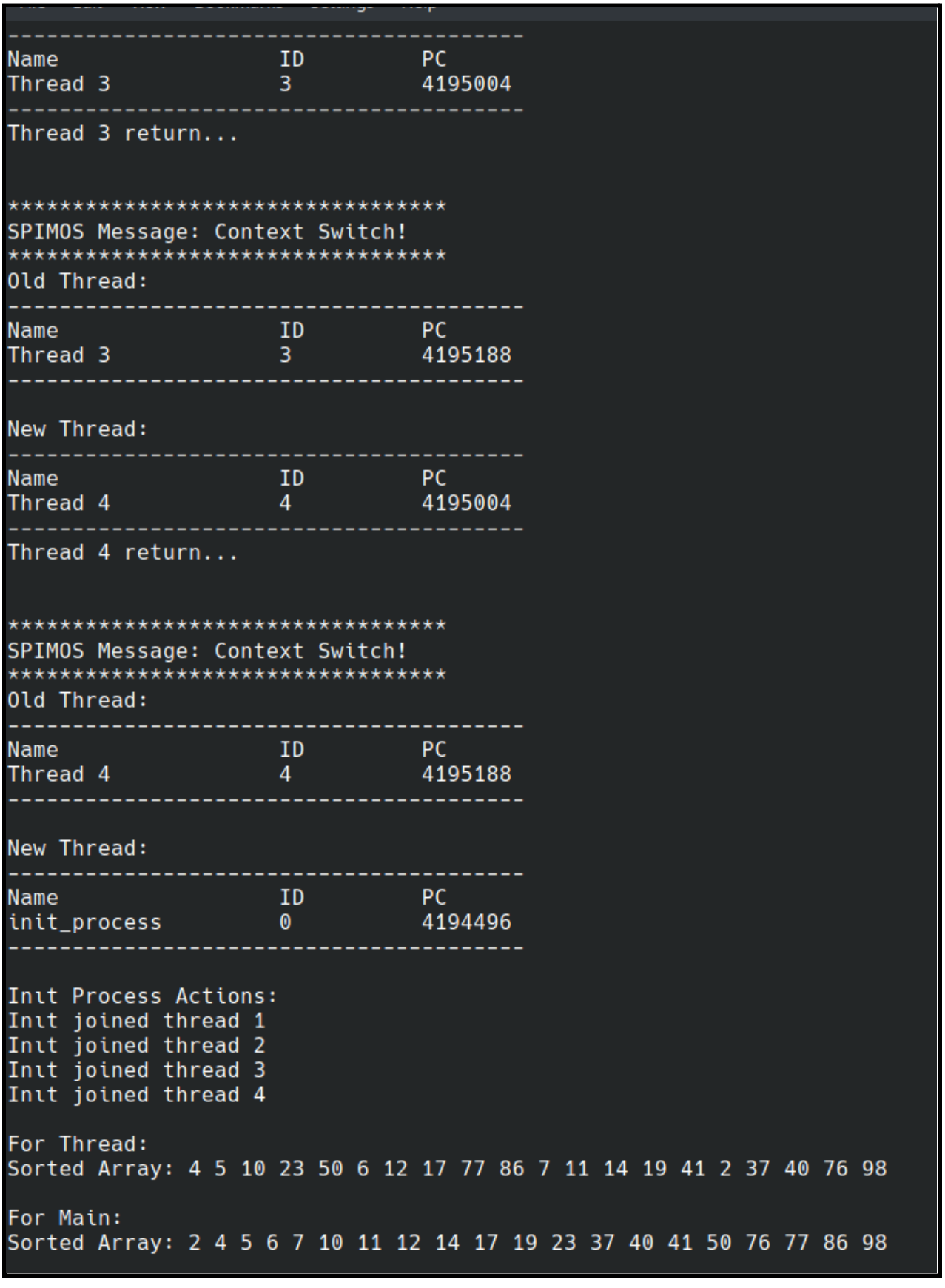
Now, in line with this information, _pthread_mutex_lock() returns a bool if bo is instruction command with PC = PC - BYTES_PER_WORD and does the same thing again.
Here again, the context switch could be thrown without waiting 100ms, but this is against the principle of round robin.

_pthread_mutex_unlock(): Here, the algorithm is simple: if the mutex is locked and the thread that took the lock calls this method, the lock is unlocked and the mutex is updated. However, if the thread is trying to unlock a mutex that does not belong to it, the system gives a kernel panic message as an undefined behavior as in the real POSIX API and shuts down.

EXTRA

System call to pthread return: When threads return, the operating system must understand this and DEAD the status of that thread. Threads also return with a system call, according to my design. The system recognizes that the thread has returned and updates it to DEAD. context_switch could be done, but it may not exceed 100ms during this time, which is against round robin scheduling. That's why there are dummy loops on the assembly side. When the operating system has already interrupted the thread in that loop, it will not run it again.

Some Pictures
For SPIM_OS_GTU1.s:



```
-----------------------------------------
Name                    ID          PC
Thread 3                3           4195004
-----------------------------------------
Thread 3 return...


**********************************
SPIMOS Message: Context Switch!
**********************************
Old Thread:
-----------------------------------------
Name                    ID          PC
Thread 3                3           4195188
-----------------------------------------

New Thread:
-----------------------------------------
Name                    ID          PC
Thread 4                4           4195004
-----------------------------------------
Thread 4 return...


**********************************
SPIMOS Message: Context Switch!
**********************************
Old Thread:
-----------------------------------------
Name                    ID          PC
Thread 4                4           4195188
-----------------------------------------

New Thread:
-----------------------------------------
Name                    ID          PC
init_process            0           4194496
-----------------------------------------

Init Process Actions:
Init joined thread 1
Init joined thread 2
Init joined thread 3
Init joined thread 4

For Thread:
Sorted Array: 4 5 10 23 50 6 12 17 77 86 7 11 14 19 41 2 37 40 76 98

For Main:
Sorted Array: 2 4 5 6 7 10 11 12 14 17 19 23 37 40 41 50 76 77 86 98
```

For SPIMOS_GTU_2.s (Race Condition)

```
Thread 1 take and lock mutex...
Producer produced 26 product
Thread 1 bring and unlock mutex...

Producer Thread Actions:
Thread 1 take and lock mutex...
Producer produced 27 product
Thread 1 bring and unlock mutex...

Producer Thread Actions:
Thread 1 take and lock mutex...
Producer produced 28
**********************************
SPIMOS Message: Context Switch!
**********************************
Old Thread:
----------------------------------------
Name                 ID         PC
Thread 1             1          4194756
----------------------------------------

New Thread:
----------------------------------------
Name                 ID         PC
Thread 2             2          4194776
----------------------------------------

Consumer Thread Actions:
Mutex has already locked...
Thread 2 waiting to unlock mutex...

Micro kernel 2 Race condition:
Note that: if mutex was not used it would be race condition.
There would be corruptions in the buffer before the process is finished or incomplete.

**********************************
SPIMOS Message: Context Switch!
**********************************
Old Thread:
----------------------------------------
Name                 ID         PC
Thread 2             2          4194816
----------------------------------------

New Thread:
----------------------------------------
Name                 ID         PC
init_process         0          4194612
----------------------------------------

Init Process Actions:
```