School of Computer Engineering

Operating Systems

Dr.Entezari

Deadlock Detection in XV6

Ali Momen

Start Date 13/10/1403

Due 6/11/1403

Contents

1 Project Introduction

As you have seen in the class, Deadlock is any situation in which no process(or thread) can proceed because there exists a circular wait. Our aim in this project is to design a scheme in Xv6 Kernel so that we can detect deadlocks among **threads** of a user process. The structure of the rest of the document is as follows:

- First we will explain the initial setup
- We will explain how to manage dynamic memory in Xv6 kernel in order to mange our resources
- We will explore how we can use a graph to find cycles and effectively Deadlocks by the mean of DFS algorithm.
- We will discuss the required System Calls
- Introduce some bonus features

2 Initial Setup

2.1 project overview

Overall, there are 3 main milestones in this project. Since the resources that we are talking about are effectively buffers in which we write to or read from, we need to allocate a kernel page and then divide it accordingly between all threads. The second phase to allocate another Kernel page so that we can store our adjacency list and other metadata for our graph there. After that, we also need to implement the System Calls for requesting and releasing the resources. First we need to understand that our aim is to detect Deadlock between **threads** and not processes. Since Xv6 is not originally Multi-threaded we need to use a modified version. Hence, for your convenience we have implemented Multi-threaded Xv6 and showed you the parts you should complete. You can witness an example of a Multi-threaded code below:

```
#include"types.h"
#include"user.h"
#include"stat.h"
Lock My_Lock;
void function(void* arg1,void* arg2){
    int* X=(int*)arg2;
    Lock_Acquire(&My_Lock);
    printf(2,"Thread %d Finished with value =%d\n",(*X),2*(*X)+1);
    Lock_Release(&My_Lock);
    exit();
}
int main(){
    void* x=0;
    int l=3;
    int* size=&l;
    int list[3];
    printf(0,"***This Program will calculate 2x+1 for 3 threads where x is the tid passed to thread as its
    Lock_Init(&My_Lock);
    for(int i=0;i<3;i++){
        list[i]=i+1;
```

```
21          thread_create(&function,(void*)size,(void*)&list[i]);
22      }
23      for(int i=1;i<=3;i++){
24          join(i);
25      }
26      exit();
27  }
28
```

All the functions used in the code above are either among the standard Xv6 libraries or implemented by us.

2.2 Repository & Required Libraries

In order to run our Xv6 code we need **QEMU**. Which is a free and open-source machine emulator and virtualizer. We will use the following command to install QEMU:

```
sudo apt-get install qemu-system-x86
```

In order to run the project you can use one of the following approaches:

```
make qemu
```

```
make qemu-nox
```

Here is the repository you can use for the start of your project:
https://github.com/merlin6990/Multi_Threaded_xv6.git

3 Resource Memory Initialization & Management

The main chunk of this part in proc.c file. First you need to allocate a kernel page using **kalloc()**. After this we will do the following: First we will divide this 4K bytes page in to two 2K halves. The upper one is for Resource structs that looks like the following:

```
typedef struct resource
{
  int resourceid;
  char name[4];
  int acquired;
  void* startaddr;
}Resource;
```

Since our Resources are buffers the code above is only for the metadata of our Resources. This struct consists of 4 main objecs. Namely:

- The ID of the resource for identifying the resources uniquely.
- The resource name
- The thread ID of the thread holding the resource

- A pointer which points to the start address of the actual buffer

We assume that we have a fixed number of resources and define it by using a macro in params.h file like the following.

```
#define NRESOURCE    4
```

Please note that you should setup this page and assign the pointers once and once when the init process is just being started.

4 Graph Management

Having completed the first part will start the second part. Since we have assumed that we have only one instance of each resource you can attest the fact that we are dealing with a **Directed bipartite graph** in which a partition is for the threads and one for the resources.

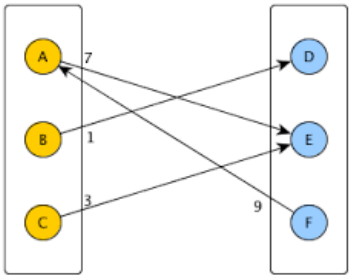

Figure 1: A Directed Bipartite Graph

In this graph an incoming edge into a resource node is a request edge for that resource and an outgoing edge means a process is holding that resource. Please note that due to our problem's constraints there exists **no more than 1** outgoing edge from our resource nodes. In this setting we can check whether there is a deadlock by using DFS to find a cycle. In order to achieve this we need to do 3 main things.

- Design how we would like to represent our graph
- Manage another kernel page for the adjacency list of this graph
- Implementing the DFS algorithm and additional functions you might need for your graph

The first part has been done for you :

```
enum nodetype {RESOURCE, PROCESS};
typedef struct Node {
  int vertex;
  enum nodetype type;
  struct Node* next;
} Node;
struct {
  struct spinlock lock;
  Node* adjList[MAXTHREAD+NRESOURCE];
  int visited[MAXTHREAD+NRESOURCE];
  int recStack[MAXTHREAD+NRESOURCE];
} Graph;
```

In fact we need another page so that the Node* pointer in Node struct points to a location in that page.

5 Required System Calls

Finally, we will test our implementation by bridging the gap between kernel and user space by the means of 2 system calls namely RequestResource and ReleaseResource. The prototype of the above mentioned functions looks like the code below:

```
int requestresource(int Resource_ID);
int releaseresource(int Resource_ID);
```

6 Bonus Section

There are a few cool features that you can add to your project. Please note that implementing one of the features is enough for the bonus section of this project.

6.1 Complete 2 additional Syscalls

In this current scheme we only have dummy resources. meaning that we do not have System Calls to be able to write to and read from these buffers. Complete the code for these System Calls in proc.c file. Right now these functions look like this:

```
int writeresource(int Resource_ID,void* buffer,int offset, int size)
{
//#################ADD Your Implementation Here#######################

//####################################################################
  return -1;
}
int readresource(int Resource_ID,int offset, int size,void* buffer)
{
//#################ADD Your Implementation Here#######################

//####################################################################
  return -1;
}
```

The description of the arguments are as follows:

- The ID of the resource we want to access
- the pointer to the start of the memory location we want to Read/Write from/to.
- the offset from the initial pointer
- size indicates how many bytes to read/write

6.2 Internal fragments

With the setting that we have describes right now you might create internal memory fragments in the page allocated for the adjacency list of your graph. Overcome this issue by periodically calling a clean function that restructures the positions of the Nodes in the memory and sets all the pointers.

6.3 Order Scheduling

With the setting that we have describes right now there is no priority or any kind of algorithm which ensures that no thread would starve. In fact without this algorithm some threads might not even get the chance to get this resource. Your job is to design this algorithm.

7 Additional Notes

- Please note that for any of the bonus features described above there needs to be a proof or a code that certifies the correctness and effectiveness of your approach
- This project is extremely challenging and requires a deep understanding of memory allocation, pointers and How threads operate.
- If you have any questions feel free to reach out to me via Telegram. Please only send your questions between 12:30 PM to 1:30 PM so that I can answer all your questions in an organized way. Due to my schedule I might not be available out of that interval.

Good Luck with your projects!!!