14.06-79 Sebastian Ostenfeldt Jensen GJX653

Datalogisk Institut

Københavns Universitet

Operating Systems and Concurrent Programming

2015

27. marts 2015

UNIVERSITY OF
COPENHAGEN

Indhold

1 P1 benchmarking: Thread-safe queues

The two given implementations use a linked list as a data structure for the queue. The main difference is that fg_queue uses fine-grained locking and cg_queue uses coarse grained locking, when adding and remove elements from the queue. When fine-grained locking is used the head and tail are locked independently. Therefore can the head and tail be accesssed at the same time. E.g. a thread can remove an element at the same time another thread adds an element. When coarse grained locking is used a global lock on the hole queue is used. This result in only one thread can accesss the queue at a time. // To benchmark the speed of the different implementations it is needed to let multiple threads insert and remove elements at the same time under different circumstances. A benchmark has been implemented in the folder benchmark in the handed in source tree. To start the benchmark one can under Linux issue the command "make benchmark". First the benchmark will measure the time it takes to insert and remove 1.000.000 elements width fine and coarse-grained locking. This is done with one thread adding elements while another thread remove elements. If the queue becomes empty before the thread that removes elements has removed 1000.000 elements the thread will make a loop and try again until all 1000.0000 elements has been removed. This could theoretically happen if the adding thread is not adding elements as fast as the removing thread consumes them. The same test is then made using 4 threads, where 2 threads are adding and 2 threads removing elements. Then we move to using 8 threads (4 adding and 4 removing). All this is then repeated with 2-, 3-, 4-, 5-, 10- millions items. The output to the console is as follow.

		time	
Threads	items	fine	coarse
2	1000000	0.41s	0.28s
4	1000000	0.46s	0.43s
8	1000000	0.71s	1.00s
2	2000000	0.70s	0.34s
4	2000000	0.84s	0.80s
8	2000000	0.79s	1.98s
2	3000000	0.96s	1.49s
4	3000000	1.17s	1.28s
8	3000000	1.18s	2.95s

2	4000000	1.27s	2.19s
4	4000000	1.41s	1.49s
8	4000000	1.54s	4.04s
2	5000000	0.89s	2.71s
4	5000000	1.67s	2.28s
8	5000000	1.73s	4.92s
2	10000000	2.89s	4.82s
4	10000000	3.49s	4.38s
8	10000000	3.51s	9.97s

Threads is the total number of threads. E.g. 4 threads means 2 threads adding and 2 threads removing. Items is the total number of threads to be inserted. So if there are 4 threads and 1.000.000 items each adding thread adds 500.000 items and each removing thread removes 500.000 items. Fine is the time used with fine-grained locking. Coarse is the time used with coarse grained locking.

As we see fine-grained locking is in all cases faster than coarse-grained. This is what we can expect as fine grained locking allow an adding and removing thread to accesss the queue at the same time. What is noticeable is that using more than 2 threads make the job slower with fine-grained locking. The test is done on a eight core CPU so there are enough cores to facilitate the threads. The reason is that no more than 2 cores can accesss the queue at the same time. So using more threads just adds overhead as nearly the only thing the threads are doing are accesssing the queues. If the threads were to do other stuff like heavy calculation in between accesssing the queues, then more than 2 threads would result in an increase in speed. We especially see a lot of overhead when using couarse-grained locking and 8 threads. Here there will most of the time be up to 7 threads waiting to accesss the queue. One other interesting finding is that using 4 threads on coarse grained locking are in some cases a little faster than using 2 threads. A conservative guess could be that when using 4 threads it will happen often that when a thread accesss the list the previous accesss to the list was to the same end. and therefore the address is still in the CPU's register. E.g. if 1 thread has just accesssed the tail and updated the tail, then if the next thread accesssing the list also accesss the tail, then the data might still be in the CPU's register. But if the previous thread that accesssed the list was accesssing the head, then there is a bigger chance that the tail's date was paged out to the memory. And with only 2 threads this might happen more often than with 4 threads.

The test can be reproduced by invoke make benchmark when in the folder benchmark on a Linux system.

2 P2. Buenos system calls for basic I/O

The syscall functionn syscall_rand should return a pseudo random number between 0 and range -1. Buenos already has kernel functionn _get_random that calculate a pseudo random number from the command line argument to buenos randomseed. The _get_rand is defined in lib/rans.S. The the syscall syscall_rand is implemented as follow.

- in proc/syscall.h the SYSCALL_RAND is defined as follow

```
/* Random number*/
#define SYSCALL_RAND 0x310
```

- in proc/syscall.c the according syscall functionn is defined to call the _get_random to let the kernel produce the number for us. This is the definition

```
case SYSCALL_RAND:
V0 = _get_rand((int) A1);
break;
```

- To give accesss to userland processes to call the new syscall we declare the functionn int syscall_rand(int n); in test/lib.h. The functionn is defined in test/lib.c and the only thing it does is to make the syscall with the right parameters, get the return value and return it to the userland process. This is the code changed in test/lib.c

```
#ifdef PROVIDE_RANDOM_NUMBER
int syscall_rand(int n)
{
        return (int)_syscall(SYSCALL_RAND, (uint32_t) n, 0, 0);
}
#endif
```

To test the new functionn a test program has been implemented which print 6 pseudo random numbers between 0 and 100. The test program be run by invoking make rand when in the buenos folder and fyams-term is running in the same directory.

see appendix P2 for a list of all code changed.

3 P3. Process communication: Unidirectional pipes

a Implement system call syscall_pipe

To allow userland processes accesss to the syscall_pipe the functionn is declared and defined in test/lib.h, test/lib.c, proc/syscall.h and proc/syscall.c. This is done the same way as in P2. See appendix P3 for the code. The core of the pipe functionnality is implemented in proc/pipe.c and proc/pipe.h. proc/pipe.h defines the pipe structure and declare the functionns for working with the pipes. The structure used for the a pipe is a circular list. It is implemented as an array of chars with a pointer to the read end and write end. The pipe has space for 100 elements which is defined in proc/pipe.h The implementation has room for three pipes but can easily be extended. The three pipes are named pipe_a, pipe_c, pipe_d and are initialized together with a corresponding semaphore in the top of proc/pipe.c. each pipe also has 2 file descriptors which is associated with writing and reading from the pipe. The file descriptors or file handlers are defined in proc/syscal.h. Here follow a detailed description of the implementation when a functionn is mentioned it is assumed it is located in proc/pipe.c if nothing else is stated.

- pipe_start

  When the syscall_pipe functionn is called, pipe_start is is called. This functionn go true the pipes until it find a pipe that is not in use. If all pipes are in use it returns -1. The way it check if a pipe is in use is tp try opening its associated semaphore. If the pipe is in use a call to usr_sem_open will return NULL as the semaphore would then already be open. When a free pipe is found, the assoiciated semaphore is opened with the state 0 indicating theres a thread in it. The file descriptor array which was supplied as a pointer is updated with the value of the pipes read and write file handlers. There by allowing userland processes accesss to the pipes file handlers. What also happens is that the pip is initialized by setting the pipes read and write end pointers to 0. Then the semaphore for the pipe is signalled so it is ready to be used. See appendix P3 for the code

b Modify existing system calls

- io_read and io_write

To let the userland processes use the pipe functionnality changes to the sytcall read and write need some modificationn. As these syscall already make use of the functionn io_write and io_read in the file proc/io.c it is convenient to let these functionns handle what happens when read and write is made to a pipe. Both for io_write and io_read there is as switch case branch which calls appropriate functionns according to the right filhandler. Therefore it was easy to just insert some more cases in io_write and io_read. E.g. was the following cases inserted in io_write

```
case FILEHANDLE_PIPE_A_WRITE:
    res = pipe_write(file, buffer, length);
    break;
case FILEHANDLE_PIPE_B_WRITE:
    res = pipe_write(file, buffer, length);
    break;
case FILEHANDLE_PIPE_C_WRITE:
    res = pipe_write(file, buffer, length);
    break;
```

Similar cases where inserted in io_read.

- pipe_read and pipe_write

  When pipe_write is called it first check what file handle it was called with. then it ask for the semaphore for the according pipe. When it get accesss it checks if the pipe is full. If its not full it write to the pipe release the semaphore and return 0. if the pipe was full it spins until the pipe is not full any more. Between each time it checks if the pipe is full the semaphore is signalled and acquired to allow a reader to get in. Nearly the same happens when pipe_read is called, except that it now read if its not empty and spins if its empty until it is not empty any more. Both functionns return -1 if the pipe is not in use. These is indicated by the associated semaphore is not open. When the functionns try to get accesss to a semaphore that is closed the functionn returns -1. See appendix P3 for full code of pipe_read and pipe_write.

- syscall_close

  Leting syscall_close work on unidirectional pipes has not been fully implemented do to time pressure. But it is relatively easy to implement. In pipe.c a functionn pipe_destroy is defined which only thing it does is to close the associated semaphore of the pipe that was given as argument. to let syscall_close work on unidirectionalpipe the functionn

in io_close in proc/io.c should be expanded with an if statement that checks if the filedescriptor given is one of the pipes filedescriptor. And if it is then call the pipe_destroy with the given pipe as argument.

- child process inherent file descriptors to pipe's

  When a fork is made the child will get a full copy of the parents address space. Therefore it automatically happens that the child inherent the file descriptors as these are pointers.

c Implement additional system support

- syscall_dup

  Again to allow userland processes access to the syscall_dup functionn it is declared and defined in test/lib.h, test/lib.c, proc/syscall.h and proc/syscall.c. This is again done the same way as in P2. syscall_dup is implemented in proc/dup.h and proc/dup.h. The way to manage what file descriptors is duplicated to what file descriptors a mapping is implemented. The mapping is quit simple. it consist of an array of int with the size that there are file descriptors in the system. In our case 9. The array is named dup_map. Each entry in the array correspond to a fildescripter. E.g. dup_map[0] is stdin. The array is initialized with the functionn dup_init which set each entry to the number as its index. This means that each filedescriptor points to it self. the dup_map_init function does this and it is called when buenos start up from the init/-main.c file. When a syscall_dup is called the value of the filedescriptor being duplicated is set in the index of the file descriptor it is being duplicated to. See appendix P3 for the code. When as syscall to io_write or io_read is called these functionns has been modified to use the filedescriptor the given file descriptor points to. eg. if stdin which has value 0 has been copied to PIPE_A_READ which has value 3, then dup_map[0] = 3 and now when we read from stdin we actually read from pipe a.

- syscall_select

  This has not been implemented yet du time pressure. But one way to implement would be to let the functionn spin and try to acquire a semaphore on the pipes one by one and when it gets in to a semaphore it has found a pipe that is ready.

For all the files dub.c, dub.h, pipe.c and pipe.h these are added to proc/-module so they are being compile.

d Test the implemented functionns

To test the implementation pipe1 has been run and is working. Only modification that is done is that we copy the fildescripter of stdout back to it self, else we were not able to print hello world to the screen after receiving it from the pipe. if we did not do this, hello world would obviously just be copied back to the pipe.

more test has been implemented in pi1.c. These test are self explaining.

The test can be run by issuing the command make pipe1 and make pipe2 when in the root directory of buenos and fyams term is running

BLANK PAGE

4 T1. Concurency:Parts must come together

a Pseudo-code

The pseudo code for the thread functionns frame and wheel are provided in the file T1.c in the folder T. It is recommendedto look at the file as it provide nice layout and syntax highlighting if using an editor that provide this for c code. We use 3 variables and a semaphore for each of them:

- int bikeInProduction: which can be either 1 or 0.
- arrayList frames: which holds which holds the produced frames
- FIFOQueye readyBikes: which is a queue where the bikes are placed when they are produced.
- When the frame thread want to create a frame it first check that bikeInProduction is 0. if it is it is allowed to start production of a frame. I does that by setting the bikeInProduction to 1 indicating that it has started the production. When the frame is produced the new frame is put in to the list of frames and it will the try to create a new frame. As mentioned the frame thread can only start production of a frame if bikeInProduction is set to 0. if it is 1 then the frame thread will keep spinning and checking if it change to 0.
- The whell thread will only start production of wheel's if there is a bike in production, which means that bikeInProduction is set to. If it is one 1 the wheel thread will set it to 0 and start producing 2 wheels. When the wheels are produced it will try to get the frame from the frames list where the frame thread is suposed to put it. If the frame thread has not finished the frame jet the whell thread will keep spinning until the frame thats in production are added to the frames list.
- Then it gets the frame and assemble the bike and put it in the readyBikes queue. This queue can be seen as the conection with the outside world where the bikes can be collected from.
- The wheel thread will then try to produce more wheels.
- When ever the wheel thread want to start production of wheels it check to see if bikeInProduction is 1. if its 0 then the thread keeps checking until it becomes 1 before it start the production.

- In this way the frame thread will never start production of a frame unless the whell thread is ready or in production of wheels for the last created frame. As well will the wheel thread never start production of wheels for which a frame is not produced or about to be produced.

Here follow the code of the two threads.

```
int bikeInproduction = 0;
sem mutex1 = 1;

arrayList frames;
sem mutex2 = 1;

FIFOqueue readyBikes
sem mutex3 = 1;

void frame() {
    /*produce frames*/
    while(1){
        P(mutex1);
        if(bikeInProduction){
            bikeInProduction++;
            V(mutex1);
            frame frame = produceFrame());
            P(mutex2);
            frames.add(frame);
            V(mutex2)
        } else {
            V(mutex1)
        }
    }
}

void wheel() {
    /*produce whells*/
    while(1){
        P(mutex1);
        if(bikeInProduction == 1){
            /*there is a frame in productio so lets make 2 wheels*/
            bikeInProduktion-- /*we make wheels allow more frames to be produced
            V(mutex1);
```

```
            wheel w1 = produceWheel();
            wheel w2 = produceWheel();
            P(mutex2);
            while(frames.size == 0){ //spin until there is a frame
                V(mutex2);
                sleep 1 sec;
                P(mutex2)
            }
            /*If we are here there must be a a ready frame, lets assemble*/
            frame frame = frames.remove(frame.size); /*remove the frame and retu
            V(mutex2); /*we got the frame so release the lock*/
            bike bike = assemble(frame, w1, w2); //made a bike

            /*add the bike to the pile of finished bikes*/
            P(mutex3)
            readyBikes.add(bike);
            V(mutex3)
            else {
                V(mutex2);
        } else {
            /*we are waiting for a frame to get in production*/
            V(mutex1);
        }
    }
}

int main(){
    pthread44create(frameProducer);
    pthread_create(wheelProducer);
}
```

b Correctness

Whit the current implementation it is trivial that only usable bikes will be produced as the wheel thread will create 2 wheels for each frame that goes in to production as well the frame thread will newer start producing more frames if the wheel thread has not indicated that it will produce frames for the last created frame. Therefore the 2 threads will wait for each other so we are not ending up with to many frames or wheels. It is trivial that the solution is dead lock free as there are no place in the code where a thread holds a

lock while it wait for the other thread. In all the cases where a thread waits for the other thread it is spinning in a wild loop and continuously acquiring and leleasing the semaphore on the variable it waiting on to be changed. Therefore the other thread will eventually get accesss in to the semaphore.

5 T2.Online algorithms:Binery body system

a Implementation details

The pseudo code of the two functionns split and coaleace can be found in the file T2.c in the folder T. It is recommendedto look at that file instead as it provides a nicer layout and syntax highlighting if using an editor that provides this for c code. The functionn split is implemented as follow.

- First it get the log size of the node and if it is 1, then the note can not be split any further and the functionn return NULL
- Then if the functionn can be split we get to the else branch. Here we first calculate the log size of the notes by subtracting 1 from the current log size.
- We then create a new node named node2. We then let node2 next point to the old node's next and node2 previous point to the old node.
- Now is only left to let the old node's next point to node2.
- We then return node2, and we are done

Here follow the code for the split functionn

```
struct free_list_node* split(struct free_list_node* node){
    unsigned char old_block_log_size = node->block_descr->log_size;
    if(old_block_size == 1) { // the node can not be split any further
        return NULL;
    }
    else { //split the node
        unsigned char new_block_log_size = old_block_log_size/2;
        free_list_node node2;
        node2->next = node->next; // point to old node's next
        node2->previous = node;   //point to old node
        node->next = node2;       //point to new node
        return node1->next;
    }
}
```

For the functionn coalesce we need to determine the address of the body to the given node. As the body is just before or after the node we can determine the address of the body by inverting log $size^{th}$ bit of the node's address.

Here is an example where block 1 and 2 has been split into two blocks of log size 2. That means we need to invert the 2^{th} bit of the address of the block we wants it body from. The address of block 1 is 000, inverting 2^{th} bit gives 010 which is block 3's address. inverting it again gives block 1

```
0 0 0 |     Block 1
0 0 1 |
0 1 0 *     Block 2
0 1 1 *
1 0 0 |     Block 3
1 0 1 |
1 1 0 |
1 1 1 |
```

With these in place we are now ready to implement the functionn coalesce.

- First we store the provided node's log size in a var named log_size
- Then we create a number n which a binary number with the only the log_size^{th} bit set to 1. to create the number we use c's build in function to shift the number 1 $\log_sizetimestotheleft.ThenweXORtheaddressofthenodewithnandwege$

- We then need to determine which node is first in the list. We do this by checking if the body's next point to the node. if it does the body is first in the list else it was the node that was first. We then store the node and body to the variables old_first and $old_lastaccordingtoourfindings.Thenwecancoalescethenodes.Firstwecreatear$ $1anditsnexttoold\_last'snextanditspreviousstoold_first'sprevious.$

- It is trivial that the two functionns run in O(1) worst case as there are no loops and it does not call any functionns that has higher running time than O(1). Here follow the code of the function coalesce.

```
struct free_list_node* coalesce(struct free_list_node* node){
    log_size = node->block_descr->log_size

    //get address of the body
    //we can get the address of the body by fliping the bit of node's address
    // at position log_size
    unsigned char n = 1 << log_size; //make an n for the XOR operation
    free_list_node node_body = n^node           //XOR n with node which return


    //check if body is free
```

```
    if(!(node_body->block_descr->in_use)){ //its not in use we can free it
        free_list_node old_first:
        free_list_node old_last;

        //we need to determine which node is first in the list
        if(node_body->next == node){    //body is before node in list
            old_first = node_body;
            old_last = node;
        }
        else {                           //node is before body in list
            old_first = node;
            old_last = node_body;
        }

        //now we coalesce the nodes
        free_list_node return_node;
        return_node->block_descr->in_use = 0;               //set the new node t
        return_node->block_descr->log_size = log_size + 1;  //set the new log_si
        return_node->next = old_last->next;                 //set the new next t
        return_node->previous = old_first->previous;        //set the new previo
        return return_node;
    }
    return NULL; // body was not free return NULL
}
```

b Runtime analysis

- Malloc

  When malloc is called on the binary body system it iterates true the free list until a block that is big enough is found. If it is to big it will split the node. splitting the node is only O(1) and iterating the list is O(n) therefore a call to malloc is in worst case O(n) because it might need to iterate the hole list.

- Free

  Free works like this: Check if the body is free. if it is the coalesce with the body. this result in a new free block. with this block we also have to check if this's block's body is free. If it is free then coalesce with the body. This could continue all true the hole list of memory resulting in a worst case of O(n). Its ofcoarse not so likely that this will happen. But in worst case it is O(n)

c Fragmentation analysis, NOTE: for this part examV1.1 was used

The size of the body systems arena need to be big enough to hold the memerory when the blocks in use is devided in the most insufficient way. If it is big enough malloc want break. There exist a lemma that describes this upper bound as follow. See reference[1]

- Let M be the maximum amount of memory space used by a program and n be the smallest allocation blocksize. If T is the total amount of memory storage in an insufficient scenario then

$$T <= \frac{M^2}{4n} + \frac{M}{2} - \frac{3n}{4}$$

  for all allocators

As well there exist a theorem for the lower bound (also see reference [1]) which is as follow.

- Let M be the maximum amount of memory space used by the program and n be the smallest allocation block size. If S is the minimum amount of memory storage sufficient for all allocators, then

$$S <= \frac{M^2}{4n} + \frac{M}{2} - \frac{n}{4}$$

M is the maximum allocated memory at any time which correspond to the M in the examtext. n is the minimum allocated block size which correspond to k in the examtext. S Will then be the lower bound for the arena and T the upper bound.

6 T3.Protocols: The fundamentals

NAND memory, wear leveling

Nut much has been mentioned in Dinosaur Book about wear out of NAND memory. But it is an interesting subject and highly associated with the increasing use of NAND as storage. NAND memory is most often the type of memory used by flash cards, usb storage and ssd disks. One big problem with NAND is that it wears out after a certain number of reads and writes. This is not a big problem when the NAND is used for long time storage (up to some few rewrites a day). E.g when they are used in cameras or as storage on a usb stick the writes and reads are normally distributed out over the hole storage and the are not continuously many writes. But the wear out becomes a problem when the NAND memory is uses for running operating systems from as is the case when using NAND memory in embedded systems and as primary storage in a computer. The reason is that an operating system often make many small writes to the disk. With ssd disks this has been solved by having a controller build in to the ssd disk that distribute the writes over the disk while it keeps track of how many writes each block have had. This is called wear leveling. The simple form of wear leveling just writes to the free block that has least writes. This is called static wear leveling. A more advanced wear leveling technique is dynamic wear leveling that moves data around the disk avoiding a part of the disk to not get rewrite to often while other parts do. These advanced wear leveling controllers also consider reads as they also has an effect on the wear out.

But with cheap NAND memory like flash cards there are often only a very simple storage allocater implemented. A protocol for implementing wear leveling on a storage device operated by the operating system would be convenient. Here follow a description of a simple wear leveling protocol/algorithm to implement static wear leveling on a storage. We only consider the writes

- The memory storage need to have an index that keep track of each block's number of writes. To keep it simple for the NAND controller this is kept as an array with as many entrances as there are blocks (on most flash cards the block size is either 512kb or 4M so this does not take much space).
- When the operating system initialize the information about number of writes is copied in to the operating systems memory as a minimum priority queue. A more efficient way would be to use some kind of hash map together whit more priority queues. But lets keep it simple.

- when the operating system writes to the NAND memory it chose the first block in the priority queue which is the block whit least writes. When the operating system receives a syscall halt() it will the copy the information about number of writes back to the NAND memory.
- The reason the information about number writes is stored directly to the NAND memory and not in the operating systems files are that it should be accessible from other systems if the card is moved to a new system. Therefore the important part is that all systems that would use this kind of system agree on a protocol for how and were to store the information on the NAND memory
- The purposed way is very simple and can easily be optimized with dynamic wear leveling and not necessarily writing data to the least used block but also predict how often the date being written will be accessed. E.g. pictures might not get accessed as often as log-files. An other important part to mention is that flash cards and usb sticks don't allow direct access to the memory and therefore it can not be done with the these storage devices before the producers allow direct access to the memory. But with raw NAND memory this can be achived.

7 References

[1] IEEE TRANSACTIONS ON COMPUTERS,VOL. 59,NO. 4,APRIL 2010
Upper Bounds for Dynamic Memory Allocation
Yusuf Hasan, Wei-Mei Chen, Member, IEEE, J. Morris Chang,
Senior Member, IEEE, and Bashar M. Gharaibeh.
http://www.ece.iastate.edu/ morris/papers/10/ieeetc10.pdf
Page 471

8 Appendix P2 code changes

syscall.h

```
/* Random number*/
#define SYSCALL_RAND 0x310
```

syscall.c

```
case SYSCALL_RAND:
V0 = _get_rand((int) A1);
break;
```

test/lib.h

```
int syscall_rand(int n);
```

test/lib.c

```
#ifdef PROVIDE_RANDOM_NUMBER
int syscall_rand(int n)
{
        return (int)_syscall(SYSCALL_RAND, (uint32_t) n, 0, 0);
}
#endif
```

9 Appendix P3 code changes

proc/syscall.h

```
/* Random number*/
#define SYSCALL_RAND 0x310

/* Pipe */
#define SYSCALL_PIPE 0x311
#define SYSCALL_PIPE_DESTROY 0x312

/* DUP */
#define SYSCALL_DUP 0x313

/* Console file handles. */
#define FILEHANDLE_STDIN    0
#define FILEHANDLE_STDOUT   1
#define FILEHANDLE_STDERR   2

/* pipe file descriptors*/
#define FILEHANDLE_PIPE_A_READ 3
#define FILEHANDLE_PIPE_A_WRITE 4
#define FILEHANDLE_PIPE_B_READ 5
#define FILEHANDLE_PIPE_B_WRITE 6
#define FILEHANDLE_PIPE_C_READ 7
#define FILEHANDLE_PIPE_C_WRITE 8
#endif
```

proc/syscall.c

```
  case SYSCALL_RAND:
    V0 = _get_rand((int) A1);
    break;
  case SYSCALL_PIPE:
    V0 = pipe_start((int*) A1);
    break;
  case SYSCALL_PIPE_DESTROY:
    V0 = pipe_destroy((int*) A1);
    break;
  case SYSCALL_DUP:
```

```
    V0 = dup((int*) A1);
    break;
```

proc/pipe.h

```
#ifndef BUENOS_PIPE
#define BUENOS_PIPE
#define PIPE_SIZE 100
#define P_SIZE (PIPE_SIZE + 1)

typedef struct pipe_t_struct{
  char *usr_sem;
  int read;
  int write;
  char p[P_SIZE][256];
} pipe_t;

int pipe_init(pipe_t *pipe);
int pipe_write(int file, char* buf, int length);
int pipe_read(int file, char* buf, int length);
int pipe_destroy(int *pipe);

int pipe_start(int *fds);

#endif
```

proc/pipe.c

```
#include "drivers/gcd.h"
#include "fs/vfs.h"
#include "kernel/assert.h"
#include "proc/syscall.h"
#include "lib/debug.h"
#include "proc/pipe.h"
#include "proc/usr_sem.h"
#include <stdarg.h>
#include <stddef.h>
#include "proc/usr_sem.h"
#include "lib/debug.h"
#include "lib/libc.h"
```

```
pipe_t pipe_a;
pipe_t pipe_b;
pipe_t pipe_c;
usr_sem_t *pipe_a_sem;
usr_sem_t *pipe_b_sem;
usr_sem_t *pipe_c_sem;

/**
 * Write to the write end of the pipe.
 */
int pipe_write(int file, char* buf, int length) {
    pipe_t *pipe;
    switch(file) {
        case FILEHANDLE_PIPE_A_WRITE:
            pipe = &pipe_a;
            break;
        case FILEHANDLE_PIPE_B_WRITE:
            pipe = &pipe_b;
            break;
        case FILEHANDLE_PIPE_C_WRITE:
            pipe = &pipe_c;
            break;
        default:
            return -1;
    }
    /* Add the new item to the write. */
    int ret = usr_sem_p((*pipe).usr_sem);
    if(ret < 0){
       return -1;
    }
    while(pipe->write == (( pipe->read - 1 + P_SIZE) % P_SIZE)) /* Pipe is Full*
    {
        usr_sem_v((*pipe).usr_sem);
        usr_sem_p((*pipe).usr_sem);
    }
    stringcopy(pipe->p[pipe->write], buf, length);
    pipe->write = (pipe->write + 1) % P_SIZE;
    usr_sem_v((*pipe).usr_sem);

    return 0;
}
```

```
/**
 * read from the read end.
 *
 * Returns: a pointer to the value of the node at the read end of the pipe; NULL
 * if the pipe is empty, or an error occurred.
 */
int pipe_read(int file, char* buf, int length) {
    pipe_t *pipe;
    DEBUG("debuginit", "pipe.c read 1\n");
//    kprintf("file %d\n", file);
    switch(file) {
        case FILEHANDLE_PIPE_A_READ:
            pipe = &pipe_a;
            break;
        case FILEHANDLE_PIPE_B_READ:
            pipe = &pipe_b;
            break;
         case FILEHANDLE_PIPE_C_READ:
            pipe = &pipe_c;
            break;
         default:
            return -1;
    }

    int ret = usr_sem_p((*pipe).usr_sem);
    if(ret < 0){
        return -1;
    }

    DEBUG("debuginit", "pipe.c read 1\n");

    while(pipe->write == pipe->read)         /* Pipe is Empty*/
    {
        usr_sem_v((*pipe).usr_sem);
        usr_sem_p((*pipe).usr_sem);
    }

    stringcopy(buf, pipe->p[pipe->read], length);
    pipe->read = (pipe->read +1) % P_SIZE;
    usr_sem_v((*pipe).usr_sem);
```

```
  return length;
}

/**
 * Destroy the pipe after use.
 *
 * put and get must not be called after destroy. Unless init has been called
 * again.
 */
int pipe_destroy(int *pipe) {
    switch(*pipe){
    case 3:
        return usr_sem_destroy(pipe_a_sem);
    case 5:
        return usr_sem_destroy(pipe_b_sem);
    case 7:
        return usr_sem_destroy(pipe_c_sem);
    default:
        return -10;
    }
}

int pipe_start(int *fds) {
    //find a free pipe
    if((pipe_a_sem = usr_sem_open("pipe_a_sem", 0))){
       pipe_a.read = 0;
       pipe_a.write = 0;
       pipe_a.usr_sem = pipe_a_sem;
       *fds = FILEHANDLE_PIPE_A_READ;
       *(fds + 1) = FILEHANDLE_PIPE_A_WRITE;
       usr_sem_v(pipe_a_sem);
       DEBUG("debuginit", "proc/pipe.c pipe_init sem closed\n");

    }
    else{
        if((pipe_b_sem = usr_sem_open("pipe_b_sem", 0))){
           DEBUG("debuginit", "proc/pipe.c pipe_init sem b opened\n");
           pipe_b.read = 0;
           pipe_b.write = 0;
           pipe_b.usr_sem = pipe_b_sem;
           *fds = FILEHANDLE_PIPE_B_READ;
```

```
            *(fds + 1) = FILEHANDLE_PIPE_B_WRITE;
            usr_sem_v(pipe_b_sem);
            DEBUG("debuginit", "proc/pipe.c pipe_init sem b closed\n");
        }
        else {
            if((pipe_c_sem = usr_sem_open("pipe_c_sem", 0))){
               DEBUG("debuginit", "proc/pipe.c pipe_init sem b opened\n");
               pipe_c.read = 0;
               pipe_c.write = 0;
               pipe_c.usr_sem = pipe_c_sem;
               *fds = FILEHANDLE_PIPE_C_READ;
               *(fds + 1) = FILEHANDLE_PIPE_C_WRITE;
               usr_sem_v(pipe_c_sem);
               DEBUG("debuginit", "proc/pipe.c pipe_init sem b closed\n");
            }
            else {
                DEBUG("debuginit", "proc/pipe.c pipe_init return -1\n");

                return -1;
            }
        }
    }
    return 0;
}
```

proc/dup.h

```
#ifndef BUENOS_PROC_DUP
#define BUENOS_PROC_DUP

#include "kernel/config.h"
#include "lib/types.h"

#define DUP_ENTRIES 9

int dup(int *fd);
void dup_map_init();
int get_fd(int fd);
```

```
#endif
```

proc/dup.c

```
#include "proc/dup.h"
#include "proc/process.h"
#include "proc/elf.h"
#include "kernel/thread.h"
#include "kernel/assert.h"
#include "kernel/interrupt.h"
#include "kernel/sleepq.h"
#include "kernel/config.h"
#include "fs/vfs.h"
#include "drivers/yams.h"
#include "vm/vm.h"
#include "vm/pagepool.h"
#include "lib/types.h"
#include "lib/debug.h"
/**
 * This module contains controls the mapping of file descriptors
 *
 */
int dup_map[DUP_ENTRIES];

/* We need a spinlock to lock accessses to the dup map. */
spinlock_t dup_map_table_slock;

int dup(int *fd)
{
    dup_map[*(fd +1)] = *fd;
    return 0;
}

void dup_map_init()
{
    for(int i = 0; i < DUP_ENTRIES +1; i++){
        dup_map[i] = i;
    }
}
```

```
int get_fd(int fd){
    return dup_map[fd];

}
```

proc/io.h

```
int io_read(openfile_t file, void* buffer, int length)
{
  int res;
  //check the dup mapping to se if another fd should be used
  file = get_fd(file);
  switch(file) {
  case FILEHANDLE_STDIN:
    res = tty_read(buffer, length);
    break;

  case FILEHANDLE_STDOUT:
  case FILEHANDLE_STDERR:
    res = VFS_INVALID_PARAMS;
    break;

  case FILEHANDLE_PIPE_A_READ:
    res = pipe_read(file, buffer, length);
    break;

  case FILEHANDLE_PIPE_B_READ:
    res = pipe_read(file, buffer, length);
    break;

  case FILEHANDLE_PIPE_C_READ:
    res = pipe_read(file, buffer, length);
    break;

  default:
    file -= 3;
    if (!process_has_open_file(file)) {
      res = VFS_NOT_OPEN_IN_PROCESS;
    } else {
```

```
      res = vfs_read(file, buffer, length);
    }
  }

  return res;
}

int io_write(int file, void* buffer, int length)
{
  int res;
  //check the dup mapping to se if another fd should be used
  file = get_fd(file);
  switch(file) {
  case FILEHANDLE_STDIN:
    res = VFS_INVALID_PARAMS;
    break;

  case FILEHANDLE_STDOUT:
    res = tty_write_stdout(buffer, length);
    break;

  case FILEHANDLE_STDERR:
    res = tty_write_stderr(buffer, length);
    break;

  case FILEHANDLE_PIPE_A_WRITE:
    res = pipe_write(file, buffer, length);
    break;
  case FILEHANDLE_PIPE_B_WRITE:
    res = pipe_write(file, buffer, length);
    break;
  case FILEHANDLE_PIPE_C_WRITE:
    res = pipe_write(file, buffer, length);
    break;

  default:
    file -= 3;
    if (!process_has_open_file(file)) {
      res = VFS_NOT_OPEN_IN_PROCESS;
    } else {
      res = vfs_write(file, buffer, length);
```

```
    }
  }

  return res;
}
```

proc/module.mk

```
# Makefile for the kernel module

# Set the module name
MODULE := proc


FILES := exception.c elf.c process.c syscall.c usr_sem.c io.c pipe.c dup.c

SRC += $(patsubst %, $(MODULE)/%, $(FILES))
```

init.main

```
  /*initialise the fildescriptor ma*/
  kwrite("Initializing filedescriptor map\n");
  dup_map_init();
```