**OSD PROJECT DOCUMENT**

There are three lists freelist(N blocks of storage), list1, list2. These three lists are shared among three processes p1, p2, p3. As each of the processes have their own memory space any resource that need to be shared among the processes should be in the shared memory between the processes. That is freelist, list1, list2 should be there in the shared memory.

We created posix shared memory and created lists free lists in the shared memory.

**Shared memory:**

static int init\_shm(const int is\_child){

//create shared environment

shm\_unlink(SHM\_NAME); //unlink shm, if program crashed

shmid = shm\_open(SHM\_NAME, O\_CREAT | O\_EXCL | O\_RDWR, 0666);

if(shmid == -1){

perror("shm\_open");

return 1;

}

//resize the shared region to take on storage structure

if(ftruncate(shmid, sizeof(struct memoryStorage)) == -1){

perror("ftruncate");

return 1;

}

//map it to our memory space, which is inherited by all children

mem = (struct memoryStorage \*) mmap(NULL, sizeof(struct memoryStorage), PROT\_READ | PROT\_WRITE, MAP\_SHARED, shmid, 0);

if(mem == (void\*) -1){

perror("mmap");

return 1;//error

}

return 0; //success

}

The above code snippet initializes shared memory. Then freelist, list1 and list2 are placed into the shared memory. The three lists are stored in structure memory storage and assigned to the shared memory region created.

struct list{

char name[10];

int data[N]; //indexes of block

int count; //number of elements in list, used only to return last block

sem\_t b\_mutex; //binary semaphore

sem\_t c\_sem; //counting semaphore for available blocks in list.

};

//storage shared between processes, which is a structure

enum LIST\_NAME { LIST1=0, LIST2};

struct memoryStorage{

struct block blocks[N];

struct list flist; //Freelist

struct list lists[2]; //List1 and List2

sem\_t mem\_sem; //semaphore used for accessing memory

};

The above code snippet shows the structure memorystorage that have the three list freelist as flist, list1 as lists[1] and list2 as lists[2].

**Initially all the blocks in the freelist are empty:**

//initializing storage blocks

for(i=0; i < N; i++){

mem->blocks[i].id = i;

mem->blocks[i].data = 0;

}

The three process communicate in the shared memory and try to access the shared resources freelist, list1 and list2, all the three processes run concurrently where there is a possibility of accessing the shared resources at one creating in a dead lock or uncertain results. This project is a solution to control the access of shared resources by the processes without creating the deadlocks and ensuring maximum concurrency using semaphores.

**Semaphores used:**

Totally 7 semaphores have been used. free1ist has one binary semaphore(b\_mutex), one counting semaphore(c\_sem). list1 has one binary semaphore(b\_mutex), one counting semaphore(c\_sem). list2 has one binary semaphore(b\_mutex), one counting semaphore(c\_sem). The seventh semaphore is mem\_sem which is used for synchronized access of memory.

All the three processes use the respective semaphores of the lists before performing the link and unlink operations on any of the list to ensure any two processes are not trying to link or unlink the same list at once. Counting semaphore of freelist is initialized to N-1, so that there will be 1 block available for process2 to execute.

**Link, Unlink operations performed by the lists:**

Link is adding a block of information to any of the lists and unlink is removing the block of data from any of the lists, i.e link is adding the data and unlink is deleting the data. These two operations are performed by the functions

//link operation on lists

void Link(struct block \*b, struct list \* list){

print\_list(list, "before link");

list->data[list->count] = b->id; //push block index at end of list

list->count++;

print\_list(list, "after link");

}

//unlink operation on lists

struct block\* unLink(struct list \*list){

int i,bi; //block index

print\_list(list, "before unlink");

bi = list->data[0]; //take first free block index

//shift other blocks left

for(i=1; i < list->count; i++){

list->data[i-1] = list->data[i];

}

list->count--;

print\_list(list, "after unlink");

return &mem->blocks[bi]; //return block pointer

}

**Printing the lists after each operation on lists:**

Print\_list() function is called after and before every operation on lists to print the show the result of the particular operation on list.

//to print the list.

static void print\_list(struct list \* list, const char \* msg){

int i;

printf("%s %s[", Pname, list->name);

for(i=0; i < list->count; i++){

printf("%i ", list->data[i]); //show block index

}

printf("] %s\n", msg);

fflush(stdout);

}

**Execution of three processes performing linking and unlinking operations on lists using semaphores for mutual exclusive access:**

**Process1:** when process1 starts executing it first waits on the counting semaphore of freelist WAIT(&mem->flist.c\_sem)to ensure the availability of resource and then it will wait on its binary semaphore WAIT(&mem->flist.b\_mutex). If there is no other processing accessing freelist then process1 will unlink freelist i.e using wait on the binary semaphore a lock is applied on the resource so that no other process can access freelist. As soon unlinking is done, it releases the freelist by performing post on binary mutex of freelist(POST(&mem->flist.b\_mutex)), then process1 produces some information into block b. To link this produced block to list1 we will first lock the access to list1 by performing post on binary semaphore of list1(WAIT(&mem->lists[LIST1].b\_mutex)), after the linking is done we increase the count of full blocks on list1 using counting semaphore of list1(POST(&mem->lists[LIST1].c\_sem))and release the lock to list1.

**Process2:** When process2 starts executing it first waits on the counting semaphore of list1 to check availability(WAIT(&mem->lists[LIST1].c\_sem)), if available then we lock the list1 to ensure no other process is accessing list1, if some other process is already using list1 we need to wait until the other process release it. Then p2 unlinks list1 after which it releases list1. So, that other processes can access it. Then p2 will wait on mem\_sem semaphore to ensure no other process is using the same memorystorage block, after this it locks the access to freelist using its binary semaphore(WAIT(&mem->flist.b\_mutex)), unlinks a block from freelist and releases freelist. Then p2 uses block x to produce information into block y. Now p2 locks(WAIT(&mem->flist.b\_mutex)) freelist link block x to freelist increase the count on counting semaphore of freelist(POST(&mem->flist.c\_sem)), then release freelist(POST(&mem->flist.b\_mutex)). Then it locks list2(WAIT(&mem->lists[LIST2].b\_mutex)) , links the block y to list2 , increase the count on the counting semaphore of list2(POST(&mem->lists[LIST2].c\_sem)) and realses list2(POST(&mem->lists[LIST2].b\_mutex))

**Process3:** When process3 starts executing it will try to gain access to list2 and if know other process is using list2, p3 will decrease the count on counting semaphore of list2(WAIT(&mem->lists[LIST2].c\_sem)), lock list2(WAIT(&mem->lists[LIST2].b\_mutex)), unlink a block from list2 and then release it(POST(&mem->lists[LIST2].b\_mutex)) so, that other processes can use it.

P3 will consume information in block c, then lock freelist (WAIT(&mem->flist.b\_mutex)) link c to freelist and then does post on mem\_sem(POST(&mem->mem\_sem)and then releases freelist (POST(&mem->flist.b\_mutex)).

**Output code snippet(randomly selected while running):**

P1 FreeList[9 ] before unlink

P1 FreeList[] after unlink

P2 List2[6 ] after link

P1 List1[7 2 3 5 4 0 1 8 ] before link

P1 List1[7 2 3 5 4 0 1 8 9 ] after link

P3 List2[6 ] before unlink

P3 List2[] after unlink

P3 FreeList[] before link

P3 FreeList[6 ] after link

P2 List1[7 2 3 5 4 0 1 8 9 ] before unlink

P2 List1[2 3 5 4 0 1 8 9 ] after unlink

P2 FreeList[6 ] before unlink

P2 FreeList[] after unlink

P2 FreeList[] before link

P2 FreeList[7 ] after link

P2 List2[] before link

P2 List2[6 ] after link

The above output snippet shows that the three processes are running concurrently linking and unlinking nodes on the three lists.

From the above snippet:

P1 is accessing freelist during which p2 and p3 may be waiting on FL and as soon as p1 releases freelist and while it is using list1, p3 starts using freelist. Initially p2 is accessing list2, once p2 released list2, p3 started using list2. p2 starts accessing list1 as soon as it is released by p1. This shows each of the processes are running concurrently accessing the shared resources and no process is unnecessarily blocking any of the lists from being accessed by other process. From the above snippet it is clear that while p1 is using freelist, p2 is using list2 and p3 is using list1, as soon as p1 released FL and started using list1, p3 started using FL. As soon as p1 released list1 p2 started using list1 and this continues.