CS 50300
Spring 2017

(Taking one extension day)

Lab 3

Submitted By: Rashmi Soni
PUID: 0029136200

1. Blocking Message Send & Receive

To implement the blocking functionality of send and receive; a buffer is maintained which stores a max of 3 1-word msgs.

Modified files:

In include directory, ‘process.h’

Changes: A new process state ‘PR_SND’ is defined. In the struct of ‘procent’, six new fields are added, namely; a buffer which stores 3 1-word msgs, another buffer which maintains whether there is msg present at the reading or writing position of buffer, field that points to the front of the buffer, field that points to the rear of buffer, size of the buffer, associated queue which stores the processes in PR_SND state for a particular receiver:

umsg32 prmsgbuf[3]; bool8 prhasmsgbuf[3]; int32   prfront; int32   prrear; int32   prsize; qid16 prqueue;

In include directory, ‘queue.h’

Changes: To accommodate the ‘blocking queue’ for each process, queuetab is overloaded and the value of NQENT is changed to ‘(NPROC + 4 + NSEM + NSEM + NPROC + NPROC)’ because it now accounts for the head and tail of blocking queue for each process. (Since the number of processes can be max NPROC, hence there will be 2*NPROC entries for all processes). With this, we can utilize the existing functions of queues, but there will be some ripple effects in this implementation. We need to initialize all queues associated with each process (changed files are mentioned below). Again, we need to initialize all variables associated with the queue and changes in kill.c file.

In system directory, ‘initialize.c’

Changes: The values for each of the new fields except ‘blocking queue’ of ‘procent struct’ (mentioned above) are initialized for NULL process. And for all processes, the blocking queue is initialized to a new queue.

In system directory, ‘create.c’

Changes: The values for each of the new fields except ‘blocking queue’ of ‘procent struct’ (mentioned above) are initialized for all the processes.

In system directory, ‘kill.c’

Changes: A new case for process state ‘PR_SND’ is included in the switch case, wherein if a blocking sender gets killed, it should be removed from the ‘blocking queue’. Moreover, when a process is terminated, all the processes in its associated blocking queue should be unblocked.

In include directory, ‘prototypes.h’

Changes: extern syscall sendb(pid32, umsg32);

extern umsg32 receiveb(void);

New files added:

In system directory, 'sendb.c'

Function added: syscall sendb (pid32 pid, umsg32   msg)

Purpose: This function sends a msg to the receiver is there is space available in receivers' buffer. It puts the msg at the rear of the buffer and also checks there was no msg already present at the rear position; and update the front, rear, size and associated fields of the buffer; and making the receiver ready if it is in blocking state. Else, if the buffer is full, or the writing position already has some msg, then this function blocks the sender and enqueues it into the 'blocking queue' of the receiver and reschedules other process.

In system directory, 'receiveb.c'

Function added: umsg32 receiveb(void)

Purpose: This function checks if the buffer of the receiver is empty or the reading position of the buffer doesn't contain any msg, then the receiver gets blocked as 'PR_RECV' state. Else, it checks if the reading position contains some msg, then reads the msg (in FIFO fashion) from buffer and update the associated fields of the buffer. It then unblocks one of the senders' from the 'blocking queue' before exiting.

2. Asynchronous Message Receive

Modified files:

Approach: The receiver needs to explicitly register the callback function to allow itself to receive the msg asynchronously. If the callback function is not registered by the receiver or it de-registers itself from the callback functionality, then normal processing of send, receive sys calls follows.

In include directory, 'process.h'

Changes: Two new fields for 'procent struct' are added, namely,

umsg32 *abuf;

int (* func) (void);

First field stores the location where the callback function will write the msg.

Second field stores the function pointer of the callback function.

In system directory, 'initialize.c'

Changes: The values for each of the new fields of 'procent struct' (mentioned above) are initialized for NULL process.

In system directory, 'create.c'

Changes: The values for each of the new fields of 'procent struct' (mentioned above) are initialized for all the processes.

In system directory, 'send.c'

Changes: This function is modified so that it checks if the receiver has registered for the callback function. If so, then it places the msg to the location pointed by the 'abuf' pointer and calls the callback function. Rest of the functionality of original send remains the same.

In include directory, 'prototypes.h'

Changes: extern syscall registerrecv(umsg32 *abuf, int (* func) (void));

New files added:

In system directory, 'registerrecv.c'

Function added: syscall registerrecv(umsg32 *abuf, int (* func) (void))

Purpose: This function is used by the receiver to register for the callback function. It specifies the location where the msg needs to be put as well as the function pointer of the callback function.

Bonus Problem:

Some of the approaches through which the callback function is not executed in the kernel mode are:

1) The sender process in the application calls the send() sys call. So, the send() call can return the pointer which points to the callback function. Thus, it will be received in the sender process which thereby implies that the callback function runs in the user mode instead of kernel mode.
2) The msg buffer pointer is defined by the receiver, which in turn will be used by the sender to send the msg. So, as soon as the receiver process starts executing, the stack of the receiver should be modified(during the resched functionality) in a such a way that for every msg, the receiver always imitates the callback function and then comes back. This approach ensures that the callback function is executed in the user mode.
3) Instead of running the callback function in send() sys call, a newly created process inside send(which always runs in user mode) accepts the callback function and arguments to be its first execution step. This makes sure that callback function runs in the user mode because a newly created process always runs in the user mode.
4) The first execution step of the receiver process should be to execute the callback function and then move on to its normal execution. The knowledge about the layout of the stack when the context switch is applied to a process is required here. So, we can implement such that when a resched is returned, it resembles like a function call to the callback function. Thus, point should be taken care as the callback function should always return to the normal execution where the resched might have returned in the execution. So, we can argue that the callback function is executed in the user mode.

3. Garbage Collection Support

Design and its rationale: A per-process list is maintained which stores all the book keeping nodes. This book keeping node stores the allocated memory block to the process (at a given instant) as well the length of that allocated address and corresponding next book keeping address. When a user requests for a memory block, getmemgb() sys call allocates the memory block to the user as well as allocates some extra amount of memory to store the book keeping information for each allocated memory. This approach has the advantage since it allocates these two memory blocks(by calling getmem sys call twice) in a non-contiguous manner. So if the memory chunk is available for the 'requested bytes' and the 'book keeping node bytes' in a non-contiguous manner, then the request of the user is satisfied. However, this approach accounts to utilize more memory from the free list to accomodate a book keeping node (as we are storing three fields).

Another approach is to store the book keeping information of the allocated address in the global free list itself. So this approach calls getmem() sys call once, but tries to allocate the two memory blocks in a contiguous manner. But this approach may not satisfy the user request if these two memory chunks are present in the non-contiguous manner. It also accounts to complex the arithmetic (rounding, truncating) of the number of bytes. So, the first approach is followed.

To implement per-process centric gb collection, the changes made to Xinu are:-

Modified files:

In include directory, 'process.h'

Changes: new structure 'prcblk' is defined for book keeping node list and each process gets associated with a header field to point to the head of book keeping data structure.

struct prcblk { struct prcblk *pnext; char *add; uint32 plength; };

In include directory, 'prototypes.h'

Changes: The following lines are added

extern char *getmemgb(uint32);

extern syscall freememgb(char *, uint32);

extern syscall freememgbpid(pid32, char *, uint32);

In system directory, 'create.c'

Changes: The header field of Per-process book keeping data structure is initialized for all the processes.

In system directory, 'initialize.c'

Changes: The header field of NULL-process book keeping data structure is initialized.

In system directory, 'kill.c'

Changes: De-allocating book keeping collection and the corresponding allocated memory segments of a process when it is terminated (if the process has not done so already))

New files added:

In system directory, 'getmemgb.c'

Function added: char *getmemgb(uint32 nbytes)

Purpose: This function allocates 'nbytes' number of bytes(rounding and truncating as required) from the free memory of kernel to the user and also allocates number of bytes required to store the 'prcblk' structure size to store the book keeping collection node. This node stores the allocated address from 'memlist' data structure and its corresponding length. The next pointer points to the next node in the book keeping data structure. This new node is then appended in the per-process book keeping data structure.

In system directory, 'freememgb.c'

Function added: syscall freememgb(char *blkaddr, uint32 nbytes)

Purpose: This function de-allocates 'nbytes' number of bytes(rounding and truncating as required) to the free memory of kernel and also de-allocates book keeping collection node which stored this 'blkaddr'.This node is then excluded from the per-process book keeping data structure.

In system directory, 'freememgbpid.c'

Function added: syscall freememgbpid(pid32 pid, char *blkaddr, uint32 nbytes)

Purpose: Similar to the freememgb() sys call, this function de-allocates memory for a specified process id.

(Note: In main.c, uncomment different test cases to test for all three parts. The functionality of a test case is mentioned above it and its corresponding output at the below)