CS 503 - Spring 2014

Lab 3: Interprocess Communication and Low-level Memory Management

Due Date: Monday 03/24/2014 11:59PM

In this lab, we will delve into kernel implementation issues of IPC (interprocess communication) and low-level memory management.

Begin with a fresh copy of `xinu-14spring-linksys.tar.gz`

---

1. Blocking Message Send & Receive (30%)

Problem 1 concerns the implementation of a blocking version of *send*(), call it *sendb*(). *sendb*() has the same function definition as *send*(). Each receiver has a message buffer, which could hold 3 1-word messages. If the buffer is not full, *sendb*() behaves as *send*() and it should check if receiver is blocked and unblock blocked receiver. If the receiver's buffer is full, however, *sendb*() blocks, i.e., the kernel context switches out the calling process and puts it into a waiting state, call it SENDING. This is implemented by defining an additional state constant PRSND (choose what value to assign it to) in process.h.

On the receiver side, please implementing a new systemcall *receiveb*(). It's similar to *sendb*(). If the buffer is empty, it should be blocked until another process sends message to it. If the buffer is not empty, it always picks up the first message in the buffer. Here your implementation should guarantee the FIFO of receiving messages. And before a receiving process returns from *receiveb*() it must check if there are any blocked sending processes, and if so, unblock one of the waiting processes. When implementing "unblock" there are at least several issues to consider: one, whom to unblock, two, how to unblock, and three, where to queue the list of blocked processes. Fourth, when the receiver is terminated, it should unblock all blocked sending processers before it exits. And those senders are not allowed to continue to put their messages into receiver's buffer at this moment. Last, but not least, there is a degree of freedom in deciding where to maintain the queue of blocked processes. Due to performance considerations, it is preferable that there is a per-receiving process queue of blocked processes in SENDING state so that dequeueing and enqueueing time are dependent on the number of processes sending to the same process. Dynamic allocation of queue memory should be avoided for performance reasons.

A default solution is to overload queuetab[] in queue.h (look into related source code, including initialize.c, to see how processes in READY and WAITING states are stored in different segments of queuetab[]). How you choose to implement the specifics is up to you. Make sure to explain your design decisions and rationale in a separate write-up (be brief and to-the-point). Your discussion should also touch upon dependence issues, i.e., do data structure modifications have ripple effects elsewhere in the kernel requiring explicit code changes.

Create 2-3 test cases whose output (e.g., senders may print sender-specific 1-word messages before calling *sendb*() and a receiver may print a received message) demonstrate the correct functioning of the blocking message send extension of kernel services. If you find a need, you may use the *sleepms*() system call in conjunction with busy looping (i.e., for-loops performing non-I/O instructions) to craft your test cases. It is important to benchmark a system when it is stressed, hence don't be shy with respect to the number of sending processes in your tests.

---

2. Asynchronous Message Receive (40%)

Problem 2 concerns implementation of asynchronous receive in Xinu using a callback function that the application programmer (i.e., Xinu process) registers with the kernel. This is in addition to a pointer to 1-word buffer in user space to which the kernel is asked to copy an arriving message. Thereafter whenever a 1-word message is sent to the receiver, the message is copied from kernel space to user space and the registered callback function is executed which handles the chore of processing the received message (in our case just printing). As discussed in class, there are a range of flavors by which asynchronous receive can be realized. In this problem we will commit to the following overall structure.

First, the kernel exports a system call, *registerrecv*(), that is used by an application to specify a user space 1-word buffer and a user space callback function that should be executed when a message is received. For brevity, we will adhere to the convention that a callback function must be of type *int* and contains no argument. The function definition is given by

- SYSCALL registerrecv(umsg32 *abuf, int (* func) (void))

where abuf is a pointer to a 1-word user space message buffer and func is a function pointer to a user space callback function. The application programmer might write a callback function that reads

```
int myrecvhandler(void) {
   kprintf("msg received = %d\n",recvbuf);
   return(OK);
}
```

where recvbuf is *extern*. *main*() might register asynchronous message receive with the kernel by doing

```
if (registerrecv(&recvbuf,&myrecvhandler) != OK) {
   kprintf("recv handler registration failed\n");
   return 1;
}
```

The design and implementation issues facing the kernel designer include, one, where to maintain the pointers to user space message buffer and callback function, and two, how to implement callback function execution upon message receipt.

In the first approach, we will borrow the context, i.e., process stack, of the current process (in our case the sending process) to run the callback function registered by the receiving process after copying the message to the receiver's user space message buffer. That is, the kernel invokes the callback function (e.g., *myrecvhandler*() in the above example) from within the kernel. Since the C compiler injects prologue code when compiling *myrecvhandler*() for pushing/popping the run-time context of *myrecvhandler*() in the process stack pointed to by the current process's stack pointer we are assured correctness with respect to function call book keeping. A possible place for the kernel to call *myrecvhandler*() through its registered function pointer from within the kernel is right after a sender process has successfully written a 1-word message to the receiver's 1-word message buffer in the receiver's process table entry.

It is important to note that the above solution is fraught with danger since the kernel executes user-supplied code (i.e., callback function *myrecvhandler*()) which could contain bugs, or even worse, malware that can wreck havoc with the system. Of course, in our version of Xinu that runs in real mode and does not provide isolation/protection, bugs in one process could negatively impact other processes since they share everything except their process stack. And even the process stack is not immune from corruption due to stack overflow.

Implement the above approach that handles the details needed to achieve correctness. As always, pay attention to performance issues when choosing one design decision over another and indicate your rationale. As in Problem 1, provide 2-3 test cases that demonstrate the correct functioning of the asynchronous message receive extension of the Xinu kernel. Imparting stress during benchmarking is critical for meaningful evaluation of the kernel's correctness and performance.

Bonus Problem (10%)

If you have spare cycles, you may consider finding an alternate solution that does not entail running the callback function from within the kernel. That is, it is executed after returning to user space (i.e., toggling back to user mode from kernel mode in protected mode kernels). It is not needed that you implement the solution but your design and explanation should clearly establish that your solution is correct.

3. Garbage Collection Support (30%)

Xinu uses *getmem*() to allocate heap memory from a single linked list of free memory segments and *freemem*() to return unused memory to the free memory pool. The kernel keeps track of per-process stack memory so that when a process terminates its stack memory is returned to the free memory list via *freestk*(). This is not the case, however, for memory allocated by *getmem*() which gets only returned if a process explicitly deallocates memory by calling *freemem*() which is voluntary. This puts the kernel at the mercy of user processes which is highly undesirable. Even when an application programmer ardently tries to free allocated memory before exiting, programming mistakes and bugs may result in imperfect return. These memory leakage problems may be addressed by injecting garbage collection support inside the kernel that keeps track of allocated memory on a per-process basis and returns them to the free list when a process terminates (should it not have done so). Ignoring shared memory issues — we have not provided system call support for orderly sharing of memory segments between processes — we may institute a process-centric garbage collection mechanism inside Xinu which allows straightforward book keeping of per-process allocated memory without worrying about maintaining per-memory segment reference counts (i.e., when a memory segment is shared by two or more processes, only when all processes have dereferenced or exited can the memory segment be freed).

Design and implement process-centric garbage collection support in Xinu by modifying the relevant system calls, including *getmem*() and *freemem*(), and relevant parts of the kernel such that perfect garbage collection is assured. That is, when a process terminates, it is guaranteed that all memory segments allocated to the process have been returned to the free list. To achieve clean design and modularity, we will not rewrite *getmem*() and *freemem*() (and related systems calls such as *getstk*() and *freestk*() if there is a need) directly but export garbage collection enabled sibling calls (*getmemgb*(), *freememgb*(), etc.) that the application programmer is asked to use. Of course, the kernel can always enforce garbage collection by making *getmem*() and *freemem*() to be wrapper functions to system calls *getmemgb*() and *freememgb*().

The first design decision you will have to make is whether to explicitly include *getstk*() and *freestk*() within the scope of modification. Dependence issues exist (e.g., *freestk*() is a macro that invokes *freemem*()) and a simple, straight-and-narrow solution is possible by including stack memory allocation/deallocation within the scope. But this is not necessary to achieve a correct solution. The choice is up to you. It is important that you understand how Xinu initializes system resources including memory (sys/initialize.c is a good starting point) so that accurate accounting can be maintained when application processes are created and executed. In a separate write-up, discuss your design choices and their rationale.

Benchmark your garbage collection enabled Xinu kernel on 2-3 test applications that demonstrate correct functioning of your system. To do so, at the very least, your test applications should do their own accounting to keep track of how much memory they have been allocated and how much memory they have freed. Of course, one must be careful to do the accounting arithmetic correctly so that they accurately mirror the rounding (and truncating) performed in the Xinu kernel. This, in turn, must be compared with accounting performed inside the modified Xinu kernel.

---

4. Turnin Instructions

Include your writeup as a text or postscript file in your xinu (*xinu-14spring-linksys*) directory as the file Lab3Answers.txt, Lab3Answers.ps, or Lab3Answers.pdf (depending on the format used). Your write-up should clearly specify what changes to Xinu have been made. That includes addition of new files containing what functions have been added (follow the file naming convention of legacy Xinu to the extent possible) and what functions have been modified (including header files). Do not rely on Makefile alone to convey this information.

Pre-turnin instructions

1. Make sure you named the written part of the lab correctly: the filename, before the extension, should be: Lab3Answers
2. Make sure the lab submissions clearly bear your name.
3. As always, make sure your kernel compiles correctly:

   ```
   make clean
   make
   ```

4. Make sure that you turned off all debug printing statements. Before turning in make sure you do a -

   ```
   make clean
   ```

Turnin lab3 code using:

```
turnin -c cs503 -p lab3 xinu-14spring-linksys
```