CS 503 Fall 2022

Lab 4: Preemptible and Reentrant IPC, and Asynchronous Event Handling using Callback Function (280 pts)

Due: 11/2/2022 (Wed.), 11:59 PM

1. Objectives

The objective of this lab is two-fold. First, support asynchronous event handling via user callback functions. Second, utilize synchronization/coordination primitives to implement a preemptible, reentrant XINU upper half that supports message passing IPC. Both make use of ROP to manipulate the runtime stack and dynamically reroute kernel and user code execution flow.

---

2. Readings

1. XINU set-up
2. Read Chapters 8 and 11 of the XINU textbook.

Please use a fresh copy of XINU, xinu-fall2022.tar.gz, but for preserving the myhello() function from lab1 and removing all code related to xsh from main() during testing. main() serves as an app for your own testing purposes. The TAs will use their own main() to evaluate your XINU kernel modifications.

---

3. Asynchronous event handling using callback function [140 pts]

3.1 Overview

Programmers utilize asynchronous event handling supported by kernels to build apps to manage concurrency without having to resort to multithreading which can incur significant overhead. For example, a client app may send a request to a server then block on a response. Before making a blocking system call to await a response, the client may set a timer alarm to go off 500 msec in the future. Along with the alarm, the client registers a callback function with the kernel so that the kernel executes the callback function when the alarm is triggered. The callback function may do a number of things such as resending the request in case it was lost.

In UNIX/Linux parlance events exported to user mode are called signals. Callback functions to be executed when signals are raised are called signal handlers. Similar constructs exist in Windows. Since callback functions are user code, an important consideration when implementing asynchronous event handling kernel support is ensuring that they are executed in user mode in the context of the process that registered them to preserve isolation/protection. Even though the kernel arranges for the asynchronous, event triggered execution of a callback function -- the callback function is not invoked by synchronous user code -- it must do so while preserving isolation/protection.

3.2 Timer alarms and asynchronous handler registration

We will introduce a system call, syscall alarmx(uint32 timeval, void (* ftn) (void)), which registers a function pointer ftn with XINU to be executed after timeval milliseconds have elapsed. alarmx() returns SYSERR if timeval is not greater than 0 or two existing alarms are already set (i.e., at most 2 alarms may be outstanding). Otherwise, alarmx() returns 0. The number of alarms set for a process is tracked in a new process table field, uint16 prnumalarms, which is initialized to 0. We will utilize XINU's sleep queue to implement timer alarms. To set a new alarm with time interval timeval, alarmx() calls

insertd(pidalarm, sleepq, timeval)

which inserts a process with PID pidalarm into XINU's sleep queue, sleepq, to be woken up after timeval milliseconds have elapsed. Since XINU assumes that a process can appear in at most one queue, to support up to 2 outstanding alarms we will relax this assumption by reconfiguring NQENT in include/queue.h to NPROC + NPROC + NPROC + 4 + NSEM + NSEM. The increase in queuetab[] size by 2 * NPROC provides space to allow a process to appear twice more in the sleep queue -- in addition to being in the queue due to a process calling sleepms() -- to support timer alarm events. The static variable nextqid must be accordingly reconfigured in newqueue() in system/newqueue.c so that heads/tails of XINU queues start from queuetab[3*NPROC].

The first argument pidalarm of insertd() is set to NPROC + currpid if there are no outstanding timer alarms for the current process. pidalarm is set to (2 * NPROC) + currpid if there is one existing timer alarm already set (but not expired) for the current process. Note that sanity checks that disallow process IDs exceeding NPROC must be relaxed in select XINU kernel functions. Similarly the range checks for queue IDs must be changed to reflect the revised queuetab[] configuration. Modify wakeup() which is called by clkhandler() so that arrangements can be made to execute the registered callback function which is remembered in a new process table field, void (* prcbftn) (), when alarmx() is called. wakeup() needs to check the PID returned by dequeue() to determine if the wakeup event associated with the dequeued process is for sleepms() (i.e., PID < NPROC) or alarmx() (i.e., NPROC < PID < 3 NPROC). For a process whose timer alarm just expired, wakeup() sets the value of a process table field, uint16 prmakedetour, to 1 (initial value 0). Other chores include decrementing prnumalarms. Regular sleeping processes that awoke are readied after dequeueing from sleepq. alarmx() may be called multiple times by a process with the restriction that at most two timer alarms are allowed to be outstanding. Function pointers are allowed to change when alarmx() is called with the understanding that the one passed in the last call of alarmx() overwrites earlier ones. If successive alarmx() calls by a process result in timer alarms being triggered at the same time, the callback function is executed only once.

3.3 Arranging execution of callback function in user mode

We will consider two cases for a process whose timer alarm has expired to make arrangements for the process's callback function to be executed in the context of the process in user mode.

Case (i) The process whose timer alarm expired is the current process, or a ready process that executed before and was context-switched out due to depleting its time slice. In the former clkhandler() returns to clkdisp which executes iret to return to interrupted user code. Although we will not implement trapped system calls in lab4, there is nevertheless a well-defined boundary between kernel code and user code where iret executed by clkdisp jumps from kernel code to user code. Before clkdisp executes iret, it calls, void executedetour(void), in system/executedetour.c which decides if a detour to a callback function needs to be made based on the value of prmakedetour of the current process. If prmakedetour equals 1 then executedetour() uses ROP techniques considered in lab1 to manipulate the runtime stack of the current process so that upon execution of iret a jump is made to the callback function. When the callback function completes and executes ret a jump is made to a function, void restoreregs(void), in system/restoreregs.S whose main task is to restore the 8 general purpose register values saved onto the stack by clkdisp and jump to the original return address of clkdisp. Hence two outcomes, in addition to executing the registered callback function, must be achieved by the detour mechanism: one, put the runtime stack in the same state as it would have been had clkdisp executed iret without a detour, and two, restore the general purpose register values of the current process whose context is being borrowed to execute lower half kernel code to their values before an interrupt.

The same control flow holds for a ready process that was context-switched out due to depleting its time slice. When the process becomes current, it transitions from kernel code to user code upon executing iret in clkdisp. We do not need to consider ready processes that execute for the very first time since, by definition, to register a callback for timer alarm by calling alarmx() a process must have executed before.

In general, restoring the content of EFLAGS is desirable although a case may be made that when a function calls another function (i.e., synchronous execution of code) caller/callee convention does not save/restore the content of EFLAGS. In the case of callback functions executed asynchronously, synchronous code that was interrupted by an asynchronous event may not be coded so as to be reentrant in the sense of not being adversely affected by the execution of a callback function. For example, a computation carried out by synchronously by a process may, just before being interrupted, have resulted in the ZF (zero flag) being set to 1. Subsequent synchronous code execution may then depend on whether ZF is set. However, asynchronous execution of callback function code interleaves in-between, and it may be that computation carried out by the callback function sets ZF to 0. When the synchronous computation resumes, its computation may have been unpredictably influenced by asynchronous interleaving of callback function code. This is different from synchronous invocation of callback function code where the programmer knows where in synchronous code a call is being made, and takes steps before synchronously calling the callback function so that it does not cause unintended side effects. For Problem 3, it is recommended that EFLAGS be saved and restored before returning to the code of the process that was interrupted. However, points will not be deducted if EFLAGS is not saved/restored. Implementing EFLAGS restoration is a straightforward matter. Understanding its ramifications with respect to semantics of kernel intervention is more involved and subtle, and important to consider.

Case (ii) We will condense all other cases to processes blocking by calling receive() or sleepms() after registering a callback function for timer alarm using alarmx(). Since we will not implement trapped system calls as in lab2, we will call a kernel function, void executedetour2(void), in system/executedetour2.c, before receive() returns which decides if a detour to a callback function needs to be made based on the prmakedetour value of the current process. That is, when a process blocking on receive() becomes current, instead of returning to the caller of receive(), say main(), receive() makes a detour to the registered callback function before returning to its caller. Since receive() and sleepms() are regular C function calls, we will modify them to call the callback function after interrupts are restored. When the callback function returns to receive() or sleepms(), they return to their caller. In trapped versions of receive() and sleepms(), ROP techniques are used as in Case (i) to untrap to user mode to execute the callback function before resuming execution of user code after the call to receive() or sleepms().

3.4 Testing

Test and verify that your implementation works correctly. Describe in lab4.pdf your method and test cases for gauging correctness.

---

4. Preemptible and reentrant IPC support [140 pts]

4.1 Overview

XINU is a non-preemptible kernel in that both upper and lower halves run with external interrupts disabled which on our uniprocessor x86 Galileo backends assures kernel code is executed atomically. System calls such as read() and write() in Linux are referred to as slow system calls since completion time depends linearly on the size of the data to be read/written. General purpose production kernels (e.g., Linux, Windows, MacOS) are designed to be preemptible to facilitate responsiveness. Only parts of kernel code that access shared resources and can be performed quickly may be executed non-preemptively.

Reentrant IPC support means that even though system calls such as read() and write() may share kernel data structures they do so in an orderly manner. That is, even though the two instances of system calls that share resources execute with instructions interleaved, the shared data structures are not corrupted. In particular, the outcome of interleaved execution is equivalent to the system calls executing sequentially (serializability semantics).

A third feature applies to asynchronous event handling support of Problem 3. When a read() system call is in the midst of executing upper kernel code and an event arises such as the timer alarm in Problem 3 for which a callback function has been registered, the system call returns with partial read. The same goes for write() if an asynchronous event relevant to the process interrupts its execution. If read() and write() are reentrant, the kernel guarantees that when read() or write() are interrupted a callback function that executes asynchronously as a result of the interrupt and, in turn, calls read() or write(), preserves correct execution of read() and write().

4.2 Function prototypes of sendx() and receivex()

We will implement variants of XINU's send() and receive() system calls whose interface supports variable length messages. Unlike send() which is nonblocking, sendx() is made blocking for the first process that attempts to send a message to a receiver process whose message buffer is already occupied. Its state is changed to PR_SENDBLOCK. Define its value in include/process.h. Subsequent sendx() calls by processes are nonblocking and return SYSERR if there is a sender process blocking. When a receiver process reads the message in its buffer, a blocker sender process is unblocked by copying the sender's message temporarily saved at sender side to the receiver buffer and changing its state from PR_SENDBLOCK to PR_READY and readied by calling ready(). Define PR_SENDBLOCK in include/process.h. The function prototype of sendx() in system/sendx.c is

syscall sendx(pid32 pid, char *buf, uint16 len);

where pid specifies the PID of the receiver process, buf is pointer to the sender process's user buffer containing the bytes of a message to send, and len specifies the number of bytes to send. sendx() has a similar interface as the write() system call in Linux but for the first argument that specifies the receiver's PID instead of a file descriptor, The function prototype of receivex() in system/receivex.c is

syscall receivex(pid32 *pidptr, char *buf, uint16 len);

where pidptr is used to communicate the sender's PID, buf is pointer to the receiver process's user buffer, and len specifies how many bytes the receiver wants to read. receivex() remains a blocking system call. Before returning receivex() checks if there is a blocked sender process. If there is, the sender is unblocked after copying its message to the receiver's message buffer. As noted in class, please be aware that blocking sendx() due to racing conditions can lead to deadlock. XINU will adopt the Ostrich approach for (not) dealing with deadlocks.

4.3 Variable message size

sendx()/receivex() support messages of variable length maximum size is given by IPCX_MAXLEN. Set its value to 6000 in a new header file include/ipcx.h. Add two process table fields, char prrecvbuf[IPCX_MAXLEN] and char prsndbuf[IPCX_MAXLEN], that are used to store messages at the receiver and sender sides. We will impose a one message semantics where only one message resides in prrecvbuf[] and prsndbuf[], albeit of variable length. Even if there is available space in prrecvbuf[] and a message from a second sendx() would fit in the available space, the receiver's buffer is treated as occupied. prsndbuf[] is used to temporarily hold a sender's message that enters state PR_SENDBLOCK. Add process table field, uint16 prrecvlen, which specifies the size of the message (in bytes) stored in prrecvbuf. If prrecvlen equals 0 it means the receiver's message buffer is empty. Introduce process table field, pid32 prsenderpid, which specifies the PID of the sender process whose message is held in prrecvbuf[]. Add process table field, pid32 prblockedsender, which specifies the PID of a blocked sender process. If prblockedsender equals 0 it means there is no blocked sender process. Add a process table field, pid32 prblockonreceiver, which specifies the PID of the receiver process a sender process is blocking on. Set prblockonreceiver to 0 if the sender is not blocking on a receiver.

4.4 Reentrance of sendx() and receivex()

We will execute part of the XINU's upper pertaining to sendx() and receivex() system calls with interrupts enabled. A process executing kernel code sendx() may be interrupted in the midst of executing instructions of sendx() which leads to preemption by another process that may call receivex() or sendx(). Since sendx() and receivex() share kernel data structures, interleaving of instructions may lead to outcomes that violate correctness with respect to serializability. For example, depending on how sendx() is coded it may happen that a process that is about to block and enter state PR_SENDBLOCK is preempted by a receiver process which checks if there is a blocked process to be readied after consuming the bytes in its message buffer. The receiver may conclude that there is no blocked process because the sender who is about to block was preempted before it had a chance to update the prblockedsender field of the receiver process. As a consequence, the blocked sender in state PR_SENDBLOCK may not become unblocked after the receiver's buffer has become empty which violates the semantics of sendx(). We will use a per-receiver process semaphore, sid32 pripc, to assure correctness of concurrent execution of sendx() and receivex() system calls whose instructions may interleave. pripc is initialized to 1, i.e., pripc = semcreate(1).

A process, sender or receiver, first calls wait() with pripc of the receiver process to acquire the right to perform operations that may change shared kernel data structures. For example, a sender process calls wait(receiver_ptr->pripc) upon entering sendx() before attempting to perform sending related operations where receiver_ptr is a pointer to the receiver's process table entry. Only after wait() returns does sendx() perform copy of bytes from user buffer to kernel buffer and update relevant kernel data structures. After the requested service by sendx() is complete, sendx() calls signal(receiver_ptr->pripc) to allow another process to perform IPC operation on the same receiver. As a second example, when a receiver process tries to read a message using receivex(), receivex() first calls wait(receiver_ptr->pripc) on its IPC semaphore to acquire the right to access its message buffer. Only after wait() returns does receivex() proceed with actual message read related operations. If the receiver's message buffer is empty, the receiver will block since receivex() is a blocking system call. Before receivex() blocks the receiver process by changing its state to PR_RECV and calling resched(), it calls signal(receiver_ptr->pripc) to release the semaphore.

A sender process, after successfully acquiring the receiver's pripc semaphore, performs its sending operation. Before sendx() returns, it checks if the receiver is in state PR_RECV, and, if so, unblocks the receiver process. Only then does the sender release the semaphore by calling signal() before returning from sendx(). Although only one process may block by entering state PR_SENDBLOCK when attempting to send a message, multiple sender processes attempting to send messages to the same receiver process may block in the receiver process's semaphore queue associated with pripc. As long as the process that holds the semaphore is in ready state, it will eventually become current (if resched() does not cause starvation) and release the semaphore thus unblocking one of the processes blocked on the receiver's semaphore.

4.5 Testing

Test and verify that your implementation works correctly. Describe in lab4.pdf your method and test cases for gauging correctness.

Bonus problem [25 pts]

As indicated in 4.1, a third feature of reentrant sendx() and receivex() system calls pertains to interaction with asynchronous IPC of Problem 3. Suppose a process that makes sendx() and/or receivex() system calls also engages in asynchronous IPC using callback function by calling alarmx(). For example, alarmx(500, &myalarmhandler), where myalarmhandler() is a function to be executed in user mode in the context of the process that called alarmx() 500 msec in the future when the timer alarm expires. The process that called alarmx() may be in the midst of executing kernel code receivex() when clkhandler() calls wakeup() which detects the 500 msec timer alarm has expired. executedetour() called by clkdisp arranges for the receiver process to make a detour to its callback function myalarmhander() before returning to kernel code in receivex() that was interrupted by XINU's 1 msec system timer. One option is for receivex() to continue executing and return normally to its caller. Another option is for receivex() to cease execution prematurely by returning in EAX the number of bytes it has read before the timer alarm expired. The programmer then decides what to do next, by default, calling receivex() again to read the remaining bytes of the message that was cut short. We say that receivex() supports reentrance with automatic restart if the programmer is shielded from having to call receivex() again to read the remaining bytes.

A second scenario is when the process that called alarmx(500, &myalarmhandler) has blocked upon calling receivex() because its message buffer is empty. To execute the callback function myalarmhandler() in user mode in the context of the process that called alarmx(), the kernel would need to unblock the receiver process, i.e., ready it. When the receiver process eventually becomes current, we want it to return -2 to indicate that blocking on receivex() was cut short due timer alarm expiration and execution of myalarmhandler() in its context in user mode. By default, the programmer can call receivex() again to put the receiver process back in PR_RECV blocking state. If this is performed by the kernel so that it alleviates the programmer from calling receivex() again, we say that receivex() is reentrant with automatic restart. If the kernel modifications of Problem 3 were combined with Problem 4, would the resultant system support reentrance with automatic restart for receivex()? If not, how would you modify the combined code so that the resultant receivex() becomes reentrant with automatic restart? Explain your reasoning and describe a concrete solution in lab4.pdf.

Note: The bonus problem provides an opportunity to earn extra credits that count toward the lab component of the course. It is purely optional.

Turn-in instructions

General instructions:

When implementing code in the labs, please maintain separate versions/copies of code so that mistakes such as unintentional overwriting or deletion of code is prevented. This is in addition to the efficiency that such organization provides. You may use any number of version control systems such as GIT and RCS. Please make sure that your code is protected from public access. For example, when using GIT, use git that manages code locally instead of its on-line counterpart github. If you prefer not to use version control tools, you may just use manual copy to keep track of different versions required for development and testing. More vigilance and discipline may be required when doing so.

The TAs, when evaluating your code, will use their own test code (principally main()) to drive your XINU code. The code you put inside main() is for your own testing and will, in general, not be considered during evaluation.

If you are unsure what you need to submit in what format, consult the TA notes link. If it doesn't answer your question, ask during PSOs and office hours which are scheduled M-F.

Specific instructions:

1. Format for submitting written lab answers and kprintf() added for testing and debugging purposes in kernel code:

- Provide your answers to the questions below in lab4.pdf and place the file in lab4/. You may use any document editing software but your final output must be exported and submitted as a pdf file.
- For problems where you are asked to print values using kprintf(), use conditional compilation (C preprocessor directives #define combined with #if and #endif) with macro XINUTEST (in include/process.h) to effect print/no print depending on if XINUTEST is defined or not. For your debug statements, do the same with macro XINUDEBUG.

2. Before submitting your work, make sure to double-check the TA Notes to ensure that any additional requirements and instructions have been followed.

3. Electronic turn-in instructions:

i) Go to the xinu-fall2022/compile directory and run "make clean".

ii) Go to the directory where lab4 (containing xinu-fall2022/ and lab4.pdf) is a subdirectory.

For example, if /homes/alice/cs503/lab4/xinu-fall2022 is your directory structure, go to /homes/alice/cs503

iii) Type the following command

turnin -c cs503 -p lab4 lab4

You can check/list the submitted files using

turnin -c cs503 -p lab4 -v

Please make sure to disable all debugging output before submitting your code.

Back to the CS 503 web page