CS 354 Spring 2025

Lab 4: Asynchronous IPC using Callback Function [280 pts]

Due: 3/26/2025 (Wed.), 11:59 PM

1. Objectives

The goal of this lab is to provide kernel support for asynchronous IPC using callback functions. The design and implementation that follows the principle of isolation/protection by utilizing ROP (return oriented programming) that executes callback functions in the context of the process that registered it upon resuming user code execution on a Galileo backend.

---

2. Readings

- Read Chapters 7-9 of the XINU textbook.

Please use a fresh copy of XINU, xinu-spring2025.tar.gz, but for preserving the output from lab1 identifying authorship, and removing all code related to xsh from main() (i.e., you are

starting with an empty main() before adding code to perform testing). As noted before, main() serves as an app for your own testing purposes. The GTAs when evaluating your code will use their own main() to evaluate your XINU kernel modifications.

3. Run-time modification of return addresses

3.1 Modifying return address of ctxsw() [50 pts]

Code an app, void wrongturn175(pid32 traveler), in system/wrongturn175.c that takes the PID of a process, traveler, that has been context-switched out, and modifies the process's stack so that when traveler resumes running ctxsw() does not return to resched() but jumps to a function, void delphi(), to be coded in system/delphi.c, instead. delphi() outputs "Welcome to Delphi" then enters an infinite loop. To achieve run-time modification of how the traveler process behaves, the process executing wrongturn175() looks up traveler's saved stack pointer prstkptr in the process table, uses its value to find the address in traveler's stack where the return address from ctxsw() to resched() is stored. wrongturn175() overwrites the return address with the function pointer delphi. Your test application, void main3(void), in system/main3.c, creates and resumes two processes back-to-back. The first process, who plays the role of

traveler, executes function, void needarest(void), in system/needarest.c, that outputs "Need a rest" then calls sleepms() to sleep for 500 msec and upon waking up outputs "Rested and ready" in an infinite while-loop. After creating the first process whose PID is stored in a local variable x of type pid32, main3() creates and resumes the second process which executes wrongturn175() whose argument is provided as x. Define a macro, TEST3a, which if set to 0 does not cause main3() to spawn the second process. If TEST3a is set to 1, the second process is created. The two child processes are of equal priority but strictly less than the parent executing main3(). main3() terminates after resuming the second child process.

Note that neither wrongturn175() nor needarest() call delphi(). By wrongturn175() applying ROP to modify the return address of ctxsw() in the stack of the sleeping process traveler, a jump from ctxsw() to delphi() occurs when traver wakes up instead of continuing where it left off. Test and verify that wrongturn175() achieves its objective.

3.2 Modifying return address of sleepms() [30 pts]

In the second version, code void wrongturn175a(pid32 traveler) in system/wrongturn175a.c such that wrongturn175a() modifies

the sleeping process traveler's return address from sleepms() to its caller. In commodity operating systems, modifying the return address of ctxsw() would be very disruptive since traveler would make a detour to delphi() while still in kernel mode. Modifying the return address of sleepms(), a system call, would mean that a detour is undertaken after traveler returns to user mode. This is the scenario relevant to Problem 4.

Note that the location of the return address of sleepms() can be found using the saved EBP values in the run-time stack of traveler. Test and verify that the detour works correctly.

---

4. Asynchronous event handling using callback function [200 pts]

4.1 Overview

This problem concerns implementation of asynchronous receive in XINU using a callback function that the application programmer (i.e., a XINU process) registers with the kernel. If a 1-word message is sent to the receiver, the kernel ensures that the registered callback function is eventually executed in the context of the receiving process. In asynchronous IPC a sender

process must run (i.e., be the current process) to produce a message receive event by the receiving process by sending a message using send(). For our uniprocessor Galileo x86 backend, this implies that the receiver of a message cannot be the current process when a message send event occurs. In general asynchronous I/O, a message may arrive on a network interface (e.g., Ethernet card) whose intended recipient is the current process. We will consider device I/O and other general event related matters separately in lab5.

4.2 Callback function registration

We will implement a system call, syscall registercbk(void (*)), in system/registercbk.c that allows a process to register a callback function (i.e., signal handler in UNIX parlance) with XINU requesting that it be executed after a message receive event occurs in the context of the process's user code (in general, in user mode). For simplicity, we will assume that a callback function must be of type void and accepts no arguments. The function definition is given by

- syscall registercbk(void (* fcbk) (void))

where fcbk is a function pointer to a user callback function. What the callback function does is up to the app programmer. For example, a callback function myrcv() may perform

```
void mymsghandler() {
extern umsg32 mbuffer;

   mbuffer = receive();  // copy message to user buffer
}
```

where mbuffer is a global variable wherein the received message is stored. Although receive() is a blocking system call, it will not block since mymsghandler() is invoked after a message has been received.

An app running main() may register an asynchronous IPC callback function with XINU using

```
if (registercbk(&mymsghandler) != OK) {
   kprintf("Registration failed.\n");
   exit();
}
```

The body of main(), for testing purposes, can be an infinite loop that keeps the CPU occupied, e.g., ALU operations. To track events when receiver and sender processes are running, add kprintf() statements in the callback function and sender code that output the event (send or receive), PID, time stamp (reuse clkcounterfine from lab1), and message being received or sent. For example:

send 7 5000 22

receive 6 5009 22

which specifies that the process with PID 7 sent a message at time 5000 msec containing 22 (an unsigned int), and process 6 received a message at time 5009 containing payload 22. Follow this output format and use the XINUTEST macro to enable/disable test output.

4.3 Detour to callback function

To comform with isolation/protection, when an asynchronous IPC event occurs XINU will run a receiver's callback function in the context of the receiver process when it is scheduled next and resumes execution of its user code. On our uniprocessor Galileo backend machines, a message send event implies that a sender process is running on the CPU and the receiver process is not. When the receiver is context switched in at some point in the future which incurs a delay, small or large, depending on the receiver process's priority, XINU will have modified the receiver process's context so that it does not resume where it left off but first jumps to the registered callback function. Upon completion of the callback function executing the last

instruction, ret, the receiver process jumps to the original return address and resumes executing its synchronous app code.

This detour must occur seemlessly preserving correctness of the synchronous execution of the app code but for a call to its registered callback function that is interleaved asynchronously with the help of the kernel. In XINU where there is no separation between user mode/kernel mode, executing the receiver's callback function in the receiver process's context in user mode will be technically delineated to mean upon executing ret in sleepms() which jumps to the callback function instead of returning to the caller of sleepms(). In operating systems with isolation/protection, instead of the ret instruction the instruction iret from lab2 would cause untrapping from kernel mode to user mode when the detour is made to the callback function. During testing the receiver of IPC will calls sleepms() to cause it to be context-switched out first after which the sender will run and transmit a message. Other cases such as the receiver being context-switched out due to time slice depletion need not be considered.

4.4 Use of ROP for arranging seemless detour

We will utilize ROP to modify the receiver's run-time stack while it is context-switched out so that when the receiver becomes current and is context-switched in its execution makes a seemless detour to the callback function. To do so your implementation should adhere to the following code structure.

A. When send() succeeds transmitting a message (i.e., receiver buffer is empty), XINU checks if the receiver has registered a callback function for asynchronous IPC. If so, XINU modifies the stack of the receiver process which has been context-switched out to manipulate its future run-time behavior. Add a process table field, uint16 prasyncipc, that is initialized to 0 and set to 1 if registercbk() succeeds. Introduce a second process table field, void (* prrecvcbk)(), that stores the function pointer of the receiver's callback function. registercbk() returns SYSERR if a callback function has already been registered. Determine if any other error condition exists that needs to be checked. Upon executing ret in sleepms(), the receiver will jump to helper function managedetour().

B. If the receiver process has registered a callback function, we will use a helper function, void managedetour(void), introduced by the kernel to be coded in system/managedetour.S that after

the detour to the receiver's callback function is complete the receiver resumes executing its synchronous code as if a detour had not been made. That is, despite the reality of there having been a detour registers must be put back to the same state as if no detour had been made. To do so, managedetour() must save and restore registers that by CDECL the callee (i.e., sleepms()) is responsible for.

C. The first step is for send() to modify the stack of the receiver so that when the receiver process wakes up, eventually becomes current, and sleepms() executes ret to return to its caller (i.e., user code), the ret instruction causes a jump to managedetour() instead of returning to the caller of sleepms(). This is analogous to Problem 3.2.

D. In commodity kernels supporting isolation/protection the jump to managedetour() is an untrap operation that switches from kernel mode to user mode. In XINU, it will be a jump to user code. The fact that managedetour() is code injected by the kernel is immaterial in the sense that it is not code that needs to be executed in kernel mode. When sleepms() executes ret and a jump to managedetour() is made, it is managedetour()'s job to save and retore relevant register values, call the callback

function whose function pointer is stored in the process table field prrecvcbk. After the callback function returns to managedetour() it restores the original value of relevant registers, and modifies its stack so that upon executing ret a jump is made to the original return address of sleepms().

E. Upon entering managedetour() its first task is to save relevant registers then call the callback function whose function pointer is stored in prrecvcbk using the call instruction. Upon completion, prrecvcbk will return to managedetour(). The second task is for managedetour() is to restore register values to their saved values then modify its stack so that executing ret will cause a jump to the original return address of sleepms(). Note, by CDECL caller/callee convention gcc will have injected code to restore register values that the caller is responsible for before executing user code following the call to sleepms().

Thus C induces sleepms() in D to jump to managedetour(). managedetour() saves register values, then calls the receiver's callback function and, upon its return, restores the original register values as if a detour not been made, and jumps to the original return address of sleepms() through the actions in E. Problem 3 is an exercise to implement a simple form of ROP

which is expanded and applied to asynchronous IPC in Problem 4.

4.5 Testing and verification

First, use a setup involving one sender and one receiver to gauge correctness of your implementation. Place your application layer test code mysender() in system/mysender.c, myreceiver() in system/myreceiver.c, driver mymain() in system/mymain.c. Second, increase the number of senders to two. Note that the bulk of application layer testing is done in the one sender/one receiver test set-up which includes testing the behavior of registercbk(), backward compatibilty with legacy XINU where if registercbk() is not called send() behaves as before. Please annotate your code as in lab3.

Bonus problem [40 pts]

As an extension of Problem 3.2, suppose instead of terminating after printing "Welcome to Delphi" we want to modify the stack of traveler so that when wrongturn175() returns needarest() continues executing where it left off. The main difference compared to Problem 4 is there is no intermediary

managedetour() who at run-time manages the detour. The detour is completely determined by wrongturn175z() modifying the stack of traveler while it is context-switched out. Describe your solution in lab4ans.pdf. Discuss any shortcomings of your solution. Name the revised implementation wrongturn175z() coded in system/wrongturn175z.c and delphiz() in system/delphiz.c. Implement and test that it works correctly.

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 lab4ans.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-spring2025/compile directory and run "make clean".

ii) Go to the directory where lab4 (containing xinu-spring2025/ and lab4ans.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab4/xinu-spring2025 is your directory structure, go to /homes/alice/cs354

iii) Type the following command

turnin -c cs354 -p lab4 lab4

You can check/list the submitted files using

turnin -c cs354 -p lab4 -v

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

Back to the CS 354 web page