CS 354 Fall 2023

Lab 5: Asynchronous Handling of Events using Callback Function [270 pts]

Due: 11/16/2023 (Thu.), 11:59 PM

1. Objectives

The goal is to provide kernel support for asynchronous event handling using callback functions that preserve isolation/protection when events are triggered by asynchronous events.

---

2. Readings

- Read Chapters 10, 13-14 of the XINU textbook.

Please use a fresh copy of XINU, xinu-fall2023.tar.gz, but for preserving the helloworld() function from lab1 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. Asynchronous event handling using callback function [270 pts]

3.1 Overview

In Problem 5, lab4, we introduced kernel support for run-time detour to a user callback function specified in the argument of dreamvac() when sleepms() returned. Instead of the callback function being called directly by user code, ROP was used by the kernel to modify the return the address of sleepms() to arrange the detour. In this problem we will support asynchronous user callback function execution in response to asynchronous events.

3.2 Callback function registration

Code a system call, syscall cbsignal(uint16 etype, void (* cbf) (void), uint32 val), in system/cbsignal.c that can be used by processes to register a user callback function, cbf, with the kernel. If registration is successful, the kernel will arrange execution of the callback function when the event specified by the first argument etype occurs. To preserve isolation/protection, the kernel will arrange execution of cbf in user mode and in the context of the process that registered the callback function. In XINU, user mode is interpreted to mean when a process returns from a system call or from an interrupt where there are clear dividing lines between kernel code and user code. The third argument, val, is meaningful only for some event types. cbsignal() returns 0 if the specified arguments are valid and the request can be met. It returns SYSERR otherwise. Validity is determined per event type etype described below.

3.3 Alarm event

Registration. If etype is 1 then the kernel is asked to set an alarm event, a timer specified by the third argument val (in unit of msec). We will reuse the millisecond counter, clkcountermsec, from lab1 to support alarm events.

Introduce four new process table fields, uint16 pretype, void (* prcbf1)(), uint32 pralarmtime, uint16 pralarmreg, where pretype and pralarmreg are initialized to 0. When cbsignal(1, myfunc, val) is called, the kernel returns SYSERR if val is greater than 5000. That is, we will not support timers longer than 5 seconds. cbsignal() returns SYSERR if pralarmreg is not 0, meaning that we allow a process to have one outstanding timer alarm only. cbsignal() returns SYSERR if the function pointer, cbf, falls outside the text segment memory region of XINU. Check initialize.c which utilizes the boundary addresses of XINU's text segment.

If the arguments of cbsignal() are verified to be valid, cbsignal() sets pralarmreg to 1 indicating that an alarm event has been registered for the current process. prcbf1 is set to myfunc. pralarmtime is set to clkcountermsec + val.

ROP detour. To reduce coding we will consider one canonical case -- lower half system timer -- where the kernel checks if the current process has an alarm event that has expired for which a detour needs to be arranged. Each time that clkhandler() is invoked in response to XINU's 1 msec clock interrupt, clkhandler() will check if clkcountermsec is greater than or equal to pralarmtime of the current process. If so, a detour needs to be arranged. Instead of jumping to the callback function prcbf1 directly, the kernel will set up the process's stack so executing iret in clkdisp() will cause a jump to system function, void cbfmanager(void), to be coded in system/cbfmanager.S. cbfmanager() is a kernel function that does not need to be executed in kernel mode, hence we will refer to it as a system function. cbfmanager() calls the user callback function, prcbf1, which then returns to cbfmanager(). When cbfmanager() executes ret it jumps to the original return address of clkdisp().

Recall from lab2 that when an interrupt occurs x86 pushes values contained in EFLAGS, CS, EIP onto the current process's stack. To induce cbfmanager() to return to the original return address of clkdisp(), use a different method from Problem 5, lab4, where clkdisp() rearranges its stack -- upon checking that a detour to cbfmanager() needs to be made -- so that it jumps to cbfmanager() upon executing iret. Note that when clkhandler() returns to clkdisp() the current process's stack contains the 8 registers saved by pushal followed by the values of EIP, CS, EFLAGS saved by x86 when a clock interrupt occurred. The first step is to shuffle the values near the top of the stack so that the 8 register values are located below EFLAGS, CS, EIP at the top of the stack. Then savekeep the original return address EIP before overwriting it with function pointer cbfmanager. What it means to "savekeep" is up to you to determine. When iret is executed, the values of cbfmanager, EFLAGS, CS are popped by x86 upon jumping to cbfmanager(). When cbfmanager() starts executing the top of the stack must contain the 8 register values.

One of the reasons we are introducing cbfmanager() and coding it in assembly is because of the need to restore the register values saved by clkdisp() before cbfmanager() returns to the original return address. In Problem 5, lab4, this was not a concern since sleepms(), its caller, and vacation() were all coded in C. That is, per CDECL the caller of sleepms() is responsible for saving/restoring EAX, ECX, EDX, and the callee sleepms() for saving/restoring the rest. When sleepms() jumps to vacation() all the register values would be in the same state as when the caller of sleepms() had called sleepms(). Hence vacation(), per CDECL, will save/retore all register values but for EAX, ECX, EDX which the caller of sleepms() will restore when vacation() jumps to it. For the alarm event in lab5 it is the responsibility of cbfmanager() to restore state before returning to the original return address when executing ret. Also, be mindful of additional details such as whether the ESP value saved by pushal in clkdisp() is still valid or needs to be changed before cbfmanager() executes popal followed by ret.

Discuss in lab5.pdf your method for modifying clkdisp.S and coding cbfmanager.S so that the detour is correctly facilitated, including saving/restoring of register values and updating pralarmreg. The bottom line is correct execution of the interrupted process when cbfmanager() returns (i.e., jumps) to the original return address of clkdisp().

Note: When kernels implement trapped system calls following the traditional approach as in lab2, the system call dispatcher in the lower half needs to perform ROP based detour arrangement similar to clkdisp(). Also, if an alarm expires when a process has been context-switched out, it is the scheduler who decides when the process becomes current. Until such time, execution of the callback function is delayed.

3.4 Asynchronous message receive event

Registration. If etype is 2 then the kernel is asked to invoke user callback function cbf if a message arrives from a sender. In our uniprocessor Galileo x86 backends, for a message to arrive a sender process must be running on the CPU who calls send(). Hence the receiver process cannot be current, i.e., has been context-switched out. The third argument of cbsignal(), val, is ignored. As in the alarm event, function pointer cbf is verified to fall within the text segment of XINU. If etype is not 1 or 2, cbsignal() returns SYSERR. The process table field, uint16 pretype, is set to 2 if a callback function for an alarm event is not outstanding. pretype is set to 3 indicating that both an alarm event and an asynchronous message event are outstanding. When an asynchronous message receive event or alarm event has been handled, pretype must be updated accordingly.

Introduce new process table fields, void (* prcbf2)(), uint16 prmsgreg, where prmsgreg is initialized to 0. prcbf2 is used to store the function pointer cbf. prmsgreg is set to 1 if there is an outstanding asynchronous message receive event. Only one outstanding asynchronous message receive event is allowed. — SYSERR if prmsgreg != 0

ROP detour. To simplify coding, we will consider the receiver calling sleepms() as the canonical case where the receiver process has been context-switched out when the sender runs. Modify send() so that it checks if the receiver has a callback function registered for asynchronous message receive. If so, use the technique from Problem 5, lab4, to locate the return address of sleepms() to its caller in the stack of the receiver process. Modify the receiver's stack so that a detour to the receiver's callback function prcbf2 is arranged. When prcbf2 returns it jumps to the original return address of sleepms(). Thus ROP-based detour arrangement is a reuse of the implementation of Problem 5, lab4, albeit performing stack modification in send() as opposed to wakeup() or in resched() before context-switching in a process.

Update of prmsgreg must await until the receiver process wakes up, becomes ready, and is selected by the scheduler to become current. Only then is prmsgreg updated since the receiver is about to make a detour to its callback function when sleepms() returns. During testing we will ignore the scenario where the receiver wakes up and becomes current without having received a message during sleep. This can be handled by utilizing a sender flag which is not needed for our purposes.

3.5 Testing

Test and verify that your implementation works correctly. Describe in lab5.pdf your test procedure to gauge correctness. Please make sure to adequately annotate your code which will count for 10% of the total score.

Bonus problem [40 pts]

Extend Problem 3.4 so that the canonical case includes an app blocking by calling suspend() on itself, in addition to support for sleepms(). Assume that a second process calls resume() with the PID of the suspended process to unblock it. Put your code in a separate subdirectory bonus/ under xinu-fall2023/. Test and verify that your code 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 lab5.pdf and place the file in lab5/. 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-fall2023/compile directory and run "make clean".

ii) Go to the directory where lab5 (containing xinu-fall2023/ and lab5.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab5/xinu-fall2023 is your directory structure, go to /homes/alice/cs354

iii) Type the following command

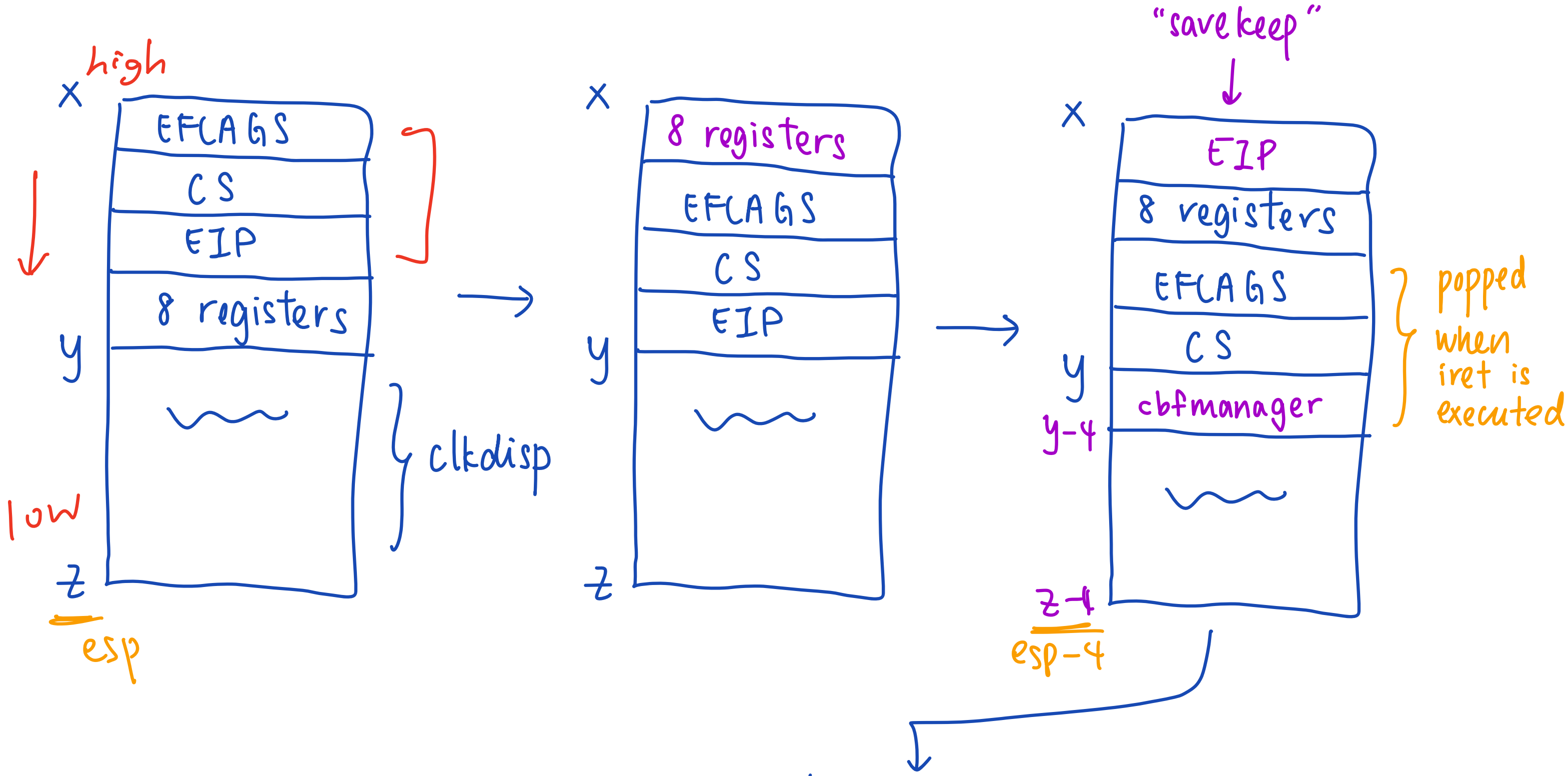
turnin -c cs354 -p lab5 lab5

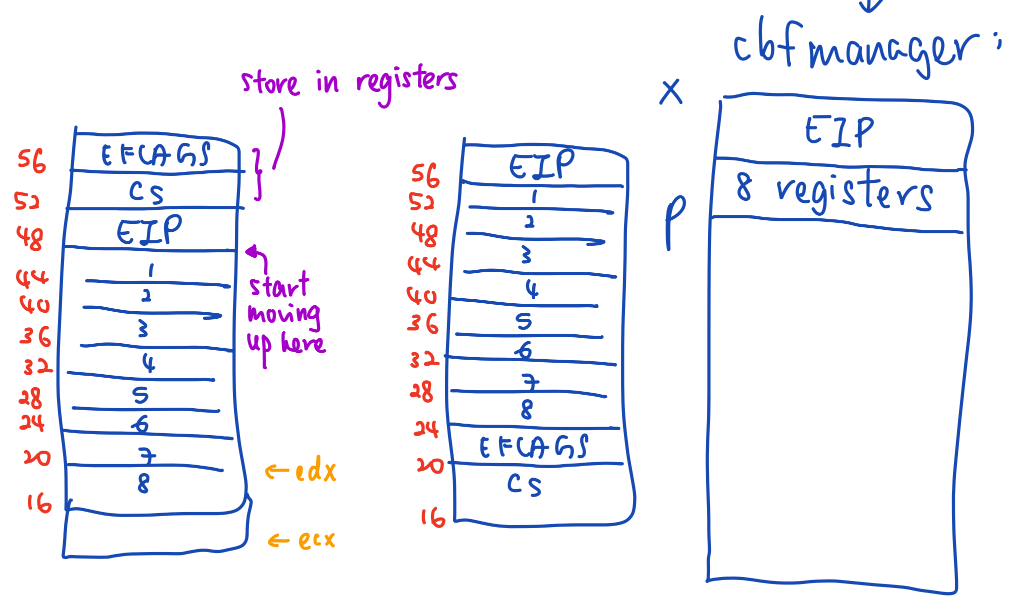
You can check/list the submitted files using

turnin -c cs354 -p lab5 -v

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

Back to the CS 354 web page





call ret in cbfmanager instead of iret bc we dh CS, EFLAGS