#### CS 354 - Spring 2018

#### Lab 6: Signal Handling Subsystem and Garbage Collection (340 pts)

**Due: 04/17/2018 (Tue), 11:59PM**

## 1. Objectives

The first part of the lab concerns a generalization of the techniques utilized in Lab 5, Problem 4, to support a signal handling subsystem in XINU similar to Linux/UNIX. The second part concerns garbage collection in XINU that reclaims memory dynamically allocated to processes when they terminate even if they omitted to perform freemem().

## 2. Readings

Read Chapters 9, 10, and 13 of the XINU textbook.

## 3. Signal handling subsystem (200 pts)

A signal handling subsystem such as the ones provided by Linux/UNIX kernels supports user control of asynchronous events (e.g., arrival of packets, timer expiration, memory access violations). This is achieved by exporting interrupt driven events as "signals" to processes. They register callback functions (i.e., "signal handlers") with a kernel that are executed when certain signals are triggered (i.e., raised). For example, the default action of Linux/UNIX when a process seg-faults is to terminate the offending process. A process, by registering a signal handler for the signal SIGSEGV, can override the default action by requesting that the kernel execute the signal handler which may perform a number of useful actions such as dumping program variables for debugging purposes. Another example is execution of callback functions when timers expire such as in the case of clients contacting a server. A timeout value, say, 3 seconds is set, and if the server does not respond within this time window a follow-up action is undertaken such as retransmitting the request. This is done by registering a signal handler for the signal SIGALRM.

In this generalized asynchronous event management setting, asynchronous IPC with callback function can be viewed as a special case where the event in question -- which is not triggered by a hardware interrupt -- is message transmission from a sender process to a receiver process. As in Problem 4 of Lab 5, isolation/protection must be preserved by running signal handlers in user mode. In XINU we continue to follow a design that is compatible with kernels such as Linux/UNIX and Windows that support isolation/protection by executing the signal handler at run-time just before the process that registered the signal handler resumes execution. In particular, just after interrupts are reenabled. A signal handling subsystem is one of the most advanced and useful features that modern kernels provide to enable app programmers build state-of-the-art applications. We will impart such capability to XINU.

Add a system call to register an asynchronous callback function (i.e., signal handler) defined as

* syscall sigcbreg( uint16 ssig, int (\* fnp) (void), uint32 tmarg )

ssig specifies a signal (an unsigned 16-bit integer) that is defined in xinu\_signal.h under include/. We will support three signals:

* #define   XSIGRCV   20
* #define   XSIGCHL   21
* #define   XSIGXTM   22

The first signal XSIGRCV corresponds to asynchronous IPC with callback function in Problem 4 of Lab 5, which is similar to SIGPOLL/SIGIO in Linux/UNIX. In Linux/UNIX, SIGPOLL/SIGIO are also used for asynchronously handling packet arrivals on network interfaces. Start by changing cbreg() in Problem 4 of Lab 5 to sigcbreg() which is a straightforward extension requiring minimal change. The second argument, fnp, is a function pointer to a user signal handler (i.e., callback function) as in cbreg() of Lab 5. The third argument tmarg conveys additional information for signal XSIGXTM. It is ignored when the signal handler corresponds to XSIGRCV or XSIGCHL.

The second signal XSIGCHL corresponds to SIGCHLD in Linux/UNIX where a parent process is notified if one of its children terminates. This is an integral part of concurrent client/server programming. The XINU process table maintains the prparent field where the PID of a process's parent (the process who created it using create()) is stored. XINU uses send() to inform the parent if a child terminates. You are extending it so that parent processes can perform asynchronous actions when children terminate. The third argument tmarg is ignored in the case of XSIGCHL. To enhance XINU's capabilities for supporting concurrent client/server apps, add a new blocking system call, pid32 childwait(void), that returns the PID of a child process that has terminated to its parent process. Put it in childwait.c under system/. childwait() puts the calling parent process in a new blocking XINU state PR\_CHLDWAIT. Define it in process.h with value 21. Unlike in Linux/UNIX, if a child process terminates and its parent has not called childwait() (synchronously through childwait() or asynchronously through a XSIGCHL signal handler) we will not make it into a zombie process (a process whose resources have been released but which hangs around in the process table hoping that its parent calls childwait()). If a parent process calls childwait() after all its children have terminated, childwait() returns SYSERR.

The third signal XSIGXTM is raised by XINU when the wall time of a process -- the total time in sec that has elapsed since a process was created -- exceeds a threshold given by the third argument tmarg of sigcbreg(). Unlike SIGXCPU in Linux/UNIX which represents the CPU time consumed by a process, XSIGXTM represents its wall time. For example, if

if (sigcbreg(&wtmcallback, XSIGXTM, 3) != OK) {

kprintf("wall time callback function registration failed\n");

return 1;

}

is called by an app process, the kernel raises the XSIGXTM signal when the app process's lifetime has exceeded 3 seconds. Add wall time (i.e., lifetime) monitoring of XINU processes by adding a field, uint32 prstarttime, to procent which is set to clktime when a process is created. prstarttime is in unit of seconds. XINU's clock interrupt handler must be modified so that exceeding a wall time specified by the third argument tmarg is detected. The clock interrupt handler need only perform this check for the current process if it has registered a callback function for XSIGXTM. This implies that a callback function for XSIGXTM registered by a process that sleeps for an extended period will only execute after the process wakes up and is selected by XINU's scheduler to be run next. As we will find when we discuss clock and time management in modern kernels, timer events in kernels are approximate. The same is true for XINU.

For all three signals, if the system call sigcbreg() is invoked multiple times, the latest call supersedes earlier calls. That is, the latest registered callback function overwrites earlier ones. The callback function for a signal may be unregistered through

* syscall sigcbunreg( uint16 ssig )

where ssig specifies a signal. Both sigcbunreg() and sigcbreg() return OK if they succeed and SYSERR if they fail (i.e., encounter invalid conditions). When callback functions are registered for two or more signals, execution of callback functions should follow the same order as the sequence of signals raised at run-time of XINU on our uniprocessor backends.

To support all three signal handlers, introduce four new fields in the process table structure procent:

bool8 prhascb1; /\* Nonzero iff XSIGCHL callback function has been registered \*/

int (\* fptr1) (); /\* Pointer to XSIGCHL cb function if one has been registered \*/

bool8 prhascb2; /\* Nonzero iff SIGXTM callback function has been registered \*/

int (\* fptr2) (); /\* Pointer to SIGXTM cb function if one has been registered \*/

For backward compatibility, the fields prhascb and fptr from Problem 4, Lab 5, remain the same and denote a Boolean flag and function pointer for XSIGRCV.

An issue that has not been considered in Problem 4, Lab 5, is how to manage back-to-back invocation (i.e., raising) of the same signal. For example, a process may be running a signal handler for XSIGRCV but is context-switched out before the handler returns. The newly scheduled process sends a message to the context-switched out receiver process which, when context-switched in sometime in the future, runs the same handler again which causes two executions of the same signal handler to be interleaved. This can result in unintended side effects including corruption of data structures. You do not need to implement a solution to this problem for this lab. However, describe how you would go about addressing this issue in the context of XINU in Lab6Answers.pdf. Your design should be detailed enough so that it points to a straightforward implementation.

Test your XINU implementation with signal handling support by performing the test cases of Problem 4, Lab 5, for XSIGRCV. This confirms that asynchronous IPC works as before in the extended signal handling framework. Create test cases that allow evaluation of correct functioning of XSIGXTM and XSIGCHL signals. In the latter, include a test app that reflects modern concurrent client/server apps where the parent process either calls childwait() (similar to wait() in Linux/UNIX) or uses XSIGCHL to manage terminating child processes asynchronously. Include test cases that evaluate handling of multiple signals. Discuss your XINU modification, implementation, and results in Lab6Answers.pdf. Following XINU's default convention, all system calls should be stored in .c files prefixed by their name and placed in system/.

## 4. Memory garbage collection (140 pts)

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 freed if a process explicitly deallocates memory by calling freemem() which is voluntary. That is, when a process terminates any dynamic memory that was allocated but has not been freed remains allocated (examine kill() which is called to terminate a XINU process). Even when an application programmer ardently tries to free allocated memory before exiting, programming mistakes and bugs that prematurely terminate a process may result in build-up of memory garbage.

Garbage collection systems aim to free allocated memory -- while a process is still running -- when it can be ascertained that a process does not need the memory anymore, i.e., all references/pointers to the memory have been deleted. We are tackling a more modest problem of ensuring that all memory allocated to a process through getmem(), whether freed or not, are reclaimed by XINU when the process terminates. This eliminates memory leakage by injecting garbage collection support inside the kernel. To do so, XINU must track dynamic memory allocation and deallocation on a per-process basis and return any unfreed memory to the free list when a process terminates.

Design and implement garbage collection support in XINU by modifying the system calls, getmem() and freemem(), and other relevant parts of the kernel that eliminates memory leakage. Test your garbage collection enabled XINU kernel on test cases where app programs omit freemem() system calls but garbage collection ensures that the unfreed memory is reclaimed when processes terminate. Verifying that your implementation works correctly requires careful bookkeeping. Describe the design of your implementation, how you go about establishing correctness, and discuss your test results in Lab6Answers.pdf.

## Bonus problem (30 pts)

As an extension of Problem 3, devise a fourth signal that is meaningful to provide to app programmers. Define it as XSIGMY with value 23. Describe in Lab6Answers.pdf its motivation and meaning. Implement and test to show that XSIGMY support works correctly.

### Turn-in Instructions

Before submitting your work, make sure to double-check the [TA Notes](http://www.cs.purdue.edu/homes/ogg/cs354_spring2018.html) to ensure that additional requirements and instructions have been followed.

*Electronic turn-in instructions:*

        i) Go to the xinu-spring2018/compile directory and do "make clean".

        ii) Go to the directory of which your xinu-spring2018 directory is a subdirectory. (NOTE: please do not rename xinu-f all2018, or any of its subdirectories.)

                e.g., if /homes/joe/xinu-spring2018 is your directory structure, go to /homes/joe

        iii) Type the following command

                turnin -c cs354 -p lab6 xinu-spring2018

You can check/list the submitted files using   
  
turnin -c cs354 -p lab6 -v

***Important: The TAs will be replacing your main.c when testing your code. Please make sure not to declare variables inside main.c that will impact compilation. Please provide comments inside your code so that its function and flow can be conveyed to the reader. Turn off all debugging output before you submit your code.***