CS 354 Spring 2025

Lab 5: Memory Garbage Collection and Asynchronous Lower Half Event Management

[320 pts]

Due: 4/16/2025 (Wed.), 11:59 PM

1. Objectives

The objectives of this lab are two–fold: first, implement memory garbage collection so that when a process terminates any memory not freed by the process is returned to the kernel, and, second, implement asynchronous event handling for general kernel events, in particular, alarm timers handled by the lower half.

---

2. Readings

- Read Chapters 10, 13–14 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. Memory garbage collection [160 pts]

3.1 Overview

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. 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 that is not returned to the free list upon termination. This a form of garbage memory in XINU. In general, garbage memory refers to unused memory at run-time that garbage collection tools aim to reclaim to free up memory.

We are tackling a more modest problem of ensuring that all memory allocated to a process through getmem(), whether freed by a process 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 through kill().

3.2 Per-process memory block list

Design and implement garbage collection support in XINU by modifying the system calls getmem(), freemem(), kill(), and other relevant parts of kernel code to eliminate memory leakage. For simplicity, assume that freemem() is always called by app code so that its first argument is a pointer returned by getmem() and the second argument is the memory size requested by getmem().

That is, freemem() behaves similar to free() in Linux where it takes a memory pointer as its sole argument. Add a new field to the process table

```
   struct perprocmem *prheapbeg;       /* Header node of allocated
memory list */
```

where the struct perprocmem

```
   struct perprocmem  {
        uint32  memsize;               /* Size of memory allocated
(multiple of 8 bytes) */
        char    *memptr;               /* Ptr to in-use memory */
        struct  perprocmem  *memnext;   /* Ptr to next list element
*/
   };
```

is used to keep track of in–use (i.e., allocated) memory blocks. Define perprocmem in new header file include/heapmem.h. prheapbeg points to a list that keeps track of all memory blocks allocated to a process using getmem() but not freed using freemem(). prheapbeg is NULL if no heap memory has been allocated to a process. Otherwise, prheapbeg–>memptr contains the address of the first in–use memory block and prheapbeg–>memnext points to the next element in the memory block list. memsize specifies the size of the allocated memory block rounded up to be a multiple of 8 bytes per XINU convention. This per–process memory block list is a priority list whose elements are sorted in increasing order of memptr. Note that the per–process linked list to track heap usage is itself

allocated using getmem(). The main distinction is that the kernel data structure is freed by XINU when a process terminates along with any unfreed per-process heap memory.

Code an internal kernel function with function prototype

```
void kheapinsert(struct perprocmem *ptr, char *memptr, uint32
msz);
```

in system/kheapinsert.c where ptr points to the header node of a process's heap memory list, memptr is a pointer to a memory block to be allocated, and msz its size. Write a counterpart of kheapinsert()

```
void kheapextract(struct perprocmem *ptr, char *memptr);
```

in system/kheapextract.c that deletes a list element whose memory block is pointed to by memptr.

Memory for a process's heap memory list is dynamically allocated using getmem() in kheapinsert(). Since a process's memory heap list is part of kernel memory, it is not included in the process's heap list. kheapinsert() and kheapextract() are part of kernel code assumed to be correctly coded. To exclude memory allocated for a process's heap list, getmem() needs to know if it was called from kheapinsert(). In a similar vein, freemem() needs to know if it was called from kheapextract(). To do so, define a global variable, uint16 kheapflag, that is initialized to 0. kheapflag is set

to 1 when getmem() is called from kheapinsert(). Similarly for freemem() from kheapextract(). If so, getmem() and freemem() reset the flag to 0 before returning.

3.3 Freeing user heap memory and kernel perprocmem memory upon termination

Modify the kill() system call (note that in XINU kill() is used both as a system call as well as an internal kernel function due to lack of isolation/protection) so that when a process is terminated any unfreed heap memory is reclaimed by calling freemem(). After all unfreed user heap has been reclaimed, dynamic memory associated with kernel data structure perprocmem must be deallocated using freemem().

3.4 Testing

Evaluate 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. You do not need to consider the scenario where the current process calls kill() to terminate another process. Discuss your test cases in lab5ans.pdf. Assume that freemem() will always be correctly invoked, i.e., with the first

argument being a pointer returned by getmem(). Please make sure to adequately annotate your code.

---

4. Asynchronous management of lower half kernel events [160 pts]

4.1 Overview

In lab4 we introduced kernel support for asynchronous IPC using callback function where the event was receipt of a message by a process. Whereas lab4 concerned upper half changes to XINU, in lab5 we will extend asynchronous management of kernel events to include the lower half.

4.2 Callback function registration for timer alarm

A common service provided by commodity operating systems such as Linux and Windows are timer alarms where a process requests that a timer alarm event be set for a future time. For example, calendar apps that implement a reminder feature and client/server apps that retransmit potentially lost message make use of timer alarms. We will introduce a timer alarm feature

which we will refer to as XALARM. We will implement a new system call

```
syscall xinualarm(uint32 alarmmsec, void (* cbf) (void));
```

in system/xinualarm.c, which is used to register a callback function cbf to be invoked with the help of XINU when a timer value (in msec) specified by the first argument alarmmsec has expired. xinualarm() returns SYSERR if alarmmsec is 0 or exceeds 15000 (an artificial cap of 15 seconds is placed for testing purposes). xinualarm() will also perform a basic sanity check of the function pointer supplied in the second argument by checking that cbf falls within the text segment of XINU. Confer initialize.c to determine how to access the start and end addresses of XINU's text segment within which all XINU functions mapped by gcc must lie.

Add a new process table field, void (* prcbf)(), where the second argument of xinualarm() is stored. A new process table field, uint32 pralrmcounter, is initialized to 0 when a process is created, which indicates that a XALARM callback function has not been registered. xinualarm() updates pralrmcounter by adding the first argument alarmmsec to clkcounterfine (ported from lab1) which makes pralrmcounter an absolute (not relative) timer. We

will disallow xinualarm() being called twice. xinualarm() returns 0 if all error checks pass, otherwise SYSERR.

4.3 Checking timer alarms

The kernel will check if a registered alarm timer has expired for the current process only. Thus handling of asynchronous alarms can be arbitrarily delayed if the process that registered it has low priority and other processes take precedence. By default, commodity kernels such as Linux and Windows provide no assurances as to when an asynchronous event will be attended to. Modify clkhandler() so that it checks if the current process has a callback function registered for XALARM and compares clkcounterfine with the process's pralrmcounter to determine if the timer has expired (i.e., pralrmcounter is greater than or equal to clkcounterfine). If the alarm timer has expired, clkhandler() sets a global variable, uint16 makedetouralrm, which is initialized to 0 in clkinit.c to 1. makedetouralrm acts as a flag for clkdisp.S to arrange a detour the registered callback function of the current process whose alarm timer has expired.

4.4 Arranging for callback function execution

Arranging for callback function execution follows the design principle that a callback function must be executed in user mode in the context of the process that registered it. In XINU where there is no user/kernel mode separation, we abide by the principle that a jump to the callback function is made when kernel code (upper half or lower half) crosses over into user code. For clkdisp.S which is part of XINU's lower half, the boundary between kernel code and user code is when iret is executed. As we know from lab2, this is the point where untrapping is performed for system calls, and, in the present case, for clock interrupt handling in the lower half.

For testing purposes we will limit events that lead to a detour to two scenarios. In the first case, the current process is a process that registered a callback function for XALARM whose expiration has been detected by the lower half as a result of a clock interrupt. In the second case, a process that registered an XALARM callback function has been context-switched out due to depleting its time slice. When it becomes current in the future where ctxsw() returns to resched() which then returns to clkhandler(), occurrence of the XALARM event must be checked by clkhander() before returning to the clock interrupt dispatcher.

Unlike in lab4 where we utilized a system function, managedetour(), that we jumped to upon executing ret in sleepms(), we will not make use of an intermediary but jump directly from clkdisp.S to the registered callback function by executing iret. When the callback function completes and executes ret, it will jump back to clkdisp.S which executes ret to return to the original return address of clkdisp.S had it not made a detour. When a clock interrupt has occured, legacy clkdisp.S before it executes iret has the following content at the top of the current process's runtime stack:

```
... EFLAGS CS EIP EAX ECX EDX EBX ESP EBP ESI EDI
```

with the register %esp pointing to the top of the stack which contains the saved value of EDI. If the global variable makedetouralrm is 1 then clkdisp.S rearranges the top of the stack so that it becomes

```
... EIP EFLAGS EAX ECX EDX EBX ESP EBP ESI EDI restoreandret EFLAGS1 CS CBF
```

where EFLAGS1 is an initial EFLAGS configuration analogous to one used by ctxsw() for context-switching in a process that runs for the first time. CBF is the callback function pointer stored in prcbf. Thus executing iret causes x86 to pop CBF, CS, EFLAGS1 atomically and jump to CBF crossing from kernel code (i.e., code that must be executed in kernel mode in kernels supporting

isolation/protection) to user code (i.e., code that can be executed in user mode). If callback functions are coded in C, per x86-32 CDECL gcc will save EBP (since we instruct in Makefile to do so) and perform additional tasks that may further modify the stack such as allocating space for local variables of the callback function. Before executing ret, the callback function will restore EBP (and EBX if it has been modified). This will cause a jump back to code at restoreandret in clkdisp.S where you will execute instructions to restore the 8 general purpose registers and EFLAGS before executing ret to return to the original return address of clkdisp.S. Even though the code at restoreandret following iret in clkdisp.S is part of kernel code, it is not code that needs to be executed in kernel mode. Similar to managedetour() in lab4, the code following iret in clkdisp.S is effectively user code.

When the callback function returns to clkdisp.S, %esp will point to the top of the stack which contains the saved value EDI. clkdisp.S executes popal to pop the 8 registers (the callback function restoring EBP and EBX was immaterial) then executes popfl to restore the original saved EFLAGS bits, and, lastly, executes ret to jump to the original return address of clkdisp.S.

4.5 Testing

Test and verify that your implementation works correctly. Discuss in lab5ans.pdf your method for establishing correctness which includes the various conditions under which xinualarm() may fail. Make productive use of the callback function such as printing "Ring, ring" and clkcounterfine. Please make sure to adequately annotate your code.

Bonus problem [60 pts]

(a) [30 pts] XINU uses getstk() to allocate stack memory from the same free memory list memlist as user heap memory but for using last–fit policy (as opposed to first–fit used by getmem()). For correct operation of XINU, does it matter whether getstk() uses last–fit or first–fit? Discuss your reasoning in lab5ans.pdf. Modify getstk(), coded as getstkf(), in system/getstkf.c which uses first–fit instead of last–fit. Based on your answer above, describe how you would go about showing that getstkf() works correctly or incorrectly. Include your test cases involving multiple processes where each makes nested function calls to help

support your answer empirically. Discuss the test cases and their results in lab5ans.pdf.

(b) [30 pts] Modify the system call interface of Problem 4 so that the task performed by xinualarm() is replaced by two system calls, xalrmreg(void (* cbf) (void)) in system/xalrmreg.c and xalrmset(uint32 alarmmsec) in system/xalrmset.c. The first system call, xalrmreg(), carries out the callback function registration component of xinualarm() while setting pralrmcounter to 0. Thus calling xalrmreg() primes or prepares the kernel without instituting a detour. The latter is accomplished by calling xalrmset() whose argument is used to set pralrmcounter to a positive value. Consider what additional error checking needs to be performed by xalrmset() so that it works in concert with xalrmreg(). In this version we will allow xalrmset() to be called multiple times where the most recent call overwrites pralrmcounter set by previous calls. Although not necessary, enhanced modularity from splitting callback function registration and activation is an approach found in commodity operating systems. Use the test setup from 4.5 adapted to the modified system call interface to assess correctness of your implementation.

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 lab5ans.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-spring2025/compile directory and run "make clean".

ii) Go to the directory where lab5 (containing xinu-spring2025/ and lab5ans.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab5/xinu-spring2025 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