# CS 354 Fall 2023

## Lab 4: Blocking Send, ROP and Run-time Modification of Process Behavior (300 pts)

## Due: 11/02/2023 (Thu.), 11:59 PM

## 1. Objectives

The objectives of this lab are, one, to enhance XINU's IPC services by adding system call support for blocking send(), two, implement ROP (return oriented programming) techniques that allow run-time modification of program execution.

---

## 2. Readings

- Read Chapters 6 and 7 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. Blocking message send [120 pts]

This problem concerns the implementation of a blocking version of send(), call it sendb(), to be coded in system/sendb.c. sendb() has the same function prototype as send(). If the receiver's 1-word message buffer is empty, sendb() behaves the same way as send(). If the receiver's buffer is full, sendb() blocks by being context-switched out to free the CPU to be used by a ready process. Before doing so, we will insert the sender in a per-receiver FIFO queue until it is its turn to become unblocked. We will also temporarily store the sender's message in the same FIFO queue.

### 3.1 Sender actions

Define a new process state, PR_SNDB, with value 14 in include/process.h which specifies that a process is blocked trying to send a message. Define a new process table field, struct blockedsenders *prsendbqueue1, and that points to the first element of a FIFO queue of blocked senders attempting to send a message using sendb() to the same receiver. If prsendbqueue1 is NULL then there are no blocked senders. Define a new process table field, struct blockedsenders *prsendbqueue2, and that points to the last element of a FIFO queue of blocked senders. Its value is NULL if there are no elements in the queue. The FIFO queue is implemented as a linked list whose elements are of type

```
struct blockedsenders {
  pid32 senderpid;              // PID of blocked sender
  umsg32 sendermsg;             // message of the sender
  struct blockedsenders *next;  // next element pointer
}
```

The next field is NULL if it's the last element in the list. Code a kernel function, int32 enqueuesndb(pid32 preceiver, pid32 psender, umsg32 pmessage), in system/enqueuesndb.c that inserts a 1-word message pmessage from sender psender into the FIFO queue of receiver preceiver. enqueuesndb() returns 0 upon success, -1 if it fails. Failure will happen if there is no more kernel memory to create a new element which will not be a concern in our benchmark tests. sendb() returns SYSERR if enqueuesndb() fails.

When implementing enqueuesndb() use XINU's getmem() function to dynamically allocate memory. getmem() takes an argument of type uint32 that specifies the number of bytes requested. getmem() returns a pointer to the first byte upon success, NULL otherwise. If getmem() fails, enqueuesndb() returns -1.

### 3.2 Receiver actions

We need to modify receive() so that it works with sendb() while being backward compatible with send(). receive() will work as before but for the case where there is a message in the receiver's 1-word buffer. In addition to returning the message to the caller, receive() checks if there are blocked sender processes by inspecting prsendbqueue1. If not NULL, receive() calls kernel function, void dequeuesndb(pid32 preceiver, struct blockedsenders *dummystore), to be coded in system/dequeuesndb.c to retrieve the PID and message of the sender at the front of the FIFO queue. The first argument specifies the receiver's PID. For our limited usage it is not necessary since dequeuesndb() will be called by the current process whose PID is available in the global kernel variable currpid. The second argument, dummystore, points to a local variable of caller receive(), struct blockedsenders dummyvar, whose fields, senderpid and sendermsg, will be used by dequeuesndb() to convey the sender's PID and message to receive(). receive() copies the sender's message to the receiver's message buffer prmsg and sets the flag prhasmsg to indicate that there is a message. receive() changes the sender's state from PR_SNDB to PR_READY and inserts the sender into the readylist by calling ready().

When coding dequeuesndb() use XINU's freemem() function to free the linked list element of the dequeued sender process. There should be no condition under which dequeuesndb() fails unless there are bugs in other parts of kernel code. Hence its return type is void.

### 3.3 Testing

Create test cases whose output indicate correct functioning of the blocking message send extension of kernel services. For example, senders may print sender-specific 1-word messages before calling sendb() and a receiver may print a received message. Reuse the global variable from lab1, uint32 clkcountermsec, to output timestamps along with the messages. Careful crafting of the test cases is required to verify for yourself that your revised kernel is working correctly, in addition to demonstrating to others. Your modified kernel should be backward compatible with legacy send(). That is, if sendb() is not invoked, the modified receive() works exactly as before. In lab4.pdf describe how you set up your test cases and their rationale.

---

## 4. Modifying return address at run-time [60 pts]

Code an app, void wrongwayhome(pid32 vic), in system/wrongwayhome.c that takes the PID of a process, vic, that has been context-switched out, and modifies the process's stack so that when vic resumes running ctxsw() does not return to resched() but jumps to a function, void curveball(), to be coded in system/curveball.c, instead. To achieve run-time modification of how process vic behaves, the process executing wrongwayhome() looks up vic's saved stack pointer prstkptr in the process table, uses its value to find the address in vic's stack where the return address from ctxsw() to resched() is stored. Then wrongwayhome() overwrites the return address with the function pointer curveball. Code curveball() so that it outputs a message, "This is a curveball", before calling exit() to terminate process vic. From a security perspective, the process running wrongwayhome() may be viewed as an attacker and the process vic a victim.

The process vic is spawned by main() by calling create(). It executes function, void abc(void), to be coded in system/abc.c where abc() just calls sleep(2). After creating (and resuming) a process to run abc(), main() creates (and resumes) a process to run wrongwayhome().

Note that neither abc() nor wrongwayhome() call curveball(). Due to wrongwayhome() modifying a return address in the stack of the process vic, a jump from ctxsw() to curveball() occurs when vic wakes up after calling sleep(2) and is eventually chosen by the scheduler to become current. Invocation of curveball() is arranged at run-time by utilizing ROP. Test and verify that wrongwayhome() works correctly.

↳Return oriented programming

---

## 5. Utilizing ROP to arrange a detour [120 pts]

vacation()

Code a new system call, syscall dreamvac(void (* vacation) (void)), in system/dreamvac.c that takes a function pointer, vacation, as argument. When a process calls dreamvac() it asks that the kernel arrange for a detour to dreamvac() when it returns from sleepms(). In general, vacation() is user code referred to as a callback function, and a detour to a callback function may be requested to occur for various events such as a timer expiring, a message arriving, a child process terminating. If dreamvac() is called multiple times with different function pointers, the function pointer should be updated with the most recent one. As discussed in class in the setting of asynchronous IPC with callback function, the technical problem that needs to be handled is executing the callback function in user mode (in XINU when kernel code returns to user code) in the context of the process that called dreamvac(). In Problem 5 we will consider a special case where the process that called dreamvac() is going to call sleepms(), and a detour to vacation() must be arranged by XINU at run-time when sleepms() returns.

Whereas in Problem 4 a process executing wrongwayhome() used ROP to affect the behavior of another process, in this problem the kernel is asked to do so. Furthermore, whereas in Problem 4 curveball() terminated the process whose run-time behavior was dynamically changed, XINU will arrange for sleepms() but make a detour to vacation() and not return to the caller directly.

Another important difference is that the process who called dreamvac() needs to eventually return from sleepms() after making a detour and not terminate. In Problem 4 curveball() terminated the process. To achieve a detour, the kernel will modify the stack of the sleeping process after it calls sleepms() and is context-switched out so that sleepms() does not return to its caller but jumps to vacation() which outputs the message "On vacation". Note that sleepms() called resched() which then called ctxsw() which caused the process to be evicted. There are three return addresses on the stack: the return address RET1 of sleepms() to its caller, the return address RET2 of resched() to sleepms(), and the return address RET3 of ctxsw() to resched(). XINU needs to modify RET1 with the function pointer, vacation, to cause a jump to vacation() when sleepms() executes ret. To find the address in the stack where RET1 is stored, make use of the fact that we have configured gcc to utilize the base pointer EBP when generating machine code. The above is only the first part of the detour.

The second part of the detour involves making additional modifications to the stack so that when vacation() executes ret it jumps to the original return address of sleepms(), RET1. Hence the run-time stack needs to be rearranged so that when the process wakes up ctxsw() returns to resched(), resched() returns to sleepms(), sleepms() returns to vacation() (i.e., the stack makes it appear as if vacation() had called sleepms()), and vacation() returns to the caller of sleepms(). Consider how the content of the stack must be modified to produce this sequence of events.

Describe in lab4.pdf your design for implementing dreamvac() and how the kernel will modify the stack layout of the process calling dreamvac() after it has been context-switched out by calling sleepms(). Use sleepms(500) during testing. Note that dreamvac() only updates kernel data structures that you newly introduce so that XINU can make needed modification to the stack when the process has been context-switched out. Specify in lab4.pdf what additional data structures you use to provide kernel support for the detour. Test and verify that your implementation works correctly.

---

## Bonus problem [25 pts]

The second part of the detour can be implemented using a technique where vacation() actively manipulates the run-time stack so that a return to the caller of sleepms() is achieved. Since vacation() is user code, modifying vacation() is not an option. Instead, in the first step of the detour the kernel overwrites RET1 to jump to a wrapper function, detourwrapper(), that can be executed in user mode. detourwrapper() calls the callback function vacation() which returns to detourwrapper(). detourwrapper() manipulates the stack so that when it executes ret detourwrapper() jumps to the original return address of sleepms(), RET1. Describe in detail in lab4.pdf how you would code detourwrapper() to perform the second part of the detour. There is no need to implement your solution.

*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:

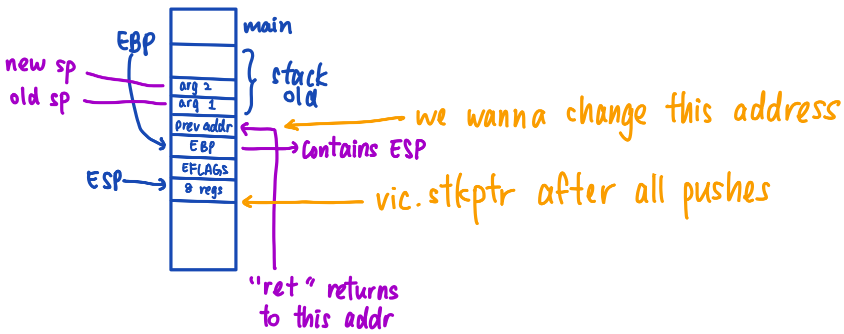
When implementing code in the labs, please maintain separate versions/copies of code so that mistakes such as unintentional overwriting or deletion of code is prevented. This is in addition to the efficiency that such organization provides. You may use any number of version control systems such as GIT 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.

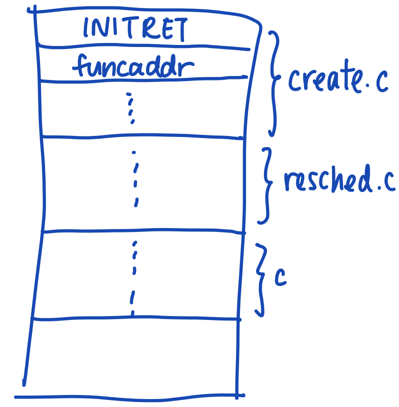
If you are unsure what you need to submit in what format, consult the TA notes link. If it doesn't answer your question, ask during PSOs and office hours which are scheduled M-F.

Specific instructions:

4)




Stack for vic



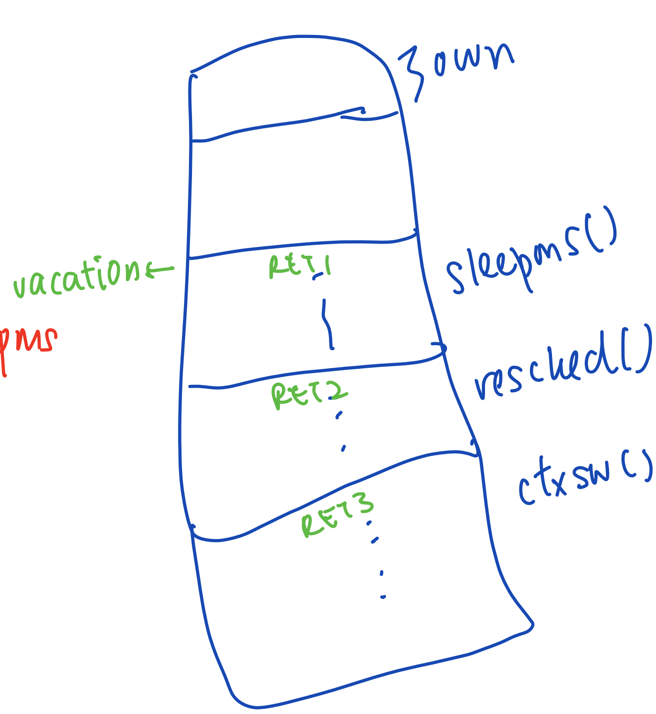

5) * stacktrace() dumps out stack content

sleepms() → resched

//modify stack

wake up in vacation()

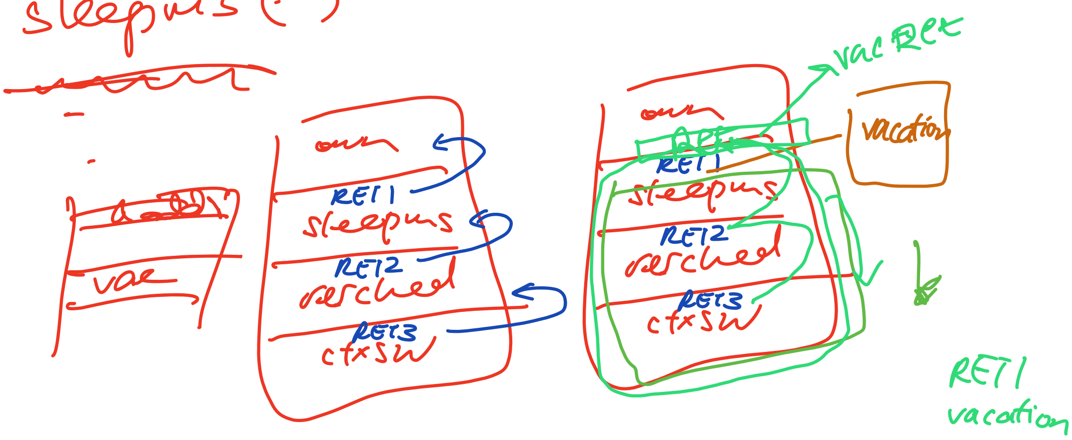
① overwrite return address in sleepms

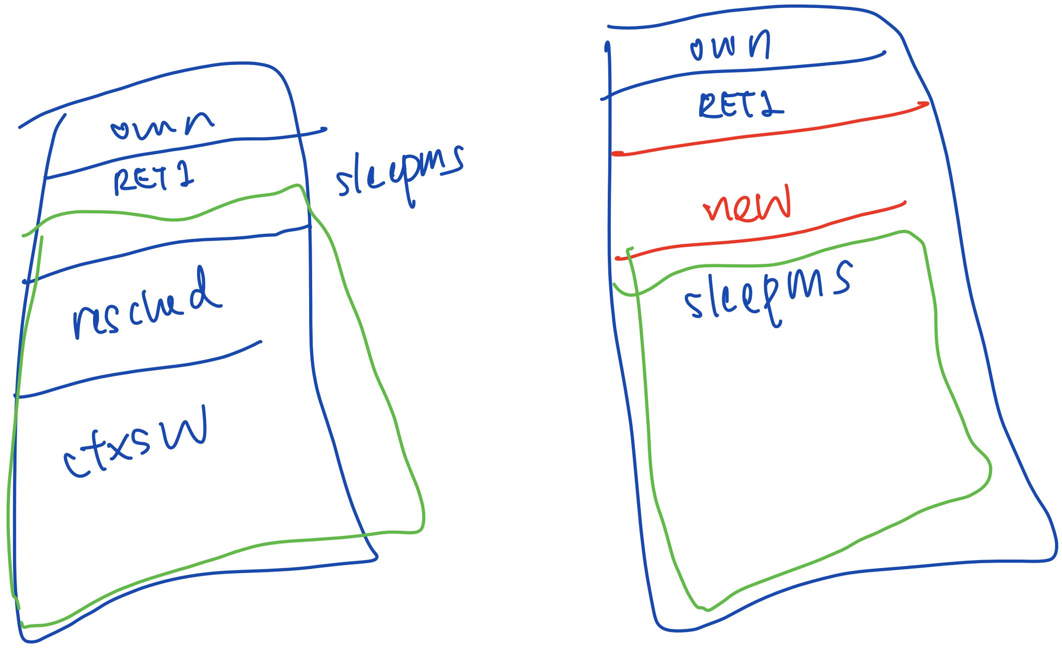



Proc (&addr)

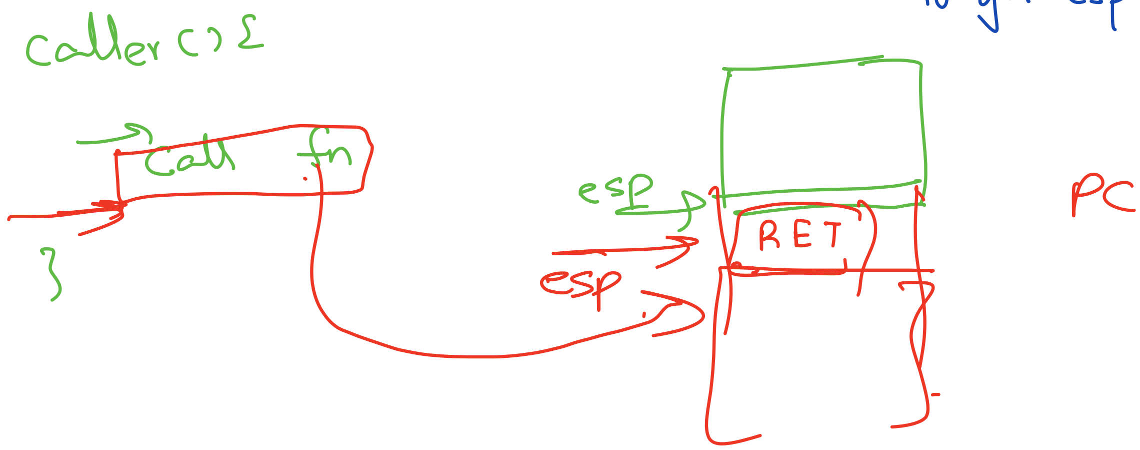
dreamvac(vac) — Register vacation

sleepms( )

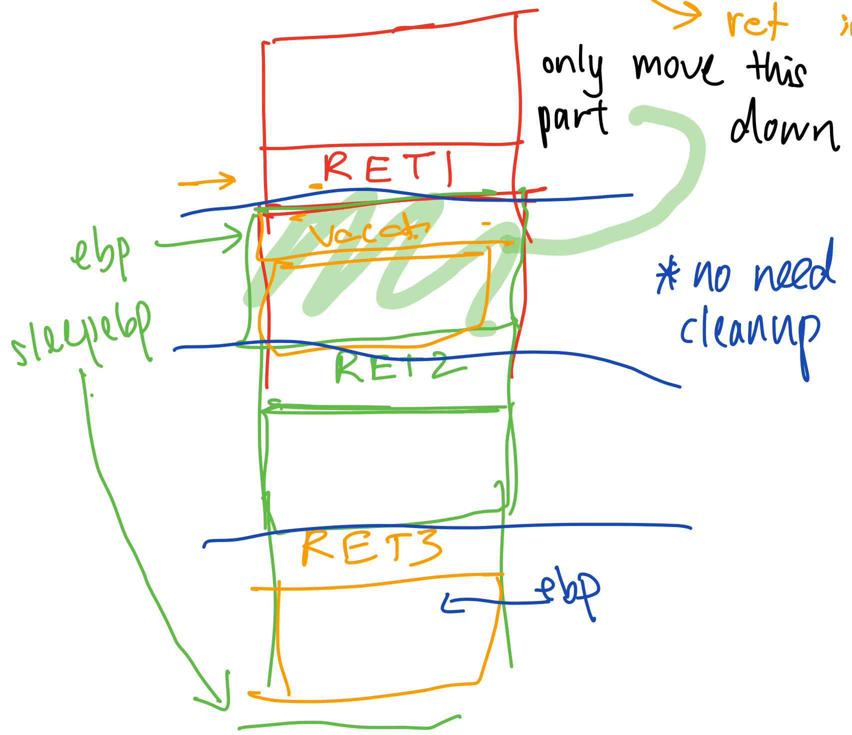


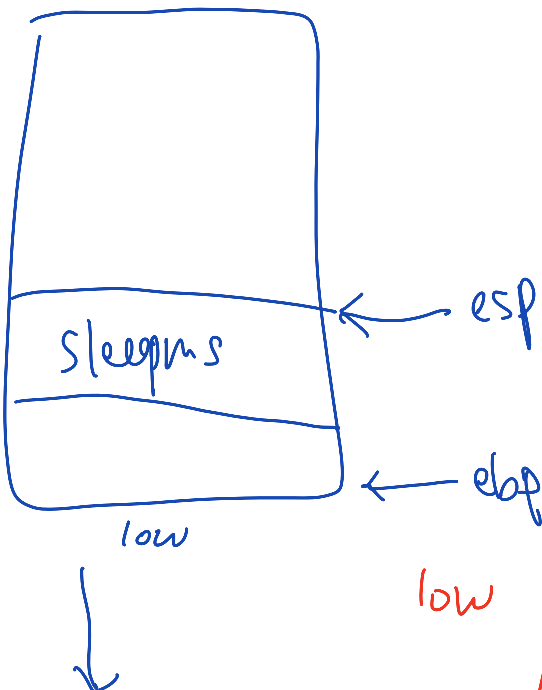


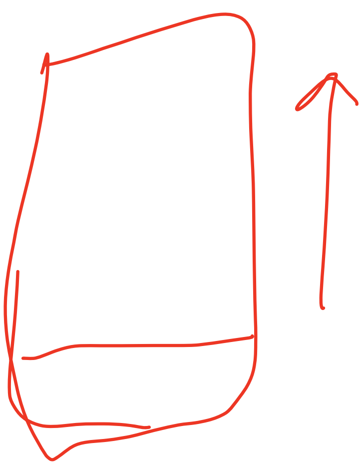

* making a new process table field to store function pointer
* check for the field in sleepms()
* update return address in sleepms() after resched() call
* look at stack trace, use inline assembly to get esp



p:r → dreamvac (vacation)
→ sleepms → resched → ctxsw
ret instruction
vacation → ret → RET1




holder

* DATA (0FDEFF7C) 00116B64 (1141604)
* DATA (0FDEFF80) 0FDEFF7C (266272636)
DATA (0FDEFF84) 0FDEFF7C (266272636)
DATA (0FDEFF88) 0FDEFF7C (266272636)
DATA (0FDEFF8C) 0FDEF464 (266269796)
DATA (0FDEFF90) 0011E290 (1172112)
DATA (0FDEFF94) 0FDEFFA4 (266272676)
DATA (0FDEFF98) 0FDEFF98 (266272664)
DATA (0FDEFF9C) 0000012E (302)
DATA (0FDEFFA0) 0000012F (303) <- Move everything here up
//vacation

FP (0FDEFFA4) 0FDEFFD4 (266272724) <- Keep this
RET 0x1049E6

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-fall2023/compile directory and run "make clean".

   ii) Go to the directory where lab4 (containing xinu-fall2023/ and lab4.pdf) is a subdirectory.

      For example, if /homes/alice/cs354/lab4/xinu-fall2023 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