Phase 3: Notes and Hints

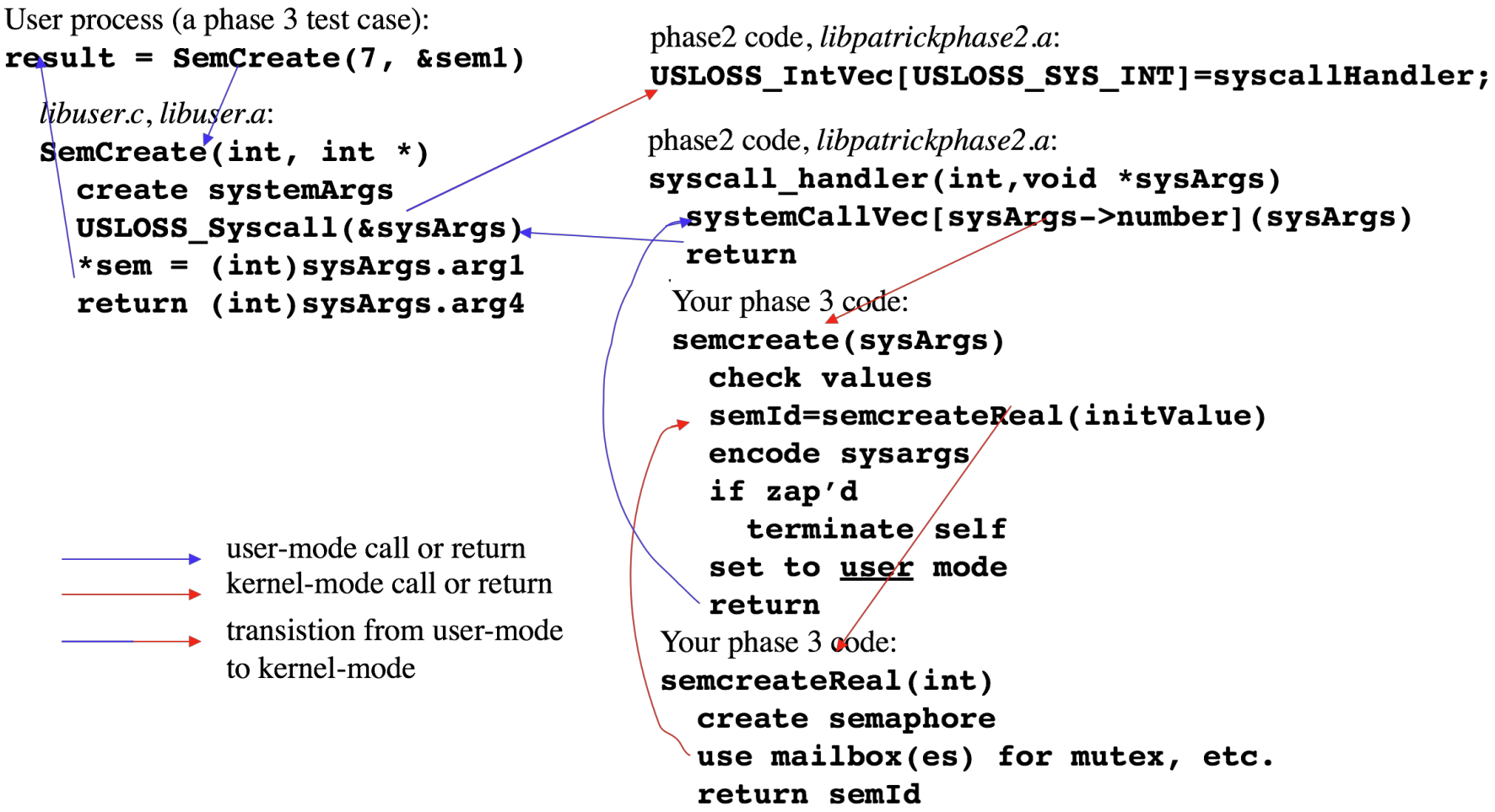
- Getting started
  - Provided starter files and test cases for Phase 3 are located in: **`/home/cs452/fall15/phase3`**.
  - USLOSS is a library: **`libusloss.a`**. It is located in **`/home/cs452/fall15/lib`**.
  - Your phase 3 code will be compiled into a library named: **`libphase3.a`** in your directory.
  - To execute a test case, you link the **`.o`** file of the test case with five libraries: phase 1, 2, and 3 libraries, plus the usloss and user libraries.
    - Typing '**`make`**' will create the **`libphase3.a`** library in your directory.
  - You can use your own **`libphase1.a`** and **libphase2.a** libraries:
    - In the provided *Makefile*, use the lines:

      ```
      PHASE1LIB = phase1
      PHASE2LIB = phase2
      ```

    - A copy of your **libphase1.a** and **libphase2.a** will need to be in your phase 3 directory for this to work.
    - Your phase 3 will be graded using Patrick's **`libpatrickphase1.a`** and **`libpatrickphase2.a`** libraries.
- Header files for Phase 3:
  - */home/cs452/fall15/include/phase[1-3].h*: function prototypes and constants to be used in this phase.
  - */home/cs452/fall15/include/libuser.h*: function prototypes for the system calls in this phase.
  - */home/cs452/fall15/include/usloss.h*: function prototypes for USLOSS functions + many useful constants.
  - Your phase 3 data structures and constants for phase 3 will go into local .h file(s) that you will provide.

- **`start2`** function:
  - The phase 2 library uses **`fork1`** to create a kernel-level process at priority 1 that will execute the **`start2`** code that you provide in phase 3. Thus, **`start2`** is the entry point for phase 3. When **`start2`** is called, there will be three processes already created: **`sentinel`**, **`start1`**, and **`start2`**.
  - Initialize your phase 3 data structures, in particular, the phase 3 process table and the semaphore table.
    - You will need a process table for phase 3. You cannot modify/extend the Phase 1 or 2 process tables! Use **`MAXPROC`** for the size of the phase 3 process table.
      - Note: When using Patrick's phase 1 library, you can use **`getpid() % MAXPROC`** to determine which slot in your phase 3 process table to use.
  - Initialize the **`systemCallVec`** array to point to the system call functions that you are creating in phase 3.
    - Make the unused parts of **`systemCallVec`** point to a **`nullsys3`** function. This function is similar to the **`nullsys`** function used in phase 2 with the difference that it will *terminate* the offending process, rather than halt the simulator.
  - Spawn the phase 3 test process: **`start3`**. Note that **`start3`** will be a user-mode process. It will run at priority 3 and will have a stack size of **`4 * USLOSS_MIN_STACK`**.

- Processes in phase 3 run in user mode, not kernel mode.
  - System calls that create and manage user mode processes:
    - **`Spawn`**, **`Wait`**, **`Terminate`**.
  - Errors cause termination of the offending process, rather than halting USLOSS.
  - **`Terminate`** will call **`quit`** after all the descendants of the terminating process have quit.
- System calls:
  - Protect the OS by not allowing user processes to execute system code.
  - Executing a system call in USLOSS causes a "software trap":
    - On USLOSS, this is an interrupt of type **`USLOSS_SYSCALL_INT`**.
    - **`USLOSS_SysCall`** changes process from user to kernel mode.
      - The only way to do this. Cannot use **`USLOSS_PsrSet`** to change from user to kernel mode.
    - No longer disable (or enable) interrupts. All the code in phase 3 runs with interrupts enabled.
      - Allows greater concurrency.
      - Time-slicing is now possible within the implementation code.
    - Must change back to user mode before exiting system code.
      - Use **`USLOSS_PsrSet`** to do this, since it is callable when in kernel mode.


User process (a phase 3 test case):
result = SemCreate(7, &sem1)
libuser.c, libuser.a:
SemCreate(int, int *)
create systemArgs
USLOSS_Syscall(&sysArgs)
*sem = (int)sysArgs.arg1
return (int)sysArgs.arg4
phase2 code, libpatrickphase2.a:
USLOSS_IntVec[USLOSS_SYS_INT]=syscallHandler;
phase2 code, libpatrickphase2.a:
syscall_handler(int,void *sysArgs)
systemCallVec[sysArgs->number](sysArgs)
return
Your phase 3 code:
semcreate(sysArgs)
check values
semId=semcreateReal(initValue)
encode sysargs
if zap'd
terminate self
set to user mode
return
Your phase 3 code:
semcreateReal(int)
create semaphore
use mailbox(es) for mutex, etc.
return semId
user-mode call or return
kernel-mode call or return
transistion from user-mode
to kernel-mode

- Since you cannot disable interrupts, how to do mutual exclusion?
  - Use mailboxes
  - See lecture notes for examples of how a mutex mailbox can be used.

- How to block processes when needed?
  - Can not use **`blockMe`** and **`unblockProc`** from phase 1.
  - Use a *private mailbox*:
    - Each process has one private mailbox.
    - This is a zero-slot mailbox.
    - Only the process ever does **`MboxReceive`** on its private mailbox.
    - Some other process will do the corresponding **`MboxSend`** when it is time to wake up the process.

- Strongly suggested convention:
  - Put the code that extracts values from the **`sysargs`** structure, and that checks those for correctness in one function. This function then calls the corresponding **`Real`** function which does the work of the system call.
  - I.e., for the semaphore create routine:

    `static void semCreate(systemArgs *argsPtr)`

    - Is the function pointed to by **`systemCallVec[SYS_SEMCREATE]`**.
    - Extracts the initial semaphore value from the **`systemArgs`** structure and makes sure it is not negative.
    - Calls **`semCreateReal`**.

    `int semCreateReal(int initValue)`

    - Which then handles the work of the semaphore creation.
    - **`semCreateReal`** returns the result of the creation to **`semCreate`**.

  - **`semCreate`** then puts this result back into the **`sysargs`** structure, changes to user mode, then returns.