Phase 2: Notes and Hints

- Getting started
  - Provided starter files and test cases for Phase 2 are located in: **`~cs452/fall15/phase2`**.
  - USLOSS is a library: **`libusloss.a`**. It is located in **`~cs452/fall15/lib`**.
  - Your phase2 code will be compiled into a library named: **`libphase2.a`** in your directory.
  - To execute a test case, you link the **`.o`** file of the test case with the **`libusloss.a`** and **`libphase1.a`** libraries.
    - Typing ‘**`make`**’ will create the **`libphase2.a`** library in your directory.
    - Typing ‘**`make test00`**’, for example, will create an executable test case named ‘**`test00`**’ in your directory.
  - You can use your own **`libphase1.a`** library:
    - In the provided *Makefile*, use the line:

      ```
      PHASE1LIB = phase1
      ```

    - a copy of your **`libphase1.a`** will need to be in your phase 2 directory for this to work
  - You can use Patrick’s **`libpatrickphase1.a`** library:
    - In the provided *Makefile*, use the line:

      ```
      PHASE1LIB = patrickphase1
      ```

      In the provided *Makefile*, use the line:

      ```
      PHASE1LIB = patrickphase1debug
      ```

      to link to a phase 1 library with debugging output from phase 1.
  - Your phase 2 will be graded using Patrick’s **`libpatrickphase1.a`** library.

- Header files for Phase 2:
  - *~cs452/fall15/include/phase2.h* and *~cs452/fall15/include/phase1.h*: contain function prototypes and constants to be used in this phase.
  - *~cs452/fall15/include/usloss.h*: contains function prototypes for USLOSS library functions, many useful constants.
  - *message.h*: <u>your</u> data structures and constants for phase2.

- Mode and interrupts
  - All functions in phase 2 have to be executed in <u>kernel</u> mode.
  - Test for kernel mode must be done for each phase 2 function, since processes must be in kernel mode to call these.
  - Enabling and disabling interrupts
    - Manipulate the appropriate bits in the PSR register.
    - Interrupts can be turned on and off only in kernel mode
  - When should interrupts be disabled? (A key point in writing <u>correct</u> phase 2 functions.)
  - Remember that calls to **`unblock_proc`** and **`block_me`** will result in interrupts being enabled!

- **`start1`** function:
  - The phase 1 library will use **`fork1`** to create a process at priority 1 that will execute the **`start1`** code that you provide in phase 2. Thus, **`start1`** is the entry point for phase 2. When the code in **`start1`** starts executing, there will be two processes already created: **`sentinel`** and **`start1`**.
  - Initialize your phase 2 data structures, in particular, the mailbox and mail slot arrays, and the phase 2 process table.
    - The **`struct mailbox`** and **`struct mail_slot`** structures in the provided *message.h* are not complete. Add/change fields as needed.
    - You will need a process table for Phase 2. You cannot modify/extend the phase 1 process table! Use **`MAXPROC`** for the size of the phase 2 process table.
      - Note: When using Patrick's *libpatrickphase1.a* code, you can use **`getpid() % MAXPROC`** to determine which slot in your phase 2 process table to use.
  - Create I/O mailboxes and initialize the **`int_vec`** and **`sys_vec`** arrays (see below).
  - Use **`fork1`** to create the test process: **`start2`**.
    - priority 1.
    - stack size = **`4 * USLOSS_MIN_STACK`**.
  - **`start1`** should then block on a **`join`** to wait for **`start2`** to quit.

Mailbox functions and notes:

- **MboxCreate**:
  - Allocate and initialize a location in your mailbox array. **MAXMBOX** from *phase2.h* is the size of this array.
  - Do not allocate slots for the mailbox at this time — they are allocated only as needed to hold messages.

- **MboxSend** basics:
  - Check for possible errors (message size too large, inactive mailbox id, etc.).
    - Return -1 if errors are found.
  - If a slot is available in this mailbox, allocate a slot from your mail slot table. **MAXSLOTS** determines the size of this array. **MAX_MESSAGE** determines the max number of bytes that can be held in the slot.
    - Note: if the mail slot table overflows, that is an error that should halt USLOSS.
    - Further note: for conditional send, the mail slot table overflow does not halt USLOSS.
      - Return -2 in this case.
  - Copy message into this slot. Use **memcpy**, not **strcpy**: messages can hold any type of data, not just strings.
  - Block the sender if this mailbox has no available slots. For example, mailbox has 5 slots, all of which already have a message.

- **`MboxReceive`** basics:
  - Check for possible errors (inactive mailbox id, etc.).
    - Return -1 in this case.
  - If one (or more) messages are available in the mailbox: **`memcpy`** the message from the slot to the receiver’s buffer.
  - Free the mailbox slot.
  - Block receiver if there are no messages in this mailbox.

- Basic tests:
  - *test00.c* — *test03.c*: Create **`start2`**, create mailbox(es), maximum number of mailboxes test.
  - *test04.c*: Creates two processes. Higher priority process sends to a mailbox. Lower priority process receives from mailbox.
  - *test05.c*: Creates two processes. Higher priority process receives (and will block since no message has yet been sent). Lower priority process then sends, unblocking higher priority process.
  - *test06.c*: Same as *test04.c*, but 5 messages are sent instead of 1. All should fit, since mailbox has 5 slots.

Interrupt handlers and i/o mailboxes:

- **`MboxCreate`**, **`MboxCondSend`**, and **`MboxReceive`** must be working before starting work on these!
- Will need to create i/o mailboxes in **`start1`** for the following devices:
  - Clock: 1 mailbox.
  - Terminals: 4 mailboxes, one for each terminal unit.
  - Disks: 2 mailboxes, one for each disk unit.
- New version of **`clock_handler`** needed for phase 2.
  - Error check: is the device actually the clock device?
  - Still must handle time slicing: call the phase 1 **`time_slice`** function when necessary.
  - Must conditionally send to the clock i/o mailbox every 5th clock interrupt.
- Disk and terminal handlers.
  - Error checks: Is the device the correct device? Is the unit number in the correct range?
  - Read the device’s status register by using the **`USLOSS_DeviceInput`** function.
  - Conditionally send the contents of the status register to the appropriate i/o mailbox.
    - Conditional send is used so the low-level device handler is never blocked on the mailbox.

- **`waitdevice`**
  - Process calls this when it wants to receive results of i/o operations (see *test13.c* and *test14.c*).
  - Use **`MboxReceive`** on the appropriate mailbox, as indicated by the parameters to **`waitdevice`**.

- **`check_io`**
  - Phase 1 needs to have the following test done in **`checkDeadlock`** (is already in *libpatrickphase1.a*):

    ```
    if ( check_io() == 1 )
       return;
    ```

  - This will cause the sentinel to call the function **`USLOSS_WaitInt`** when one, or more, processes are blocked on i/o mailboxes.
  - You will need to provide **`check_io`** as part of your phase 2 code.
    - Return 1: if at least one process is blocked on an i/o mailbox.
    - Return 0: otherwise.

- Other items: Ask us about these when you get to them (that is, after you have the other stuff above working)!
  - Releasing mailboxes that have blocked processes.
  - Priority inversion when several blocked senders (or receivers) are present at different priorities.
  - Zero-slot mailboxes.
  - **`systemCallVec[]`** and system calls.