Phase 1: Notes and Hints

- Getting started
  - The USLOSS source is available at **`/home/cs452/spring18/usloss/usloss3.6.tgz`**
    - Compiling USLOSS: See Section 2.1 of the USLOSS User's Manual
  - Some files to help you get started can be found in **`/home/cs452/spring18/phase1`**. Copy these to your own phase directory.
  - Your phase1 code will be compiled into a library named: **`libphase1.a`** in your directory.
  - To execute a test case, you link the **`.o`** file of the test case with the **`libusloss<version>.a`** and **`libphase1.a`** libraries.
    - Typing '**`make`**' will create the **`libphase1.a`** library in your directory.
    - Typing '**`make test00`**', for example, will create an executable test case named '**`test00`**' in your directory.

- Header files for Phase 1:
  - After you have compiled USLOSS:
    - **`phase1.h`**: contains function prototypes & constants to be used in this phase.
    - **`~/include/usloss.h`**: contains function prototypes for USLOSS library functions, many useful constants.
    - **`kernel.h`**: your data structures and constants for phase1.
- Mode and interrupts
  - All functions in this phase have to be executed in kernel mode.
  - Test for kernel mode must be done for each phase 1 function, since processes must be in kernel mode to call these.
    - See discussion of Process Status Register (Section 3.4) and functions to access this register (Section 7) in the USLOSS manual.
    - Some useful constants are defined in **`usloss.h`**.
  - 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 correct phase 1 functions.)

- **`startup()`** function
  - Called by USLOSS. Note: **`main()`** is inside the USLOSS library, *not* in your phase 1 code.
  - Initialize your kernel data structures, in particular, the process table.
    - The **`struct procStruct`** in the provided **`kernel.h`** is not complete. Add/change fields as needed!
  - Ready List(s): for processes waiting to run. Choose one of:
    - Single queue arranged by priorities; or
    - Multiple queues, one per priority.
- Sentinel process:
  - Call **`fork1()`** to create the sentinel process.
  - Sentinel runs when there are no more runnable processes.
    - It is the only process with priority **`6`**; its status is always READY.
  - The provided **`sentinel()`** function in **`skeleton.c`** is complete.
    - You will need to provide the **`checkDeadlock()`** function that **`sentinel()`** calls.
- Call **`fork1()`** to create a process that will execute **`start1()`**.
  - **`start1()`** will be inside every phase 1 test case.

`fork1()` function:

- Test for kernel mode.
- Initialize process table entry, which will include:
  - Check for valid stack size and priorities; there is a minimum stack size.
    - Use **`malloc()`** (or **`posix_memalign()`**) to create the stack. Save stack information in process table so stack can be freed in **`quit()`**.
      - This will be the only call to **`malloc`** (or **`posix_memalign`**) in phases 1, 2, 3, and 4!!
      - You will get to dynamically allocate memory again in phase 5.
- Store function pointer and argument value:
  - **`fork1()`** does not execute the function.
  - **`launch()`** does.
- Initialize context using the USLOSS **`USLOSS_ContextInit()`** function.
  - Use **`launch()`** as the function pointer passed to **`USLOSS_ContextInit()`**.
  - Necessary: must enable interrupts before starting function, and correctly handle return from function.
    - Unix analogy: What happens if **`main`** calls **`return`** instead of **`exit`**?
- Call the **`dispatcher()`** — let the **`dispatcher()`** decide whether the parent or the child runs next!
- Enable interrupts (for the parent) and return the pid of the child process.

`join()` function: There are three cases:

- The process has no children. (What happens?)
- Child(ren) **`quit()`** before the **`join()`** occurred.
  - Return the pid and quit status of <u>one</u> quit child and finish the clean up of that child's process table entry.
  - Report on quit children in the order the children **`quit()`**.
- No (unjoined) child has **`quit()`** … must wait.
  - How?
  - After wait is over: return the pid and quit status of the child that quit.
    - Where does the parent find these?

- The child status that **`join`** returns is the argument that the child passed to **`quit()`**.
- HINT: Pick <u>one</u> of the **`3`** cases above and work on that -- <u>ignore the other **`2`**.</u>
  - HINT: look at test00.c. Which case does it fall into? Implement that case first!

`quit()` function

- Error if a process with active children calls **`quit()`**. Halt USLOSS with appropriate error message.
- Cleanup the process table entry (but not entirely, see **`join()`**)
  - Two cases:
    - Parent has already done a **`join()`**, or
    - Parent has not (yet) done a **`join()`**.
- Unblock processes that **`zap`**'d this process (see below).
- May have children who have **`quit()`**, but for whom a **`join()`** was not (and now will not) be done.
  - This is <u>not</u> an error.

`dispatcher()` function: Decides which process gets to run.

- When is the dispatcher called?
- Checks if the current process can continue running: Has it been time-sliced? Has it been blocked? Is it still the highest priority among **`READY`** processes?
- Select a new process and perform a context switch in order to get it running.
- Choose according to scheduling policy: see Section 3.2 of Phase 1 handout.

Clock interrupt handler (see Section 3.2 of USLOSS manual).

- Defer working on this until your **`fork1`**, **`join`**, **`quit`**, and **`dispatcher`** functions are all working.
- Write an clockHandler function. Its prototype will be:

```
void clockHandler(int dev, void *arg)
```

  - In the code for **`clockHandler`**, you will ignore the arg parameter; it is not used by the clock.
- Interrupt vector is defined by USLOSS as an array of pointers to **`void`** functions with 2 integer arguments:

```
extern void (*USLOSS_IntVec[NUM_INTS]) (int dev, void *arg); /* from usloss.h */
```

- Initialize the appropriate slot to point to your clock handler function.

```
USLOSS_IntVec[USLOSS_CLOCK_DEV] = clockHandler;
```

- The other slots can be ignored (they will show up again in later phases).
- **`clockHandler()`** function.
  - Checks if the current process has exceeded its time slice. Calls **`dispatcher()`** if necessary.
    - Time slice is 80 ms (milliseconds).
  - The **`USLOSS_Clock()`** function returns time in microseconds (= 1,000 ms); thus, time slice is 80,000 μs.

`zap()` and **`isZapped()`** functions.

- The zap'ing process blocks until the zap'd process quits.
- Store information in the process table of both the zap'ing and zap'd processes.
- Zap'ing process blocks itself. (How?)
- The zap'd process, when it quits, needs to know about the zap'ing process.
- Note: there can be more than one zap'ing process! During a **`quit()`**, unblock all zap'ing processes.

```
PREFIX = ${HOME}

TARGET = libphase1.a
ASSIGNMENT = 452phase1
CC = gcc
AR = ar

COBJS = phase1.o
CSRCS = ${COBJS:.o=.c}

HDRS = kernel.h phase1.h

INCLUDE = ${PREFIX}/include

CFLAGS = -Wall -g -I${INCLUDE} -I. -std=gnu99

UNAME := $(shell uname -s)

ifeq ($(UNAME), Darwin)
       CFLAGS += -D_XOPEN_SOURCE      # use for Mac, NOT for Linux!!
endif

LDFLAGS = -L. -L${PREFIX}/lib

TESTDIR = testcases
TESTS = test00 test01 test02 test03 test04 test05 test06 test07 test08

LIBS = -lphase1 -lusloss3.6
```

```
$(TARGET):  $(COBJS)
         $(AR) -r $@ $(COBJS)

$(TESTS):   $(TARGET) p1.o
       $(CC) $(CFLAGS) -c $(TESTDIR)/$@.c
       $(CC) $(LDFLAGS) -o $@ $@.o $(LIBS) p1.o

clean:
       rm -f $(COBJS) $(TARGET) p1.o test??.o test?? test??.txt core term*.out

phase1.o:   kernel.h

submit:  $(CSRCS) $(HDRS) Makefile
       tar cvzf phase1.tgz $(CSRCS) $(HDRS) Makefile
```

Selected parts of **/home/cs452/spring18/phase1/skeleton.c**

```
void startup(int argc, char *argv[])
{
   int i;      /* loop index */
   int result; /* value returned by call to fork1() */

   /* initialize the process table */

   /* Initialize the Ready list, etc. */
   if (DEBUG && debugflag)
      USLOSS_Console("startup(): initializing the Ready & Blocked lists\n");
   ReadyList = NULL;

   /* Initialize the clock interrupt handler */

   /* startup a sentinel process */
   if (DEBUG && debugflag)
       USLOSS_Console("startup(): calling fork1() for sentinel\n");
   result = fork1("sentinel", sentinel, NULL, USLOSS_MIN_STACK, SENTINELPRIORITY);
   if (result < 0) {
      if (DEBUG && debugflag)
         USLOSS_Console("startup(): fork1 of sentinel returned error, halting...\n");
      USLOSS_Halt(1);
   }
```

```
   /* start the test process */
   if (DEBUG && debugflag)
      USLOSS_Console("startup(): calling fork1() for start1\n");
   result = fork1("start1", start1, NULL, 2 * USLOSS_MIN_STACK, 1);
   if (result < 0) {
      USLOSS_Console("startup(): fork1 for start1 returned an error, halting...\n");
      USLOSS_Halt(1);
   }

   USLOSS_Console("startup(): Should not see this message! ");
   USLOSS_Console("Returned from fork1 call that created start1\n");

   return;
} /* startup */


int sentinel (char *dummy)
{
   if (DEBUG && debugflag)
      USLOSS_Console("sentinel(): called\n");
   while (1)
   {
      checkDeadlock();
      USLOSS_WaitInt();
   }
} /* sentinel */
```

```
int fork1(char *name, int (*startFunc)(char *), char *arg, int stacksize, int priority)
{
   int procSlot;

   if (DEBUG && debugflag)
      USLOSS_Console("fork1(): creating process %s\n", name);

   // test if in kernel mode; halt if in user mode

   // Return if stack size is too small

   // Is there room in the process table? What is the next PID?

   // fill-in entry in process table
   if ( strlen(name) >= (MAXNAME - 1) ) {
      USLOSS_Console("fork1(): Process name is too long.  Halting...\n");
      USLOSS_Halt(1);
   }
   strcpy(ProcTable[procSlot].name, name);
   ProcTable[procSlot].startFunc = startFunc;
   if ( arg == NULL )
      ProcTable[procSlot].start_arg[0] = '\0';
   else if ( strlen(arg) >= (MAXARG - 1) ) {
      USLOSS_Console("fork1(): argument too long.  Halting...\n");
      USLOSS_Halt(1);
   }
   else
      strcpy(ProcTable[procSlot].startArg, arg);
```

```
    // Initialize context for this process, but use launch function pointer for
    // the initial value of the process's program counter (PC)

    USLOSS_ContextInit(&(ProcTable[procSlot].state),
                       ProcTable[procSlot].stack, ProcTable[procSlot].stackSize,
                       NULL, launch);

    /* for future phase(s) */
    p1_fork(ProcTable[procSlot].pid);

    // More stuff to do here...

} /* fork1 */
```

```
void launch()
{
   int result;

   if (DEBUG && debugflag)
      USLOSS_Console("launch(): started\n");

   // Enable interrupts

   // Call the function passed to fork1, and capture its return value
   result = Current->startFunc(Current->startArg);

   if (DEBUG && debugflag)
      USLOSS_Console("Process %d returned to launch\n", Current->pid);

   quit(result);
} /* launch */
```