USLOSS User's Manual

1 General Description

USLOSS (Unix Software Library for Operating System Simulation) is a set of functions that simulate the basic hardware features of a hypothetical computer system. The purpose of the simulator is to allow students to experiment with low-level systems programming concepts such as interrupt handling, device drivers, and process scheduling. The simulator is written in the C programming language to allow fast execution and easy debugging of student programs.

The simulator provides the appearance of a dedicated, **single** CPU system. A high-level context switch operation is provided to allow easy switching between processes. A simulated interrupt system is controlled through a user-defined interrupt vector table and through functions that enable and disable interrupts. The following interrupts/devices are supported:

- a periodic clock interrupt
- a system call function
- four user terminals
- two disk storage devices

The simulator also supports a user mode and a kernel mode.

2 Getting Started with USLOSS

Start by creating a set of working directories and building the USLOSS library. Find a location on your file systems and create a directory that acts as the root folder for your development work. This directory is referred to as *yourDirectory* in the following instructions.

With a terminal, change directory (cd) into *yourDirectory*. Copy the USLOSS source code package **usloss-3.0.1.tar.gz** to *yourDirectory*. Extract the source code using a command such as:

```
tar zxvf *.tar.gz
```

Note that if the downloaded file from D2L doesn't have the .gz extension, then your OS likely extracted the tar file automatically. In this case, you would need to adjust the **tar** command as follows:

```
tar xvf *.tar
```

After you extract the source code, cd to **usloss/src** and type **make install**. This will compile and install the library **libusloss.a** into **../build/lib** and install the USLOSS header files into **../build/include**. The include folder should have the file **usloss.h** and some additional header files that are used for the different phases of the project. You can ignore any warnings from the compilation and installation of the usloss library.

Create a directory for the phase 1 development project in *yourDirectory*. The directory should be named **phase1.** Change directory (cd) to **phase1** and create a symbolic link to USLOSS within this folder using the following command:

```
ln -s ../usloss/build usloss
```

This command creates a symbolic link named usloss in your phase1 directory. This link points to the build directory (the directory that contains the lib and include folders for usloss) and is needed by the provided Makefiles to build your solution.

The next step is to write a C program to run on the simulator. (You will write a C program in your phase 1 development first.) All source files that make use of the simulator's functions will need to include the file **usloss.h**, found in **usloss/include**, assuming you are under phase1 folder and a subfolder usloss created by the **ln -s** command. This header file contains definitions that are used to interface with the simulator. **USLOSS already has a main routine defined, so users cannot have a main routine in their source code**. Execution of user code will begin in a routine called **startup**. USLOSS also expects to find a routine called **finish**, but this is intended primarily for debugging and is provided in the skeleton code for project 1.

Once the C program has been written, you should create a **Makefile** to handle compilation. (Note that you will be provided with a sample Makefile for this purpose) When compiling a source file, use a command of the form:

```
            gcc $(CFLAGS) -c- I./usloss/include yourfile.c
```

The **-I** option will allow the compiler to find **usloss.h**. The compiled code for the simulator is contained in the library **./usloss/lib/libusloss.a**. To create the executable file, use a command of the form:

```
            gcc $(CFLAGS) -o yourcmd -L./usloss/lib yourfile.o -lusloss
```

After linking with **libusloss.a** you will have a version of the simulator called **yourcmd**, augmented by the routines in **yourfile.c**. The simulator runs just like any other compiled C program. If the user code is compiled with the **-g** option, then a standard C debugger such as **gdb** can be used for debugging.

3 Processor Features

USLOSS simulates a simple CPU, providing kernel/user modes, interrupts, and simple context switch support.

3.1 Modes of Operation

The simulator has two modes of operation: a user mode and a kernel mode. The kernel mode is a privileged mode in which all the USLOSS operations can be invoked. User mode more restricted. While running in user mode, any attempt to access hardware devices and or call kernel mode functions is blocked, and the attempt will cause an illegal instruction exception, which in turn will cause the simulator to abort the execution. For a complete list of which simulator functions are disallowed while in the user mode, see the Quick Reference at the end of this manual. USLOSS starts up in kernel mode; to switch to user mode the kernel must change the mode bit in the processor status register (see Section 3.4).

3.2 Interrupts

The interface to the USLOSS interrupt system consists of an interrupt vector table and two function calls. When an interrupt occurs, USLOSS switches to kernel mode and then disables interrupts and calls the appropriate routine as indicated by the interrupt vector. Six different types of interrupts/devices are simulated (symbolic constants are shown in parentheses): clock (CLOCK_DEV), count-down timer (ALARM_DEV), system call (SYSCALL), terminal (TERM_DEV), disk (DISK_DEV), and memory-management unit (MMU_DEV). For a detailed description of each type of interrupt, see the sections on devices[1].

To handle the various interrupts, the user must fill in the interrupt vector with the addresses of the interrupt handlers. This table is already declared as a global array in the simulator in **`usloss.h`** and can be referenced by name[2]. The symbolic constants for the devices, which are also declared in **`usloss.h,`** are designed to be used as indexes when initializing the table. For example, to install an interrupt handler for the clock interrupt, the following statement could be used:

```
int_vec[CLOCK DEV] = clockHandler;
```

Thereafter, whenever a clock interrupt occurs, execution is transferred to **`clockHandler`**. If an interrupt occurs and the interrupt vector for that type of interrupt has not been initialized, it will generally cause the simulator to suffer a segmentation fault. *Always initialize the interrupt vector* **before** *enabling interrupts.* When the **`startup`** routine is called, all interrupts are disabled. This provides an opportunity to initialize all the interrupt vector entries. The interrupts can then be enabled by setting the current interrupt enable bit in the processor status register (see Section 3.4).

Interrupt handlers are passed two parameters. The first parameter indicates the type of device sending the interrupt, and the second is referred to as a unit number. If multiple units of

[1] Note that this manual ignores the descriptions of count-down timer and memory-management unit since the two devices are not related to the four projects for this course.

[2] You need check the array declaration to define the interrupt handlers in your code.

the device type exist (such as for terminals), then the unit number would identify which of the terminals had generated the interrupt.

Generally, device handlers take some action and then return, thereby allowing the currently executing process to resume at the point where it was interrupted.

3.3 Syscall

The simulator treats system calls as a form of interrupt by routing them through the interrupt vector. The function pointed to by **`int_vec[SYSCALL]`** points to a routine that is invoked each time a **`usyscall`** operation is executed. The handler resembles an interrupt handler: the simulator switches to kernel mode, disables interrupts, and invokes the system call handler with two parameters. The first parameter contains the interrupt number, which will probably be of little interest. The second parameter is the argument passed by the caller, as in **`usyscall(arg)`**. The **`arg`** parameter is normally a pointer to a structure or array containing such information as a syscall number and any other arguments the syscall may require. Returning from a usyscall is like returning from any other interrupt handler; the calling process may be resumed via a normal function return, or a context switch may be performed.

3.4 Processor Status Register (PSR)

The state of the USLOSS processor is stored in the Processor Status Register (PSR), as shown in Figure 1. The bits in the PSR indicate the kernel mode and the state of the interrupts. The *current mode* bit is 1 if the processor is in kernel mode and is 0 otherwise. The *Current interrupt enable* bit is 1 if interrupts are enabled and is 0 otherwise. When an interrupt occurs, the processor saves the *current mode* and *Current interrupt enable* bits into the *Previous* bits. When the interrupt handler returns the *Current* bits are restored from the *Previous* bits. Thus, an interrupt handler can determine which mode the processor was in prior to the interrupt by looking at the previous mode bit.

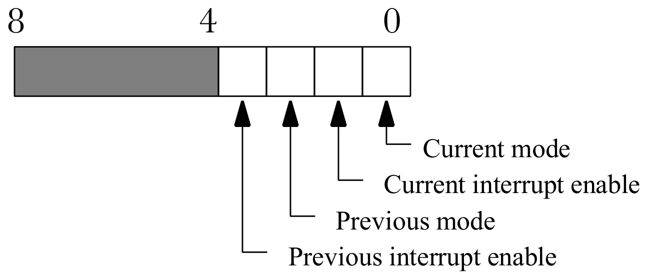

Figure 1: Processor Status Register

Changing either of the *Previous* bits in the interrupt handler changes the value of the *Current* bits when the interrupt handler returns. Changing the *Current* bits causes the mode and/or interrupt enable to change immediately. The PSR is accessed via **`psr_set`**() and **`psr_get`**(). Macros are defined in **`usloss.h`** for your use in accessing the PSR bits.

3.5 Process Support

USLOSS provides a context-switch mechanism for switching between processes. The context of each process is stored in a structure of type **`context`** (check **`usloss.h`** for the definition). The function:

```
context_switch(context *old_context, context *new_context)
```

performs a context switch, where **`old_context`** is a pointer to a context structure in which the state of the currently running process is to be stored, and **`new_context`** is a pointer to a context structure containing the state of the new process to be run. The **`context_switch`** routine saves the currently running process in the old context, including the PSR, and switchs to the process stored in the new context. If you don't want to save the current context (e.g. you're starting the first process), then pass NULL to **`context_switch`** as the value of **`old_context`**.

Prior to starting a new process, you must first allocate its context structure. You can allocate a context structure either statically (as part of a global variable or array) or dynamically (using malloc). It is likely that you will choose to have the context structure for a process simply be a field inside of its process control block. (The provided struct proc struct uses a context field for the purpose.) The context switch code does not care how the contexts are allocated; all it requires is that the pointers you give it point to valid storage.

After allocating the context structure you must then initialize it. First, you must allocate a stack for the process. The stack can be allocated dynamically (malloc). The amount of stack needed by a process depends upon the complexity of the process (i.e., the depth of the procedure call nesting and the size of local variable declarations). Stacks must be at least of size **`USLOSS_MIN_STACK`**, as defined in **`usloss.h.`** Once you have allocated a stack for the process, initialize its context by calling:

`context_init(context *context, unsigned int psr, void *stack, int stackSize, void(*pc))`

where **`context`** is a pointer to the context structure to be initialized, **`psr`** is the initial value of the PSR for the process (see the section on the PSR), **`stack`** is a pointer to the stack allocated for the process, and **`pc`** is the address of the initial function for the context to execute. Again, you can use **`malloc`** to dynamically allocate the stack memory for a new process. (Note that **`context_init`** will only initialize the context for the new process; to begin executing the process you must call **`context_switch.)`**

There is an important caveat concerning the creation of new contexts via **`context_init.`** When you call **context_switch** for the first time with a new context, the simulator will jump directly from inside **context_switch** to the starting routine of the new process. This means that the new process will not first return from **context_switch** and that any code that follows the call to **context_switch** will not be executed before the starting routine is invoked. In contrast, if the switch is to an already existing context that was previously saved by a call to **`context_switch`**, then that call will return when the process begins executing.

The internals of a context structure should not be directly modified by user source code; only **`context_init`** and **`context_switch`** are allowed to do so. Also, each process must have its own stack and its own context. You may share code between processes (since we don't allow self-modifying code), but not stacks or contexts.

4 Devices

USLOSS supports several device types. The following illustrates three of the device types: clock, terminal, and disk.

4.1 Clock Device

An interrupt by the clock device invokes the function pointed to by **`int vec[CLOCK DEV]`** at regular intervals. The interval length is determined by the resolution of the virtual timer provided by the host operating system. On our host, this interval is approximately 20 milliseconds, or one-fiftieth of a second, and is defined in CLOCK_MS. It should be noted that this clock interrupt is both far more infrequent and irregular than the one that would be found within a real computer; nevertheless, it is sufficient to implement multiprogramming and time slicing, as the code should be written in such a way as to be independent of the frequency of clock interrupts.

4.2 Terminal Devices

The simulator supports *four* terminal devices, each of which has a 16-bit status register and a 16-bit control register. The status and control registers are accessed via **`device_input`** and **`device_output`**, respectively. The four terminal devices share a single interrupt. An interrupt is generated each time there is a change in the contents of a status register, provided it is not masked by the interrupt mask in the corresponding control register, as described below. When an interrupt is generated the routine pointed to by **`int vec[TERM DEV]`** is called. The second parameter passed to the interrupt handler indicates which terminal's status changed. At this point, the terminal's status register should be read immediately using a call to **`device_input(TERM DEV, unit, &status),`** where **`unit`** is the unit number of the terminal you wish to access and **`status`** is the location in which to return the status. The contents of the status register are shown in Figure 2. Macros are provided in **`usloss.h`** to extract the fields of the status register.

The **xmit** status field indicates the status of the terminal's transmit capability, while the **recv** status field indicates its receive capability. The values for these status fields can be one of *DEV_READY*, *DEV_BUSY,or* *DEV_ERROR*, as defined in **usloss**.h. If the received status is *DEV_BUSY*, a character has been received on the terminal and stored in the **character** field of the status register. A status of *DEV_READY* means that no character has been received, and a status of *DEV_ERROR* indicates a problem. Failure to read the status register immediately upon receipt of a terminal interrupt may result in the loss of the character when another character is received. *You are guaranteed that the interval between character arrival is at least if the interval between clock ticks.*

Sending a character is somewhat more difficult than receiving one. To send a character you must first ensure that the terminal is ready to send a character, as indicated by the **xmit** status in its status register. A status of *DEV_READY* means it's ok to send a character, whereas as a status of *DEV_BUSY* means it is not. If you try to send a character while the terminal is busy, the character will be lost. Characters are sent by writing them to the terminal's control register via a call to **device_output(TERM DEV, unit, control)**, where **unit** is the unit number of the terminal to be written and **control** is the value to write to the control register. The format of a control register is shown in Figure 2. To send a character, put the character value into the upper 8 bits of the control register, and set the "send char" bit of the register. If the "send char" bit is not set, the character will not be sent. Characters can be sent at a maximum rate of one per clock tick.

The remaining two bits of the register consist of an interrupt mask for the terminal. If the *Xmit int enable* bit is set, the terminal will generate an interrupt when it transmits status changes, otherwise, it won't.

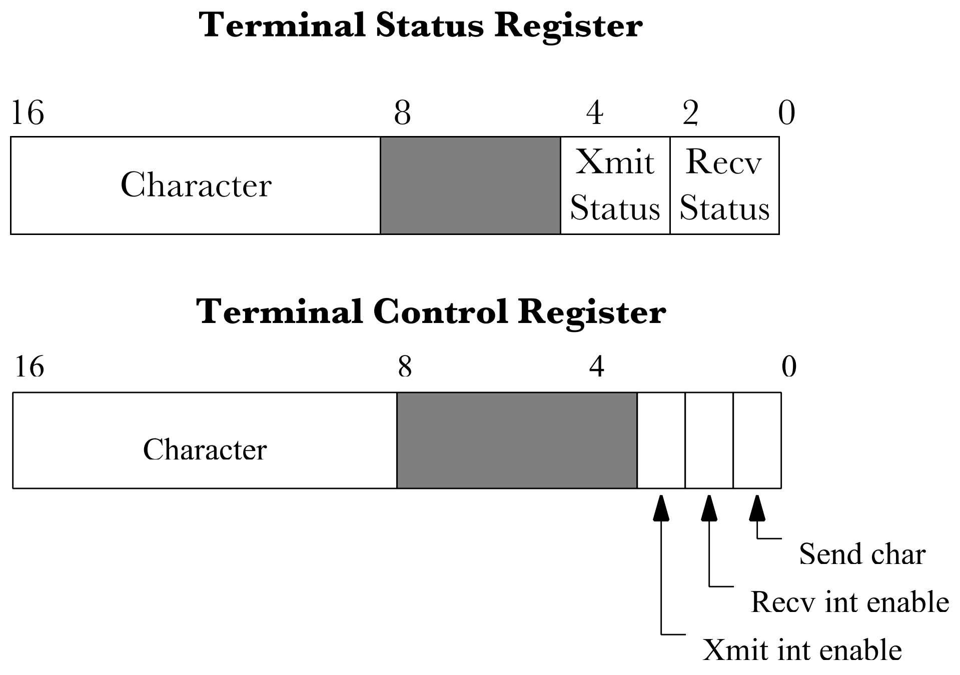

Figure 2: Terminal Registers

Similarly, if *Recv int enable* is set, the terminal will generate an interrupt when its receive status changes. If you don't want either of these interrupts, then don't set their bits in the control registers; set them if you do. You'll probably want to set the *Recv int enable* bit for all of the terminals and just leave them on, but for *Xmit int enable* you'll probably want to turn it on only when you have characters to transmit. Otherwise, you'll get a lot of spurious transmit interrupts that aren't useful.

There are macros in usloss.h to help you access the fields in the status and control registers. You should note that it is possible for a single interrupt to signify both the reception and transmission of a character on a terminal.

4.3 Disk Device

The disk device supports the following operations: seeking to a given track, and reading and writing a 512-byte sector within the current track. A disk operation is initiated by a call of the form: **`device_output(DISK DEV, unit, &request)`**, where **`unit`** is the unit number of the disk to be accessed, and **request** is a pointer to a **device_request** structure (defined in usloss.h).

This structure has three fields: **opr, reg1, and reg2**.

The **opr** field must be one of the predefined constants *DISK_READ, DISK_WRITE, DISK _SEEK, or DISK_TRACKS*. If **opr** is *DISK_READ or DISK _WRITE*, then **reg1** should contain the index of the sector to be read or written within the current track, and **reg2** should contain a pointer to a 512-byte data block into which data from the disk will be read or from which data will be written out to the disk. Note that each track on the disk has *16* sectors. If **opr** is *DISK_SEEK*, then **reg1** should contain the track number to which the disk's read/write head should be moved. If **opr** is *DISK_TRACKS* then **reg1** should contain a pointer to an integer into which the number of tracks in the disk will be stored.

After a request has been sent to the disk, further requests are ignored until the requested operation has been completed, at which point the function pointed to by **int vec[DISK DEV]** is called. The status of a disk device may be obtained by an **device_input(DISK DEV, unit, &status)** operation, which will set status to *DEV_READY* if the device is inactive, *DEV_BUSY* if a request is being processed, or *DEV_ERROR* if the last request could not be completed.

4.4 Device Files

The disk device stores the contents of each simulated disk in a file called **diskN**, where *N* is the unit number for the disk. USLOSS supports two disks, unit 0 and unit 1. A disk file is updated immediately upon every change to the disk, so no information will be lost when a simulation program terminates abnormally. The simulator requires that a disk file contain an even number of complete tracks; otherwise, an error occurs upon startup. A utility called **makedisk** is provided to create pseudo-disk files. Note that this utility is needed only when a new disk is to be created; as long as the disk file is not corrupted, the information is preserved between shutdowns and startups of the simulator.

The four terminal devices read their input from the files **term0.in, term1.in, term2.in, and term3.in**. These files must also reside in the directory in which USLOSS is being executed. If a terminal input file is not present, no input will be received from the corresponding terminal. The terminal input files should be created manually using a text editor. Terminal output is written to the files **term0.out, term1.out, term2.out, and term3.out**.

A utility called **pterm** is provided to allow users to operate real terminals with the simulator. To connect a physical terminal with the simulator, users must log in to that terminal and change to the directory in which the simulation is being run. Users should then enter a **pterm x** command, where **x** is the terminal number to be used. If an input file for that terminal exists, the user is given the choice of removing the file or aborting. The terminal is switched into cbreak input mode, and every character typed on the terminal is sent to the simulator and simultaneously written into the corresponding terminal input file (this will provide a record of what input was typed after the simulation terminates). Characters written to the terminal by the simulator are displayed on the screen. To exit, the user may strike either the interrupt or stop (ctrl-Z) keys, which will cause pterm to reset the terminal to normal mode and exit.

Normally, characters are read from terminal input files or physical terminals at the maximum rate possible, which is one character from each terminal for every *four* clock ticks. In some cases, it may be desirable to delay input from one or more terminals for a given interval. This may be accomplished by inserting '@'characters in the input files; each '@' character is read by the simulator but does not cause an interrupt and is thus invisible to a simulation program. Each '@' effectively delays the next input character from that terminal by four clock ticks.

5 Debugging Support

The primary method for debugging software in the simulator environment is through use of a standard C debugger such as **gdb**. Two printing functions are also provided. The **console** operation takes printf-style parameters and print to **stdout** and the **trace** operation prints to stderr. You should avoid using **`printf`** and **`fprintf`** as they may cause problems if interrupted.

A halt operation is provided by the simulator. When invoked, it will cause execution of the **`finish`** routine (which you define in your C file), and then terminates execution. Users might find it useful to have print statements or error checking code in **`finish`** to help in debugging.

USLOSS can also be debugged using a standard debugger such as **`gdb`**. However, be aware that the use of gdb is complicated by USLOSS interrupts which are implemented using the **`SIGUSR1`** signal. By default, `gdb` catches the SIGUSR1 signal and forces the debugged program to stop. To get around this problem, add the following in your *.gdbinit* file (either in the current directory or in your home directory):

```
handle SIGUSR1 nostop noprint
```

6 USLOSS Quick Reference

The following routines are provided by the simulator. Those whose names are in bold are accessible from kernel mode only.

`void console(char *fmt, ...)`

Printf-style write to the console device (stdout).

`void context_init(context *new, unsigned int psr, void * stack, int stackSize, void (*func)(void))`

Initializes a new context using `psr` as the context's initial PSR, the memory pointed to by `stack` as the stack memory, the `stackSize`, and the routine `func` as the starting address.

`void context_switch(context *old, context *new)`

Saves the current CPU state (including the PSR) in `old` and loads the state of `new` into the CPU.

`int device_input(int dev, int unit, int *status)`

Sets `*status` to the contents of the device status register indicated by `dev` and `unit`. If either `dev` or `unit` is invalid `DEV_INVALID` will be returned, otherwise, `DEV_OK` is returned.

`int device_output(int dev, int unit, void *arg)`

Sends `arg` to the device port indicated by `dev` and `unit`. Depending on the device, `arg` may be either an integer or a pointer to a structure of type device request containing the device request. If `dev` and `unit` are valid `DEV_OK` is returned, otherwise, `DEV_INVALID` is returned.

`void halt(int dumpcore)`

Causes execution of the finish routine and then terminates the simulator.

`unsigned int psr_get(void)`

Returns the current value of the PSR.

`void psr_set(unsigned int psr)`

Sets the PSR to the value in `psr`.

`void usyscall(void *arg)`

Causes an interrupt of type SYSCALL and passes `arg` as the second parameter to the interrupt handler.

`int sys_clock(void)`

Return the time (in microseconds) since the booting of the kernel.

`void trace(char *fmt,...)`

Printf-style write to the trace device (`stderr`).

`void waitint(void)`

Suspends execution until an interrupt is received.