

USLOSS User's Manual

Version 3.6

August 28, 2019

1. General Description

USLOSS (Unix Software Library for Operating System Simulation) is a library that simulates a simple computer system. USLOSS allows students to experiment with low-level systems programming concepts such as interrupt handling, device drivers, and process scheduling. USLOSS is written in the C programming language to allow fast execution and easy debugging of student programs.

USLOSS implements a single-core CPU with user and kernel modes, a high-level context-switch facility to allow easy switching between processes, interrupts, a memory-management unit (MMU) that allows for virtual addressing and virtual memory, and the following devices: a periodic clock interrupt, a count-down timer device, a system call (syscall) facility, four user terminals, and two disk storage devices.

2. Using USLOSS

2.1 Building from source

Untar the USLOSS tarball, e.g. `tar xvzf usloss.tgz`.

Change directory to `usloss` and type `./configure` then `make install`. By default this will install the binaries `pterm` and `makedisk` in `~/bin`, the library `libusloss<version>.a` in `~/lib`, the library `libdisk<version>.a` in `~/lib` and the header file `usloss.h` in `~/include`. The `<version>` is the library name is the USLOSS version number, e.g. `libusloss3.3.a`. You can specify a different installation directory via the `--prefix` option to `configure`; type `./configure --help` for details.

The next step is to write a C program that uses USLOSS (referred to as the operating system, or OS). All source files must include the file `usloss.h`. USLOSS defines `main` and will invoke the following functions when it runs:

```
test_setup
startup
finish
test_cleanup
```

The functions `test_setup` and `test_cleanup` are used for testing and set things up for the test then clean up afterwards (e.g. create input files in `test_setup` and delete them in `test_cleanup`). The function `startup` is where you implement the body of your OS. The function `finish` is called after `startup` returns (it is used primarily for debugging and is discussed later). Note that `test_setup` and `test_cleanup` are part of the test

infrastructure and therefore are **not** defined by the OS, unlike `startup` and `finish` which are part of the operating system and therefore **are** defined by the operating system.

Once the C program has been written, you should create a Makefile to handle compilation. When compiling a source file, use a command of the form:

```
gcc $(CFLAGS) -c -I<usloss>/build/include osfile.c
```

The **-I** option will allow the compiler to find `usloss.h` (<usloss> is the pathname of the USLOSS source directory). The above command will produce the object file `osfile.o`.

Assuming your test program is in `test0.c` use a command of the following form to create the test executable:

```
gcc $(CFLAGS) -o test0 test0.c -L<usloss>/build/lib  
osfile.o -lusloss3.00
```

This will create an executable called `test0`.

The resulting executable is run just like any other compiled C program. If it is compiled with the **-g** option, then a standard C debugger such as **gdb** can be used for debugging (see Section 6).

3. Processor Features

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

3.1 CPU Modes

The CPU has two modes of operation: *kernel* and *user*. Kernel mode is privileged, in which all USLOSS operations can be invoked. User mode cannot access hardware devices nor invoke certain USLOSS operations; doing so will cause an illegal instruction exception (interrupt). For a complete list of which operations are disallowed while in the user mode, see the command summary at the end of this manual. USLOSS starts up in kernel mode; to switch to user mode the OS 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 and two functions. When an interrupt occurs, USLOSS switches to kernel mode, disables interrupts, and calls the appropriate routine stored in the interrupt vector. Six different types of interrupts/devices are simulated (symbolic constants are shown in parentheses):

- clock (USLOSS_CLOCK_INT and USLOSS_CLOCK_DEV)
- count-down timer (USLOSS_ALARM_INT and USLOSS_ALARM_DEV)
- terminal (USLOSS_TERM_INT and USLOSS_TERM_DEV)
- system call (USLOSS_SYSCALL_INT)
- disk (USLOSS_DISK_INT and USLOSS_DISK_DEV)
- memory-management unit (USLOSS_MMU_INT)

- illegal instruction (USLOSS_ILLEGAL_INT)

See Section 4 for a detailed description the device interrupts, Section 3.3 for a description of system calls, and Section 5 for a description of the MMU.

To handle the various interrupts, the OS must fill in the interrupt vector with the addresses of the interrupt handlers. The vector declared as a global array in USLOSS (USLOSS_IntVec), and is simply referenced by name. The symbolic constants for the devices 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:

```
USLOSS_IntVec[USLOSS_CLOCK_INT] = clock_handler;
```

Thereafter, `clock_handler` will be invoked whenever a clock interrupt occurs. If an interrupt occurs and the interrupt vector for that type of interrupt has not been initialized, it will generally cause the simulator print an error message and dump core. *Always initialize the interrupt vector before enabling interrupts.* Interrupts are disabled when `startup` is invoked, providing an opportunity to initialize the interrupt vector. The interrupts can subsequently be enabled by setting the *current interrupt enable* bit in the processor status register (see Section 3.4).

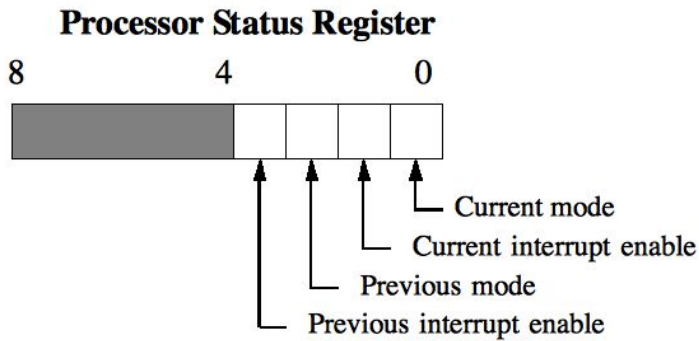
Interrupt handlers are passed two parameters. The first parameter indicates the type of interrupt, allowing the same handler to handle more than one type of interrupt, if desired. The second parameter varies depending on the type of interrupt. Generally, an interrupt handler takes some action and then returns, allowing execution to resume at the point where it was interrupted. In some cases, however, the interrupt handler may execute a context switch, in which case the current machine state is saved and execution is resumed at another point. An interrupt handler that invokes a context switch must be carefully written, as the state of the OS after the switch will almost certainly be different than the state before.

3.3 System Calls

USLOSS treats system calls as a form of interrupt, routing them through the interrupt vector. The function pointed to by `USLOSS_IntVec[USLOSS_SYSCALL_INT]` is invoked each time `USLOSS_Syscall` is called. The handler resembles an interrupt handler: USLOSS switches to kernel mode, disables interrupts, and invokes the system call handler with two parameters. The first parameter contains the interrupt number (USLOSS_SYSCALL_INT), which will probably be of little interest. The second parameter is the parameter passed to `USLOSS_Syscall`, which is typically a pointer to a structure or array containing such information as the syscall number and any other arguments the syscall requires.

3.4 Processor Status Register (PSR)

The state of the USLOSS processor is stored in the Processor Status Register (PSR). 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 0 otherwise. The *current interrupt enable bit* is 1 if interrupts are enabled, and 0 otherwise. When an interrupt occurs the processor saves the current mode and 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*. 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 `USLOSS_PsrSet` and `USLOSS_PsrGet`. Macros are defined in *usloss.h* for the OS to 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 `USLOSS_Context` (include *usloss.h* for the definition). The internals of a `USLOSS_Context` structure should not be directly modified by the OS; only `USLOSS_ContextInit` and `USLOSS_ContextSwitch` may do so. Also, each process must have its own stack and own context. The OS can share code between processes, but not stacks or contexts.

Prior to creating a new process the OS must first allocate an unused `USLOSS_Context`. The OS can allocate the structure statically (as part of a global variable or array) or dynamically (using `malloc`). Typically, the OS will embed the `USLOSS_Context` as a field in a process control block (PCB).

3.5.1 Initializing a Context

```
USLOSS_ContextInit(USLOSS_Context *context, void *stack, int
stackSize, USLOSS_PTE *pageTable, void(*func)(void))
```

The `context` parameter is the context structure to be initialized. The `psr` is the initial PSR for the process (see Section 3.4). The `func` is the address of the initial function the process is to execute. The `stack` and `stackSize` define the stack for the process; the stack can be allocated either statically (global variable) or dynamically (`malloc`). The stack size depends upon the complexity of the process (i.e., the depth of the procedure call nesting and size of

local variable declarations), but the stack must be at least of size `USLOSS_MIN_STACK`, as defined in *usloss.h*. The `pageTable` is the address of the page table for the process, if any (see Section 5.1). Note that `USLOSS_ContextInit` will only initialize the context for the new process; to actually begin executing the process the OS must call `USLOSS_ContextSwitch`.

The function `func` must not return, otherwise the stack will underflow. USLOSS will catch this and dump core, but the OS should use a “wrapper” function to detect that `func` has returned and terminate its context. The wrapper calls `func` and terminates the context if `func` returns.

There is an important caveat concerning the creation of new contexts via `USLOSS_ContextInit`. When the OS calls `USLOSS_ContextSwitch` for the first time with a new context, USLOSS will jump directly from inside `USLOSS_ContextSwitch` to the starting function of the new process. Any code that follows the call to `USLOSS_ContextSwitch` will not be executed in the context of the new process.

3.5.2 Switching Contexts

```
USLOSS_ContextSwitch(USLOSS_Context*old_context,USLOSS_Context
*new_context)
```

This function 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 `USLOSS_ContextSwitch` routine will save the currently running process in the old context, including the PSR, and switch to the process stored in the new context. If the OS doesn't want to save the current context (e.g. it's starting the first process), then it should pass `NULL` as the value of `old_context`.

4. Devices

USLOSS supports several device types: clock, alarm, terminal, and disk. Devices are read using `USLOSS_DeviceInput` and written using `USLOSS_DeviceOutput`.

4.1 Clock Device

The clock device invokes the function in `USLOSS_IntVec[USLOSS_CLOCK_INT]` at regular intervals, the length of which is determined by the resolution of the virtual timer provided by platform on which USLOSS runs. On *lectura*, this interval is approximately 20 milliseconds, or one fiftieth of a second, and is defined in `USLOSS_CLOCK_MS`. It should be noted that this clock interrupt is both far more infrequent and irregular than the one that would be provided by a real CPU; 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.

The clock device has a 32-bit status register that is accessed via `USLOSS_DeviceInput`. This register contains the time (in microseconds) since USLOSS started running.

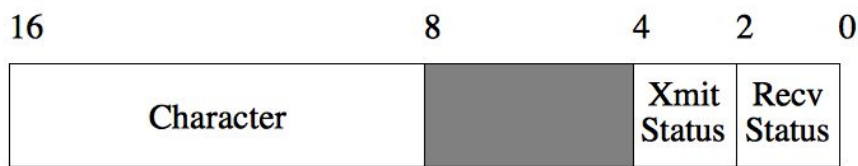
4.2 Alarm Device

The alarm device is a count-down timer intended primarily for debugging purposes. It may be set by the OS to send an interrupt a given number of clock ticks in the future. The alarm is set by calling `USLOSS_DeviceOutput(USLOSS_ALARM_DEV, 0, n)`, where `n` is a number between 1 and 255. The alarm interrupt arrives in between interrupts from the regular clock, so synchronization between the two is unnecessary. Several outstanding alarm interrupts may be scheduled at a time. When an alarm interrupt occurs, the function pointed to by `USLOSS_IntVec[USLOSS_ALARM_INT]` is called.

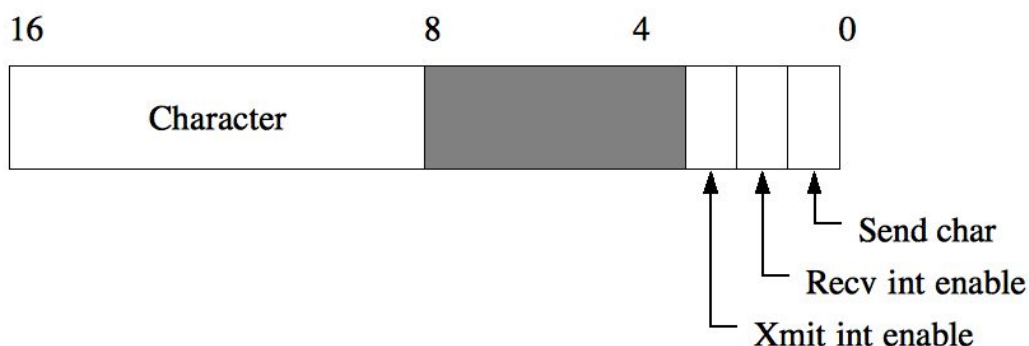
4.3 Terminal Devices

USLOSS supports four terminal devices that share a single interrupt (`USLOSS_TERM_INT`). Each terminal has a 16-bit status register and a 16-bit control register accessed via `USLOSS_DeviceInput` and `USLOSS_DeviceOutput`, respectively. 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 `USLOSS_IntVec[USLOSS_TERM_INT]` is called. The second parameter passed to the interrupt handler is the unit number of the terminal whose status changed (the terminals are numbered 0-3). The terminal's status register should be read immediately using `USLOSS_DeviceInput(USLOSS_TERM_DEV, unit, &status)`, where `unit` indicates which terminal's status register to read, and `status` is the location in which to return the status. The contents of the status register are shown in the following illustration. 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 `USLOSS_DEV_READY`, `USLOSS_DEV_BUSY`, or `USLOSS_DEV_ERROR`, as defined in *usloss.h*. If the receive status is `USLOSS_DEV_BUSY`, then a character has been received on the terminal and is stored in the *character* field of the status register. A status of `USLOSS_DEV_READY` means that no character has been received, while a status of `USLOSS_DEV_ERROR` indicates a problem. Each terminal has space to store only a single character, so a 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. The interval between character arrival is at least as long as the interval between clock ticks, so there should be plenty of time to respond to a terminal interrupt.

Terminal Status Register



Terminal Control Register



Sending a character is somewhat more difficult than receiving one. To send a character the OS 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 `USLOSS_DEV_READY` means it's ok to send a character, whereas as a status of `USLOSS_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 `USLOSS_DeviceOutput(USLOSS_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 the following diagram. To send a character, put the character value into the *character* field 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 its transmit status changes, otherwise it won't. Similarly, if *recv int enable* is set, the terminal will generate an interrupt when its receive status changes. If the OS doesn't want either of these interrupts, then it shouldn't set the bits in the control registers. In general, the OS should set the *recv int enable* bit for all of the terminals and just leave them on as a receive interrupt will only be generated when a character is received. The OS should only set the *xmit int enable* bit when it has characters to transmit on that terminal, otherwise it will get a lot of spurious transmit interrupts that aren't useful.

There are macros in `usloss.h` to help the OS 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.

4.3.1 Terminal Files

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 `p_term` is provided to allow users to use real terminals with USLOSS. To connect a real 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 `p_term x` command, where `x` is the terminal number to connect to. 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 `p_term` 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 the OS. Each '@' effectively delays the next input character from that terminal by four clock ticks.

4.4 Disk Devices

The disk devices support 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: `USLOSS_DeviceOutput(USLOSS_DISK_DEV, unit, request)`, where `unit` is the unit number of the disk to be accessed, and `request` is a pointer to a `USLOSS_DeviceRequest` structure (defined in `usloss.h`). This structure has three fields: *opr*, *reg1*, and *reg2*.

The *opr* field must be one of the predefined constants `USLOSS_DISK_READ`, `USLOSS_DISK_WRITE`, `USLOSS_DISK_SEEK`, or `USLOSS_DISK_TRACKS`. If *opr* is `USLOSS_DISK_READ` or `USLOSS_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 buffer into which data from the disk will be read or from which data will be written. Note that each track on the disk has 16 sectors. If *opr* is `USLOSS_DISK_SEEK`, then *reg1* must contain the track number to which the disk's read/write head should be moved. If *opr* is `USLOSS_DISK_TRACKS` then *reg1* must contain a pointer to an integer into which the number of disk tracks 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 `USLOSS_IntVec[USLOSS_DISK_INT]` is called. The status of a disk device may be obtained by an `USLOSS_DeviceInput(USLOSS_DISK_DEV, unit, &status)` operation, which will set `status` to `USLOSS_DEV_READY` if the request completed, `USLOSS_DEV_BUSY` if a request is being processed, or `USLOSS_DEV_ERROR` if the last request caused an error.

4.4.1 Disk 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 should USLOSS terminate abnormally. USLOSS 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 disk files. Note that this utility is needed only to create a new disk; as long as the disk file is not corrupted, its contents are preserved when USLOSS reboots. A library named `libdisk<version>.a` is also provided to allow creation of disks in test cases by calling `Disk_Create` from `test_setup`.

5. MMU

The USLOSS MMU maps virtual pages to physical page frames, allowing different processes to have different virtual address spaces. Due to the limitations of running USLOSS in a UNIX process, it isn't possible for USLOSS processes to have totally separate address spaces; instead the MMU only operates on a single region of the larger shared address space (called the *VM region*). Accesses outside of the VM region are unaffected by the MMU; accesses inside the VM region are mapped by the MMU to allow processes to see different memory contents.

The USLOSS MMU maps virtual pages to physical page frames, allowing different processes to have different virtual address spaces. Due to the limitations of running USLOSS in a UNIX process, it isn't possible for USLOSS processes to have totally separate address spaces; instead the MMU only operates on a single region of the larger shared address space (called the *VM region*). Accesses outside of the VM region are unaffected by the MMU; accesses inside the VM region are mapped by the MMU to allow processes to see different memory contents.

The MMU supports two modes of operation: page table mode, and TLB mode.

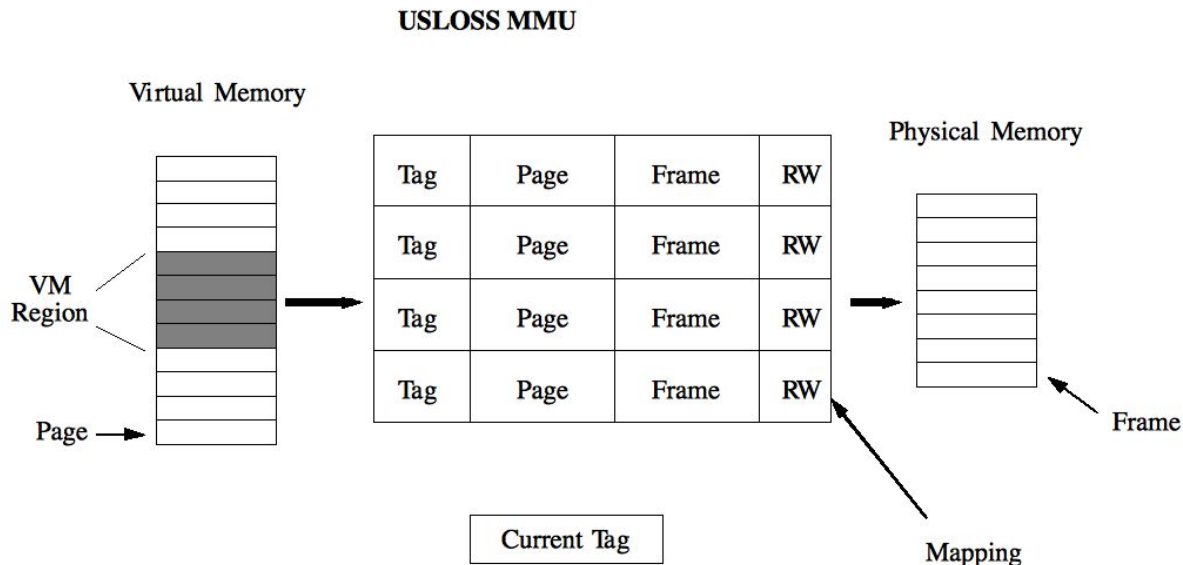
5.1 Page Table Mode

In page table mode the USLOSS MMU implements a single-level page table. A page table context is an array of `USLOSS_PTE` structures, one for each page in the VM region. Each `USLOSS_PTE` contains four fields: `incore`, `read`, `write`, and `frame`. The `incore` field is 1 if the page is in memory (core) and is 0 otherwise. If `incore` is 1 then `frame` indicates which page frame contains the page, otherwise `frame` is not used. If `read` is 1 then the page is readable, otherwise it is not. Likewise, if `write` is 1 then the page is writable, otherwise it is not. Note: the USLOSS MMU does not support write-only pages.

The page table for a context is specified when `USLOSS_ContextInit` is called. The USLOSS MMU will automatically switch page tables in `USLOSS_ContextSwitch`. The current page table can be obtained via `USLOSS_MmuGetPageTable`, and set via `USLOSS_MmuSetPageTable`. Note: if the OS modifies the current page table it must call `USLOSS_MmuSetPageTable` to notify the USLOSS MMU that the page table has changed; the USLOSS MMU cannot automatically detect changes to the current page table.

5.2 TLB Mode

The following diagram illustrates the functioning of the MMU in TLB mode:



The MMU contains a set of *mappings*, each of which contains a *tag*, page number, frame number, and two protection bits (read and write). The MMU uses these mappings to convert addresses in the VM region into physical memory addresses. The MMU performs the following four steps when a process accesses the VM region. First, the MMU determines which page within the VM region is being accessed. Second, the MMU finds all mappings whose tag matches the *current tag register*. Third, the MMU searches the resulting mappings to find one whose page number matches the page that is being accessed. If a mapping is not found after steps 2 and 3 the MMU will generate an MMU interrupt. This causes the interrupt handler associated with the `USLOSS_MMU_INT` interrupt to be invoked, allowing the OS to handle the problem. If a mapping is found, the MMU compares the type of access with the protection bits in the mapping. If the access is allowed the frame number from the mapping and the page offset from the virtual address are combined to form a physical address. If the access violates the page's protection a `USLOSS_MMU_INT` interrupt occurs.

The size of the VM region, the physical memory, and the number of mappings in the MMU are all specified when the MMU is configured. Varying the relative sizes of the VM region and the physical memory allow the effectiveness of the OS page replacement algorithm to be measured under different workloads and system configurations. Varying the number of mappings in the MME allows the MMU to function as a single-level page table, or as a translation buffer (TB). The former is achieved by configuring the MMU so that the number of mappings is equal to the number of pages in the VM region. This allows the MMU to hold mappings for all of a process's pages that are currently in memory, so that the lack of a mapping for an accessed page indicates a page fault. If, on the other hand, the MMU is

configured so that are fewer mappings than pages in the VM region, the lack of a mapping for a page does not necessarily imply a page fault. The desired page may be in memory, but not mapped by the MMU because there aren't enough mappings to go around. In this case the OS removes an existing mapping and adds one for the page that is being accessed.

The tag functionality of the MMU makes it possible for the MMU to store mappings for several processes. The MMU supports `USLOSS_MMU_NUM_TAGS` different tags, allowing up to that many different processes to simultaneously have mappings in the MMU. The MMU will not store more than one mapping with a given tag and page, so that the maximum number of mappings that can have the same tag is equal to the number of pages in the VM region. Thus the maximum number of mappings that the MMU can hold is `USLOSS_MMU_NUM_TAGS` multiplied by the number of pages in the VM region. It can, however, be configured to hold fewer as described in the previous paragraph. The advantage of using tags is that a context switch between two processes whose mappings are already loaded in the MMU can be achieved simply by changing the value in the current tag register.

Finally, the MMU supports access bits for each physical page frame. A *reference bit* is set when the frame is either read or written, and a *dirty bit* is set when the frame is written. These bits can be also be cleared and set in software, allowing the OS to implement a variety of page replacement algorithms.

5.1 MMU Error Codes

Many of MMU interface routines return the following error codes, defined in *mmu.h*:

Error Code	Meaning
<code>USLOSS_MMU_OK</code>	No error
<code>USLOSS_MMU_ERR_OFF</code>	MMU has not been initialized
<code>USLOSS_MMU_ERR_ON</code>	MMU has already been initialized
<code>USLOSS_MMU_ERR_PAGE</code>	Invalid page number
<code>USLOSS_MMU_ERR_FRAME</code>	Invalid frame number
<code>USLOSS_MMU_ERR_PROT</code>	Invalid protection
<code>USLOSS_MMU_ERR_TAG</code>	Invalid tag
<code>USLOSS_MMU_ERR_REMAP</code>	Mapping with same tag and page already exists
<code>USLOSS_MMU_ERR_NOMAP</code>	Mapping not found
<code>USLOSS_MMU_ERR_ACC</code>	Invalid access bits
<code>USLOSS_MMU_ERR_MAPS</code>	Too many mappings
<code>USLOSS_MMU_ERR_MODE</code>	Operation is invalid in the current MMU mode

5.3 MMU Functions

The following are the USLOSS MMU functions. Functions that are only valid in page table mode are denoted with (p) while functions that are only valid in TLB mode are denoted with (t), otherwise they are valid in both modes (b).

```
int USLOSS_MmuInit(int numMaps, int numPages, int numFrames,
int mode) (b)
```

Creates a physical memory containing `numFrames` frames, a VM region containing `numPages` pages. The mode must be either `USLOSS_MMU_MODE_TLB` or

USLOSS_MMU_MODE_PAGETABLE. In TLB mode the MMU will contain numMaps mappings. Initially the MMU has no valid mappings, and the current tag register is 0. In page table mode numMaps is not used. Returns a standard MMU error code.

int USLOSS_MmuDone(void) (b)

Releases all the resources associated with the MMU. Subsequent accesses to the VM region will result in a segmentation violation. Returns a standard MMU error code.

int USLOSS_MmuPageSize(void) (b)

Returns the size of a page, in bytes.

void *USLOSS_MmuRegion(int *numPagesPtr) (b)

Returns the address of the VM region, and stores the number of pages it contains in *numPagesPtr. If the MMU has not been initialized this routine will return NULL.

int USLOSS_MmuMap(int tag, int page, int frame, int protection) (t)

Stores a mapping in the MMU. USLOSS_MMU_ERR_REMAP is returned if a mapping with the same tag and page already exists. Valid protection values are USLOSS_MMU_PROT_NONE (no access), USLOSS_MMU_PROT_READ, (page is read-only), and USLOSS_MMU_PROT_RW (page can be both read and written). Returns a standard MMU error code.

int USLOSS_MmuUnmap(int tag, int page) (t)

Removes the mapping with the matching tag and page. Subsequent accesses to the page when the current tag is set to tag will result in an MMU interrupt. Returns a standard MMU error code.

int USLOSS_MmuGetMap(int tag, int page, int *framePtr, int *protPtr) (t)

Provides mapping information. If no mapping matches the given tag and page USLOSS_MMU_ERR_NOMAP is returned, otherwise the mapping's frame is stored in *framePtr, and the protection in *protPtr. Returns a standard MMU error code.

int USLOSS_MmuGetCause(void) (b)

Returns the cause of the most recent MMU interrupt. USLOSS_MMU_FAULT means that the page is not mapped to a frame. USLOSS_MMU_ACCESS means that a mapping was found, but that the protections on the page prohibit the attempted access. USLOSS_MMU_NONE means that an MMU interrupt has not yet occurred.

int USLOSS_MmuGetAccess(int frame, int *accessPtr) (b)

Stores the access bits for a frame in *accessPtr. If USLOSS_MMU_REF is set the page has been referenced. If USLOSS_MMU_DIRTY is set the page has been written. Returns a standard MMU error code.

```
int USLOSS_MmuSetAccess(int frame, int access)(b)
```

Sets the frame's access bits to `access`. Returns a standard MMU error code.

```
int USLOSS_MmuSetTag(int tag) (t)
```

Sets the current tag register to `tag`. Returns a standard MMU error code.

```
int USLOSS_MmuGetTag(int *tagPtr) (t)
```

Stores the current tag register in `*tagPtr`. Returns a standard MMU error code.

```
int USLOSS_MmuSetPageTable(USLOSS_PTE *pageTable) (p)
```

Sets the current page table to `pageTable`. This routine must also be called when the current page table is updated. Returns a standard MMU error code.

```
int USLOSS_MmuGetPageTable(USLOSS_PTE **pageTable) (p)
```

Stores the current page table in `pageTable`. Returns a standard MMU error code.

5.4 MMU Interrupts

The function pointed to by `USLOSS_IntVec[USLOSS_MMU_INT]` will be invoked when an MMU interrupt occurs. The MMU interrupt handler has the following definition:

```
void Handler(int type, void *offset)
```

`type` is the type of interrupt (`USLOSS_MMU_INT`). `offset` is an integer (cast as a void *) containing the byte offset from the start of the VM region to the address that caused the interrupt.

USLOSS will take care of restarting the offending instruction when the MMU interrupt handler returns. This means that the address that caused the problem will be given to the MMU again for mapping. If there is still a problem another MMU interrupt will be generated. The assumption is that the OS will either enter configure the MMU so that the address will no longer cause an interrupt, or kill the process.

6. Debugging Support

USLOSS provides two printing functions to help with debugging. The `USLOSS_Console` operation takes `printf`-style parameters and prints to `stdout`, and the `USLOSS_Trace` operation prints to `stderr`. You should avoid using `printf` and `fprintf` as they may cause problems if interrupted.

USLOSS also provides a halt operation, `USLOSS_Halt`. When invoked, it will cause execution of the `finish` routine (defined by the OS), and then terminates execution. A core file is produced if the parameter to `USLOSS_Halt` is non-zero. The OS developer might find it useful to have print statements or error checking code in `finish` to help in debugging. Lastly, the alarm device may also be used for debugging purposes. It can be set to interrupt at a time that is of interest so that the program status can be examined.

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
```

Similarly, the OS uses the MMU you should add the following to your `.gdbinit`:

```
handle SIGSEGV nostop noprint
handle SIGBUS nostop noprint
```

The MMU uses `SIGSEGV` and `SIGBUS` to implement memory mapping. Note that telling `gdb` to ignore `SIGSEGV` and `SIGBUS` will cause it to ignore segmentation violations and bus errors caused by real bugs the OS, so don't ignore these signals unless the OS uses the MMU.

7. USLOSS Quick Reference

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

`void USLOSS_Console(char *fmt, ...)` - Printf-style write to stdout.

`void USLOSS_ContextInit(USLOSS_Context*new, void *stack, int stackSize, USLOSS_PTE *pageTable, void (*func)(void))` - Initializes the context `new` using the memory pointed to by `stack` of size `stackSize` (in bytes) as the stack, and the routine `func` as the starting address. The `pageTable` points to the page table for the context, which may be NULL.

`void USLOSS_ContextSwitch(context *old, context *new)` - Saves the current CPU state (including the PSR) in `old`, and loads state `new` into the CPU. The `old` parameter may be NULL.

`int USLOSS_DeviceInput(int dev, int unit, int *status)` - Sets `*status` to the contents of the device status register indicated by `dev` and `unit`. If `dev` and `unit` are valid `USLOSS_DEV_OK` is returned, otherwise `USLOSS_DEV_INVALID` is returned.

`int USLOSS_DeviceOutput(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 `USLOSS_DeviceRequest` containing the device request. If `dev` and `unit` are valid `USLOSS_DEV_OK` is returned, otherwise `USLOSS_DEV_INVALID` is returned.

`void USLOSS_Halt(int dumpcore)` - Causes execution of the finish routine and then terminates USLOSS. A core file is produced if `dumpcore` is non-zero.

`unsigned int USLOSS_PSRGet(void)` - Returns the current value of the PSR.

`int USLOSS_PSRSet(unsigned int psr)` - Sets the PSR to the value in `psr`. Returns `USLOSS_ERR_INVALID_PSR` if `psr` is not a valid value for the PSR, `USLOSS_DEV_OK` otherwise.

`void USLOSS_Syscall(void *arg)` - Causes an interrupt of type `USLOSS_SYSCALL_INT` and passes `arg` as the second parameter to the interrupt handler.

`void USLOSS_IllegalInstruction(void)` - Causes an interrupt of type `USLOSS_ILLEGAL_INT`.

`void USLOSS_Trace(char *fmt, ...)` - Printf-style write to stderr.

`void USLOSS_WaitInt(void)` - Suspends execution until an interrupt occurs.

`int USLOSS_MmuInit(int numMaps, int numPages, int numFrames, int mode)` – Initializes the MMU.

`int USLOSS_MmuDone(void)` - Releases the MMU.

`int USLOSS_MmuPageSize(void)` - Returns the size of a page.

`void *USLOSS_MmuRegion(int *numPagesPtr)` - Returns the address and size of the VM region.

`int USLOSS_MmuMap(int tag, int page, int frame, int prot)` - Stores a mapping in the MMU.

`int USLOSS_MmuUnmap(int tag, int page)` - Removes a mapping from the MMU.

`int USLOSS_MmuGetMap(int tag, int page, int *framePtr, int *protPtr)` - Gets a mapping from the MMU.

`int USLOSS_MmuGetCause(void)` - Returns the cause of the most recent MMU interrupt.

`int USLOSS_MmuGetAccess(int frame, int *accessPtr)` - Gets the access bits.

`int USLOSS_MmuSetAccess(int frame, int access)` – Sets the access bits.

`int USLOSS_MmuSetTag(int tag)` – Sets the current tag register.

`int USLOSS_MmuGetTag(int *tagPtr)` - Gets the current tag register.

`int USLOSS_MmuSetPageTable(USLOSS_PTE *pageTable)` - Sets the current page table.

`int USLOSS_MmuGetPageTable(USLOSS_PTE **pageTable)` - Gets the current page table.