The Tera Two-Level Operating System

Tera Computer Company
2815 Eastlake Avenue East
Seattle, WA 98102

May 16, 1994

1 Introduction

The Tera computer hardware provides for four privilege levels. These privilege levels are used to protect various parts of the system from each other. One level is reserved for very early initialization and as an invalid protection indicator. One is reserved for user mode programs. The other two are for normal use by the operating system (OS). This document describes some of the motivation for using two privilege levels in the OS and discusses the way such an OS is built using the Tera tools.

2 Privilege Levels

The four privilege levels support by the Tera hardware are as follows. IPL (initial program load) is used for initializing some hardware tables such as various floating point approximation tables. It is also used as a protection level to mark program and data map entries as invalid. KERNEL is used for running the Tera OS microkernel (ukernel) and protecting its memory. SUPER is used for running the Tera OS supervisor level code including UNIX. USER is used for mapping and running user level programs.

2.1 Why Use Two OS Privilege Levels?

Operating systems are protected from unauthorized user access in most modern systems. This is accomplished by distinguishing USER from SYSTEM privilege levels. The motivation for this is that a buggy user program could potentially overwrite a portion of the OS thereby crashing the system. Additionally, there are always those users of systems that will try and find some way to either circumvent the protection and resource allocation mechanisms of an OS or else try to crash that OS. These users are called "malicious users".

While the Tera OS will need to protect against malicious users writing user level programs, it is not expected that programmers writing supervisor level code will be malicious. However, bugs can never be wholly avoided. Since the supervisor will be comprised of a much larger amount of code than the ukernel and some of that code will not have been written at Tera, it is desirable to protect the smaller and more critical portion of the OS from the rest. This is the motivation for protecting the ukernel, written entirely at Tera, from the supervisor. The small number of developers working on the ukernel should also serve to make it much more robust than the supervisor.

Note that since maliciousness is not being addressed between the supervisor and ukernel, some amount of sharing is permitted for the sake of performance. The interprocess communication (IPC) subsystem is one notable example of a piece of the ukernel that is currently directly accessible by the supervisor.

2.2 Privilege on the Tera Hardware

The privilege of a running stream (or chore) comes in to play in three different ways on the Tera system. First and foremost, the privilege level of a stream must exactly match the protection level at which the currently executing instruction is mapped. The only exception to this rule is when executing a `LEVEL_ENTER` instruction in which case the privilege level of the stream becomes the protection level at which the instruction is mapped. This is what allows a lower privileged stream take on a higher privilege level. It requires, however, that a lower privilege stream jump directly to a `LEVEL_ENTER` mapped at a higher level when a promotion is desired. Note that the `LEVEL_ENTER` instruction also specifies the privilege level from which the stream must have come from. This prevents a user stream from calling directly into the ukernel.

Next, the privilege level of a stream must be greater than or equal to the protection at which any data being accessed is mapped. This allows KERNEL level streams to look directly at supervisor or user memory but not vica versa.

Finally, there are some instructions on the Tera Hardware that require the stream executing them to be at least a level SUPER (e.g. not USER). These include resource counter selection instructions, data and program map flush operations, domain movement instructions, etc.

2.3 Changing Privilege Levels

Most modern operating systems require at least two privilege levels in the hardware. These are normally USER and SYSTEM. The transition from USER mode to SYSTEM mode is normally made by the execution of a trap instruction. This raises the privilege of the executing entity and then saves a significant part of the processor context that is making the system call. The system call trap handler then repackages the system call and actually calls the requested service routine. The trap provides a very firm protection boundary so that the user cannot trash the OS.

On the Tera system, this transition is accomplish by the use of the `LEVEL_ENTER` instruction. This serves to change the privilege level of the executing stream and requires a certain processor (stream) state to execute successfully. This provides some minimum amount of protection though there is other processor state that needs to be verified before it is used. In the transition from USER to SUPER, argument registers are verified before they are used and a new chore control block (CCB), stack, and trap handler are installed. The chore is also set up for upcalls if the system call is potentially blocking. Some of the argument verification may actually not occur until later on, e.g. for user pointer arguments. These arguments are checked by the routines used to read or write user space (generally copyin or copyout respectively).

The Tera hardware also provides for a transition from SUPER to KERNEL while within the OS. This is also accomplished by a `LEVEL_ENTER` instruction. This boundary does not so rigorously check arguments since it is expected that supervisor programmers will not be malicious where as users might be. A new CCB, stack, and trap handler are installed though. The overhead of installing the new CCB and stack is ameliorated by preallocating them when the supervisor level chore is created.

3 Building a Two-Level OS

There are a number of issues to address when constructing the two-level OS with the available Tera tools. Besides coding conventions for allocating and accessing the various data structures in the system, the mechanisms for doing the actual build must be considered.

3.1 Address Space Layout

The two-level OS needs to lay out its address space such that the ukernel and supervisor run on sets of program pages and data segments that allow themselves to be properly protected for the portion of the OS that uses them. Program pages are 1 Kwords long and the compiler tools already worry about rounding library text segments to multiples of this size. So text is never shared between ukernel and supervisor pages. Special code is run during system boot in the nucleus of the ukernel that knows where the boundary is between supervisor and ukernel text. It protects the program pages appropriately based on that knowledge. Currently, supervisor code comes first followed by ukernel code. If the amount of supervisor code overflows this boundary, the linker should complain. If so, the ukernel code can be relocated by specifying a different address when the ukernel library is built as described below.

Data segments, on the other hand, are 256 Mbytes long and the compilers make no attempt to observe this granularity when allocating data items. Instead, the tools used to build the ukernel, described below, are also used to relocate the data associated with the ukernel to another virtual segment. Currently segment 13 (0xd) is used for this purpose to avoid virtual segments required by the simulator. The supervisor data remains in segment zero.

While having the ukernel and supervisor data virtual spaces overlap is impossible given this layout, there is an issue with regard to physical memory. When running the simulator (zebra), a special flag is specified (`OS_kernel`) with a numeric argument that tells the simulator how many words of global data memory to allocate to virtual data segment zero. The rest of global data memory is allocated to segment 13 when this flag is specified. If not enough memory has been allocated for a two level run, zebra will complain about invalid addresses when the OS is being loaded. Note that when running ukernel only tests, an assertion check will ensure that there is enough segment zero memory.

3.2 The Kernel Library

Perhaps better named the "microkernel library", this subject encompasses two concepts. The exposed or exported routines that make up the library serve to define the protected ukernel resources and routines that are accessible from the supervisor level. The entire ukernel itself is also archived together to form a library that is linked against by the full system build.

3.2.1 Kernel Library Routines

The routines in the kernel library are called by calling a wrapper function at the supervisor level. This routine in turn calls a generic kernel library entry routine passing the address of the ukernel routine being called and the arguments for the call. This entry routine takes care of promoting the chore to ukernel privilege, installing the new CCB, stack, and trap handler and then calling the actual ukernel internal routine. The routines typically exported in this fashion perform the main

resource allocation functions within the ukernel and so are presented to the supervisor using this protected interface.

3.2.2 Building the Kernel Library

The entire ukernel and all necessary supporting routines are archived into one archive file called `libkl.a`. Currently included in this archive is a copy of libc, libm, the stack tracer, and the generic runtime. Eventually, the required pieces from libc, libm and the generic runtime will be migrated directly into the ukernel and so copies of these libraries will no longer be needed. Since any support routine referenced by the ukernel must also be mapped at KERNEL protection the entire archive must be resolved against itself. This is currently done by building a shared object out of the archive. Building the shared object also yields the opportunity to put the text and data defined in the library at specific well known virtual addresses. This allows the OS to establish different protections for this data and text than the protection on the rest of the supervisor level OS. The libc used here (and the other support libraries) must also have any functions that are redefined in ukernel objects actually removed from the support archive. This is needed since once all of the libraries and ukernel objects are archived together, no optional library member inclusion is allowed. All archive members are included in the shared object creating name conflicts if two versions of a routine exist.

Another part of building the archive is to hide all of the names that should not be accessible to supervisor level code. Naively, the only names that should be accessible are the kernel library generic entry routine and the kernel library routines themselves. The latter are necessary so that the corresponding wrapper routines can take the address of the ukernel functions that they eventually need to call. The export definitions for the kernel library routines are generated automatically from the kernel library interface definitions via `genKlibStubs`; the definitions are written into files (named `xxx_export`) which are then read in by the klib makefile.

Several other names also need to be accessible. One additional set of routines are those used in system initialization. These include `_start` and `main`. The sequence of routines use for system initialization is described later in this document. Some other symbols are needed by the stack tracing package as backstop routines. This includes the chore startup routines and trap handlers.

Another set includes all of the kernel level virtual functions for classes that exist at both levels. These are made supervisor accessible so that the virtual function tables that are built at supervisor level can be properly filled in. This list of names that need supervisor accessibility is defined in the `klib` makefile. The list is then used when the tera-export tool is run on `libkl.a`. *The tera-export tool has the effect of hiding all symbols not in the* `EXPLICIT_EXPORTS` *list.* Note that simply making these names accessible in the supervisor does not mean that the supervisor will be able to directly call these routines since a privilege violation will occur if the routine does not start with `LEVEL_ENTER`.

In summary, the following names are exported from the microkernel:

- all kernel library routines
- kernel library entry routine (`klib_entry`)
- system initialization (`_start`, `main`)
- backstops for tracestack

- kernel level virtual functions

3.3 Replicated Routines

There are a fair number of ukernel routines that need to be used directly at the supervisor level. These basically fall into two categories. The first consists of utility classes that are useful for independent tasks in both the supervisor and ukernel. Hash tables are an example of this in that they are used for both ukernel private data structures and separate supervisor data structures.

The second category consists of routines that must operate directly on shared data. How the data is allocated will be described later. The two main examples of this category are the IPC system and the supervisor `malloc` system. The latter is shared since there are many instances when the ukernel needs to allocate data for use in the supervisor.

The mechanism by which this code is replicated is by the use of the roadmap `EXPORT_SUPV_FILES` variables. To replicate a ukernel file to the supervisor, the appropriate export variable, `EXPORT_SUPV_SFILES` for assembler files or `EXPORT_SUPV_CFILES` for C and C++ files, in the ukernel subdirectory Makefile is initialized with the names of the files to be replicated. This causes a copy of the specified files to be made in the supervisor `import` directory, when "make kern_to_supv" is run, so that they can be recompiled at that level. Since the ukernel version of the routines have had their names hidden, no linker conflicts should occur even though two, usually identical, instances of the same routine exist in the final OS image. A Makefile exists in the import directory that contains a list of the files that have been imported there. When a file to be replicated is added to the export variable in the ukernel Makefile, it must also be added to the list of C or assembler files in the supervisor import Makefile.

There is occasionally a need in a replicated file for different code to be executed depending on which level the code is compiled for. An example of this is code that would like to directly reference ukernel data when executing in the ukernel, but must use kernel library calls to access that data when executing in the supervisor.

In summary, when adding a new replicated file the following steps must be followed:

- `#ifdef` the contents of the file as appropriate
- add the file to the local `Makefile` `EXPORT_SUPV_XFILES` list
- add the file to `supv/import/Makefile`

3.4 Shared Data

There also exists some amount of data, both statically and dynamically allocated, that must be directly sharable between the ukernel and supervisor. This data must all be mapped with SUPER protection so that streams at both privilege levels may access it. There are two methods for allocating shared data:

- dynamically allocate the data from kernel level via the supervisor level malloc
- statically allocate the data in the supervisor

3.4.1 Dynamic Allocation

The basic method for allocating dynamic shared data in the ukernel is to use the supervisor protected memory allocators. This is done during the microkernel boot sequence before beginning initialization of the supervisor. If the data type does not have constructors, a call is made to the supervisor level malloc to allocate the memory, and the data is explicitly initialized. For data types with constructors, care must be taken to invoke the constructors properly. There are two cases to consider. If all instances of a class should reside in supervisor level memory (such as the kernel IPC message buffers), the new and delete operators should be overloaded to call the supervisor malloc. This solution cannot be used where instances can be allocated from either kernel or supervisor memory. Instead, the shared data should be allocated directly via the supervisor malloc and initialized via an explicit call to an initialization routine. Most kernel classes already have explicit initialization routines defined for use in such cases.

3.4.2 Static Allocation

Naively, the way to do static supervisor level allocations would be to ifdef the static allocations of the data in the replicated files with `TOS_SUPV` which is only defined when compiling at supervisor level. Unfortunately, the way static constructors get executed make this solution unworkable.

Static constructors are constructors that must execute for data that has been statically allocated. In addition to any explicit constructors coded in C++ classes, the Tera compiler generates certain default constructors that are executed even for plain C static allocations. These compiler generated default constructors are used to initialize the full/empty bit of statically allocated `sync` and `future` variables. For sync variables, if the static allocation is initialized, the cell is set to full, otherwise it is set to empty in this default constructor. It is possible to turn off the compiler generated default constructors on a file by file basis by compiling with `-nopurge`. This is used for the file that allocates the OS `printf` lock which is instead manually purged during system boot.

Static constructors normally are executed sometime after the program starts (usually at the symbol `_start`) but before `main` is reached. This is fine for ukernel level static constructors but is too early for supervisor level constructors in the Tera system. The main reason for this is that constructors often allocate subsidiary data using `new` or `malloc`. Supervisor level constructors would need to allocate supervisor protected memory and there is no supervisor protected heap available prior to `main`. There is however a ukernel protected "boot" heap which allows the ukernel level static constructors to fire prior to `main`.

So the supervisor static constructors are fired later, allowing the subsidiary supervisor protected allocations to succeed. This is accomplished by noting the address of the constructor when trying to run the ukernel static constructors. Any constructor identified, by its address, as being at supervisor level is put on a separate static constructor list to be run later. Unfortunately, some of the shared data structures need to be used prior to the time that the supervisor static constructors fire. These include the supervisor daemon task, the supervisor `malloc`, and the IPC system.

Any structures in this category must therefore either be allocated with an overloaded `new`*, such that the supervisor allocator is used rather than the default* `new` *allocator, or else must rely on* `init` *routines instead of constructors.* With the overloaded `new` method, all of the constructors will be called at the appropriate time and so the data structure will be usable right away. This leaves only a statically allocated pointer in the supervisor level file which will never have a constructor.

This pseudocode demonstrates naive supervisor static allocation that has constructor timing problems.

```
#ifdef TOS_SUPV
SharedClass_k Foo;
#else
extern SharedClass_k Foo;
#endif
```

This example illustrates proper allocation of shared data.

```
// proper allocation of shared global data
void * SharedClass_k::operator new(size_t sz) {
    void * obj = malloc_supvCommon(sz, M_ANON, M_NOWAIT);
    if (! obj) panic_k("SharedClass_k operator new: out of memory");
    return obj;
}
void SharedClass_k::operator delete(void * obj, size_t sz) {
    _free(obj, M_ANON);
}

#ifdef TOS_SUPV
SharedClass_k * FooP;
#else
extern SharedClass_k * FooP;
#endif

void foo_init()
{
        FooP = new SharedClass_k;
}
```

Of course, overloading `new` will not work for the supervisor `malloc` structures since they cannot very well allocate themselves before they are initialized. Using the `init` method rather than relying on the constructors is necessary here. All base classes and contained classes must support `init` routines. Unfortunately, if the class is allocated statically in a supervisor level file, its constructors will still fire when the rest of the supervisor static constructors fire. While it is possible to eliminate all of the explicit static constructors, it is not possible to eliminate the default constructors and so after the class has been successfully initialized via `init`, the default supervisor static constructors come along a reinitialize the class at an inappropriate time. The solution to this is to statically allocate the memory for the class in such a way that the compiler does not think it needs to run

constructors on it. Here is a way to do it.

```
// class declaration
class supvObj;
// global pointer allocation
#ifdef TOS_SUPV
class supvObj * supvObjP;
#else
extern class supvObj * supvObjP;
#endif

// memory allocation
#ifdef TOS_SUPV
char supvObj_mem[sizeof(supvObj)];
#else
extern char supvObj_mem[];
#endif

void init_obj()
{
        supvObjP = (class supvObj *) supvObj_mem;
        supvObjP->init();
}
```

A pointer is statically allocated at supervisor level along with a character array of the correct size. Some global initialization routine assigns the pointer and calls the `init` method for the class. It is possible for single instance classes to overload the new operator to return the address of the statically allocated character array thereby again allowing use of constructors rather than `init` routines. Currently the supervisor allocator is not the only instance of its class. However, the supervisor daemon task is a single instance class and so does use an overloaded new that simply returns the address of a character array statically allocated at supervisor level. Here is how it is

done.

```
#ifdef TOS_SUPV
SvDaemonTask_k * supervisorDaemonTaskP;
char svDaemonTask_mem[sizeof(SvDaemonTask_k)];
bool svDaemonTask_allocated = FALSE;
#else
extern SvDaemonTask_k * supervisorDaemonTaskP;
extern char svDaemonTask_mem[];
extern bool svDaemonTask_allocated;
#endif

void * SvDaemonTask_k::operator new(size_t sz) {
    assert_k(!svDaemonTask_allocated);
    svDaemonTask_allocated = TRUE;
    return &svDaemonTask_mem;
}
```

3.4.3 String Constants

String constants are generally allocated by the Tera compiler in the linkage section of the routine that references them. There are several supervisor level data structures that are initialized by ukernel level routines that, if initialized using inline string constants, would end up referencing ukernel data. Instead of using inline string constants, these ukernel initialization routines need to use supervisor level pointers to string constants allocated explicitly at supervisor level. This is easily done as follows:

```
#ifdef TOS_SUPV
char * supvObjName = "object supv0bj";
#else
extern char * supvObjName;
#endif

init_obj()
{
        // incorrect use of string constant – references linkage section
        // if object is shared and init_obj is a ukernel routine this fails.
        init("object supv0bj");

        // correct initialization references supervisor static string constant
        init(supvObjName);
}
```

3.4.4 Sun and Microkernel Builds

Besides building the entire Tera version of the Tera OS, otherwise known as the Tera integrated build, several other executables are built with the same sources. These include the Sun version of the Tera OS (Sun integrated build) and both Tera and Sun versions of ukernel level tests that do not have any supervisor code. Note that the Sun versions of the OS and tests simply run as user mode programs and use a threads package for simulating parallelism.

When building and running the Sun version of the integrated build, using a two-level OS is not possible. As a matter of fact, the user level portions of the integrated build, like the shell, are actually built as part of the OS and the whole thing runs at one privilege level as a Sun user program. Also, name hiding is not available in the Sun tools. This means that no code replication can (nor needs to) be performed either. So when compiling for Sun, everything normally replicated or allocated at supervisor level for the purpose of sharing, must be allocated or defined in the ukernel. This is done by using the following ifdef in all previously given examples instead of the simple ifdef `TOS_SUPV`.

```
#if defined(TOS_SUPV) || !defined(_TERA_)
// code or shared data allocation
#endif
```

Doing this is sufficient for building all of the Sun tests and the Sun integrated build.

For the Tera ukernel tests, the data and virtual functions that are normally allocated or defined at supervisor level must be allocated or defined somewhere in the test fixture. For functions, this is currently being done by putting a stub version of the function in the file klib/supvGlobals.w that contains a `NotReachable_k()` implementation. For data, each web file defining a code fragment with global data should also generate an include file with a name of the form xxxGlobals.c that contains this allocating code fragment. This file should be listed as a header file in the directories makefile (even though its a "C" file). All of the xxxGlobals.c get included in klib/supvGlobals.w. If a new xxxGlobals.c file is generated somewhere in the ukernel, an include statement must be added to klib/supvGlobals.w. All test fixtures can then include supvGlobals.c which is generated

by this web file. Here is an example of how to set up a web file to do this.

```
@code declare global
extern SharedObj_k * FooP;

@code define global
#if defined(TOS_SUPV) || !defined(_TERA_)
SharedObj_k * FooP;
#endif

@output foo.h
@code
@declare global@

@output foo.c
@code
#include "foo.h"
@define global@
// other stuff

@output fooGlobals.c
#include "foo.h"
@define global@
```

3.5 Virtual Functions

Another problematic language feature in the two-level OS is the use of virtual functions. When a class has virtual functions, the compiler allocates a virtual function table in data memory in the file that has the definition for the first virtual function DECLARED in the class. These virtual function tables are not subject to name hiding since there is only one table per class (not class instance) regardless of whether it is accessed in both ukernel and supervisor modes. This necessitates two versions of virtual functions that need to be accessible from both the ukernel and supervisor. The convention is to prefix the functions with `uk_` and `sv_` respectively. Note that since the virtual function table must be allocated in supervisor memory so that supervisor level code can reference it, the `sv_` virtual function should be declared first. The base class containing the virtual functions

can wrapper the two with the appropriate ifdefs as follows.

```
class foo {
        int vfunc();
        virtual int sv_vfunc();
        virtual int uk_vfunc();
};

int vfunc() {
#ifdef TOS_SUPV
        return sv_vfunc();
#else
        return uk_vfunc();
#endif
}

#ifdef TOS_SUPV
int sv_vfunc()
#else
int uk_vfunc()
#endif
{
        // return something
}
```

3.5.1 Sun and Microkernel Builds

Again the additional executables being built from common sources need special ifdef'ing to build properly. The same exact discussion presented for special builds of shared data applies here for defining virtual functions. Unfortunately, this precludes ifdef'ing the virtual function definitions as shown above. Instead, the following is necessary:

```
#if defined(TOS_SUPV) || !defined(_TERA_)
int sv_vfunc()
{
        // return something
}
#endif

#ifdef TOS_KERNEL
int uk_vfunc()
{
        // return something
}
#endif
```

In addition, for those classes with virtual functions defined only at the supervisor level only the `sv_` version is needed.

3.6 Compiler Release E.2 Considerations

A few problems exist with the current E.2 compilers which require special consideration. All of these problems are advertised to be fixed in the next release of the compilers.

3.6.1 Class Static Data Members

Under the E.2 compilers, if a static data member of a class is declared but never defined, the compiler generates common block storage for the member. This is the case for some replicated classes, where the class declarations exist in both ukernel and supervisor, but the static data member is defined only in the supervisor. The intention is to have only a single instance of the static member that is shared between the ukernel and supervisor. However, the compiler allocates common block storage for static data member in the ukernel, and when the final link occurs, the linkers do not replace the common block allocation in the ukernel with the explicit allocation in the supervisor, resulting in two instances of the static member.

As a workaround for now, all static data members of replicated classes have been made into regular global variables that are referenced via `extern` in the ukernel and allocated normally in the supervisor. This prevents a common block allocation from occurring in the ukernel.

3.6.2 Tera-nm

Note that due to name hiding, not all ukernel symbols will appear in a run of tera-nm on the final integrated executable. In order to see all ukernel symbols, tera-nm must be run on the file `libkl.so`.

Also, there is a bug in the E.2 tera-nm that causes undefined symbols within an executable to mess up the sorting of symbols near the undefined symbols. There are a number of symbols that are undefined in the ukernel library that get defined at the supervisor level. Care must be used when examining addresses in the sorted klib map near these undefined symbols.

4 Runtime Considerations

In building the two-level OS, several runtime issues must be taken into account.

4.1 Exception Handlers

The exception handlers, like all code, must be mapped at the correct protection level for each part of the OS. Exception handlers, however, are installed in the ukernel for both ukernel level chores and supervisor level chores. This requires that the two versions of the handlers have different names so that both can be referenced in the ukernel. The current convention is to prefix the supervisor version with `sv_`.

Exception handlers also sometimes need to know what privilege level they are running at to do things like using the LEVEL_ENTER instruction to raise the ssw override. The exception handlers are currently replicated files that are ifdef'd to know what level they are running at.

4.2 OS Startup and Shutdown

Some rearranging of the Tera OS startup sequence was necessary in implementing the two level OS. What follows is a discussion of how the full system is brought up and shut down when running in two level mode on a single processor. Later, the way the ukernel only tests startup and shutdown is discussed. Multiprocessor initialization is still being designed.

4.2.1 Starting up the Two Level Integrated OS

The boot sequence begins at ukernel privilege in assembler code at `_start` where a boot ccb and the trap handlers are installed. The C++ routine `os_init` is then called which performs some more chore initialization, initializes the debug system, and then runs the ukernel static constructors. It then performs global and local ukernel initialization and starts a kernel daemon to execute `main`. The boot stream then quits. The kernel daemon stream picks up in `main` by starting a supervisor daemon stream to execute `os_main`. The kernel daemon stream then quits. `os_main` then fires the supervisor static constructors and calls `supv_main` which simply calls `unixmain`. The system then runs its UNIX initialization code.

4.2.2 Shutting Down the Two Level Integrated OS

The supervisor daemon running `unixmain` eventually decides to shutdown the system. Actually, any supervisor chore, daemon or promoted, can perform the following operations. `SupvShutdown` is called which runs the supervisor static destructors and makes the klib call `osShutdown` which in turn calls `os_shutdown`. `os_shutdown` runs the ukernel local and global destruct routines and then fires the ukernel static destructors. The system is now shut down.

4.2.3 Running a Ukernel Only Test

When not running in two level mode, `main` directly calls `os_main` rather than starting a supervisor daemon to do it. `os_main` then runs whatever test code is needed for the specific test. When `os_main` returns, `os_shutdown` is called automatically, shutting down the system as described above.

5 Debugging

Various problems can occur when trying to run the two level OS. This section describes some of the problems that are likely to occur.

5.1 Exceptions

The following exceptions can be triggered by mistakes in setting up the two level OS.

- privilege violation: A privilege violation occurs when a stream has insufficient privilege to execute an instruction or the instruction is invalid.
- program protection violation: A program protection violation occurs when a stream tries to execute an unmapped address or an instruction mapped at the wrong protection level, i.e. not equal to the stream's privilege level.
- data protection violation: A data protection violation occurs when a stream references an unmapped address or an address mapped at a higher protection level than the stream's privilege.
- poison trap: A poison trap can occur when a speculative load gets a data protection violation.

5.1.1 Diagnosing Exceptions

All of the trap handlers associated with these exceptions are set up to attempt a stack trace when the exception occurs. If a run yields one of these exceptions and the stack trace gives insufficient information to diagnose the problem, the following methodology is recommended for pursuing the problem. First, a checkpoint is made within a few minutes (or 10s of thousands of ticks) of the original exception. Then a breakpoint should be set on the primary trap handlers. Note that there are two primaries, one beginning with `os_primary_` the other beginning with `sv_primary_`. If it is known which one is handling the exception, only that one need be breakpointed. When the run hits the breakpoint, type "regs except" to verify that this is the exception being pursued. Keep running until the correct one is reported. It may then be possible to diagnose the problem by simply looking at the various target registers. For example, in a standard privilege or program protection violation when the supervisor incorrectly tries to call a ukernel function, t0 will contain the address of the incorrectly called function and t1 will contain the address of the caller of that function.

For data oriented exceptions, It may be possible to diagnose the problem using the result code registers and t0. Otherwise, the time of the exception can be noted and the checkpoint rerun with tracing on. For exceptions where the supervisor is referencing ukernel data, the exception will occur just after a reference using the ukernel address. These ukernel protected addresses can be easily identified by consulting the privileged common address map that accompanies this document. The usual addresses being incorrectly referenced in this manner are ukernel static data with addresses in the 0xd0000000 range and the ukernel heap in the 0x100000000 range. There are also other ukernel protected memory regions. Additionally, dereferences at or near zero will also cause a data protection map limit exception.

5.2 Multiple Definitions

If a function is multiply defined, the `shared_obj` tool will generate an error message. Here is an example.

```
shared_obj -v -Ltera -t 0x50000 -d 0xd0000000 tera/libkl.a
Error merging elf implementation files together.
Symbol (_privileged_handler.text) is being redefined in module .
relocator: common symbol (_privileged_handler) size needs to change
after its been   initialized                                        5
```

5.3 Shared Library Out of Date

Currently, if the microkernel shared library is out of date, you may get undefined results from zebra. The reason is that zebra loads the microkernel shared library dynamically, and does not detect if the library is out of date. Eventually, shared library versioning will be properly implemented and the microkernel shared library will be part of the `unix` executable, so all should work properly. In the meantime, you may get error messages such as the following:

```
% zebra unix
zebra% s
****************************************
Time 1: Last stream just quit
****************************************
zebra% quit
```

5.4 Microkernel Data and Program Segment Addresses

The following constants define addresses used to initialize the data and program maps during the boot process (protDomain.w). They must correspond to the addresses specified to the shared object tool that makes the kernel library (klib/Makefile). During boot, the protection on the program pages below the microkernel text is reduced to `LEV_SUPER`. If the size of the supervisor data or text segment grows larger than the available space, the linker will give an error message when linking the supervisor to the kernel library.

- `UKERN_START_DATA_VADDR` defines the virtual address of the microkernel data segment for the two-level OS.
- `UKERN_START_TEXT_VADDR` defines the virtual address of the microkernel program segment for the two-level OS.

These are defined in a configuration file (vmemConfig.w) that produces both the C definition (via `#define`) and definitions suitable for inclusion in a Makefile. These definitions are then included by the kernel library makefile (klib/Makefile).

5.5 Zebra Options

The following zebra option specifies the size of the OS executable data segment. It is used by zebra to size the global physical memory segment (mapped to virtual segment 0).

- `-OS_kernel 0x110000`

Zebra options are normally specified in the following setup files. Thus, if the size changes these files must be manually updated:

- `e.x.setup.csh`
- `sys/MakefileSetup`

The data segment size is computed by doing a tera-nm on the executable and grepping for the symbol `_end` (symbol `_etext` will give the size of the text segment). The size of the boot heap is defined in `protDomain.w`.

```
% tera-nm tera/test_kernelDaemonTeam | grep _end
00000000008333e8 D _end

% grep BOOT_HEAP_SIZE ../nucleus/protDomain.w | grep define
#define BOOT_HEAP_SIZE 0x10000
```

Both of these quantities are in bytes. Next, add the size of the boot heap to the size of the data segment and convert to words.

`(0x83333e8 + 0x10000) >> 3` yields the new result: `0x10867D`. Round up to get the new value: `0x110000`.

Summary of procedure to change the data segment size:

1. use tera-nm to lookup the size of the data segment
2. add the size of the boot heap (defined in protDomain.w)
3. convert from bytes to words
4. round up to leave a little extra room
5. update the setup files

Diagnostics There is an assertion in the BootMalloc initialization code that verifies that the size of the physical data segment is greater than or equal to the size of the OS data image plus the boot heap (see `mallocManager.w`). You will get the following error message if the assertion fails.

```
Assertion failed at 470: (frames2bytes(dmeP[BOOT_GLOBAL_VIRT_SEG].
getLimit()+1) >= (roundFrame((uint)&_end) + BOOT_HEAP_SIZE)),
object: No name, file mallocManager.w, line 378
```

6 Summary of Build Process

This section summarizes the steps involved in building the two level OS. As discussed earlier, the microkernel is built as a shared library. The library is built in directory `kernel/klib`. The following tools are invoked in the specified order.

1. `tera-ar`: creates an archive of all the microkernel objects
2. `tera-ranlib`: creates the archive dictionary (CAUTION: do not run `tera-ranlib` after `tera-export`).
3. `tera-export`: hides all names except the specified ones

4. `shared_obj`: creates a shared library

The exported names are generated via the `genKlibStubs` script in `klib/Makefile`. They are generated from the kernel library interface definition files (`xxx-interface.h`). There are some additional names that are explicitly exported. These are defined in `klib/Makefile`.

7 Summary of Modifications

This section summarizes what needs to be done when code modifications affect the two level OS.

- new ukernel exported name:
  - add to `EXPLICIT_EXPORTS` list in kernel library makefile (`klib/Makefile`).
- new ukernel exported file:
  - Add the name to the local Makefile's `EXPORT_SUPV_XFILES` list.
  - Add the name to `supv/import/Makefile`.
  - From the microkernel directory where the file resides, execute `make kern_to_supv` to export the file.
  - `cd` to `supv/import` and execute `make install`.
- ukernel data segment size change:
  - Replace definition of zebra options in `e.x.setup.csh` and `MakefileSetup`.
  - Check against definitions in `vmemConfig.w`.
- new supervisor level data visible at kernel level:
  - Define the data at supervisor level (for details on how to do this see section 3.4).
  - Add the definition to the appropriate `xxxGlobals.c` file.
  - Add `xxxGlobals.c` to the local Makefile `HFILES` list.
  - Add `#include xxxGlobals.c` to `supvGlobals.c`.

8 Summary of Definitions

This section summarizes the various places and files where definitions related to the two level OS exist.

- `klib/Makefile`: commands to make the kernel library
- `EXPLICIT_EXPORTS (klib/Makefile)`: list of microkernel exported names
- `klib/xxx_export`: generated automatically from `xxx-interface.w` by genKlibStubs. These are read by klib/Makefile.
- `supv/import/Makefile`: files imported from the microkernel

- `e.x.setup.csh`: size of boot global data segment (zebra option)
- `MakefileSetup`: size of boot global data segment (zebra option)
- `vmemConfig.w`: microkernel virtual text and data segment addresses

9 TODO

This section lists future enhancements, simplications, etc.

- generate supv/import/Makefile automatically