What will you open?

Solaris Device Drivers Hands-on Labs

Max Bruning
max@bruningsystems.com

solaris Java

SUN TECH DAYS 2006-2007
A Worldwide Developer Conference

Introduction

- Purpose
  - > Learn how to build, install, test, implement, and debug Solaris Device Drivers
- Labs start simple and become more complex
- References
  - > Device Driver Tutorial at http://docs.sun.com/app/docs/doc/817-5789
  - > Writing Device Drivers at http://docs.sun.com/app/docs/doc/816-4854
  - > Sections 7D, 9E, 9F, 9P, 9S or man pages
  - > Source code

Disclaimer

- Please note that these driver labs may cause your system to panic or hang.
- It is possible, though unlikely, that worse things could happen...
- Make sure to back up /etc/name_to_major, /etc/driver_aliases, and /etc/driver_classes before you begin! (A backup of /etc/path_to_inst may not be a bad idea as well...)

Lab Files

- Lab files are under .../exercises/lab1, exercises/lab2, etc.
- Solutions for each lab are in a subdirectory, Solution
- Lab exercises 2 through 4 build on the solution from the previous lab
- If something doesn't work, or you are stuck, ask for help. There is not a lot of time.

A Driver for /dev/dummy

- A driver that does nothing, but does it correctly
- Shows minimum code needed for a driver on Solaris (plus a little bit more)
- Can use as a template for drivers on Solaris
- Does not use any available framework, i.e., no STREAMS, USBA, SCSA, etc.
- You will use this driver as a starting point in the lab work
- Files: skeleton.c, skeleton.conf

/dev/dummy Source Code Include Files

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/buf.h>
#include <sys/modctl.h>
#include <sys/open.h>
#include <sys/kmem.h>
#include <sys/poll.h>
#include <sys/conf.h>
#include <sys/cmn_err.h>
#include <sys/stat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

/dev/dummy Source Code Function Prototypes

```
/* more prototypes added as labs progress, not all are shown here */
/* prototypes can be found in corresponding man pages (man open.9e) */
/* and header file /usr/include/sys/devops.h */

static int skel_open(dev_t *devp, int flag, int otyp, cred_t *cred);
static int skel_read(dev_t dev, struct uio *uiop, cred_t *credp);
static int skel_write(dev_t dev, struct uio *uiop, cred_t *credp);
static int skel_getinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
    void **result);
static int skel_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int skel_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

/dev/dummy Source Code Data Structures

```
/*
 * The entire state of each skeleton device.
 */
typedef struct {
    dev_info_t  *dip;       /* my devinfo handle */
} skel_devstate_t;

/*
 * An opaque handle where our set of skeleton devices live
 */
static void *skel_state;
```

/dev/dummy Source Code Data Structures Continued

```
static struct cb_ops skel_cb_ops = { /* see cb_ops(9s) for details */
    skel_open,
    nulldev,    /* close */
    nodev,      /* strategy */
    nodev,      /* print */
    nodev,      /* dump */
    skel_read,
    skel_write,
    nodev,      /* ioctl */
    nodev,      /* devmap */
    nodev,      /* mmap */
    nodev,      /* segmap */
    nochpoll,   /* poll */
    ddi_prop_op,
    NULL,       /* streamtab */
    D_NEW | D_MP,
    CB_REV,
    nodev,      /* aread */
    nodev       /* awrite */
};
```

/dev/dummy Source Code Data Structures Continued

```
static struct dev_ops skel_ops = {  /* see dev_ops(9s) */
    DEVO_REV,
    0,                  /* refcnt */
    skel_getinfo,
    nulldev,                /* identify */
    nulldev,                /* probe */
    skel_attach,
    skel_detach,
    nodev,              /* reset */
    &skel_cb_ops,
    (struct bus_ops *)0,
    nodev               /* power */
};
```

/dev/dummy Source Code Data Structures Continued

```
extern struct mod_ops mod_driverops;

static struct modldrv modldrv = {     /* see modldrv(9s) */
    &mod_driverops,
    "skeleton driver v1.0",
    &skel_ops
};

static struct modlinkage modlinkage = {  /* see modlinkage(9s) *//
    MODREV_1,
    &modldrv,
    0
};
```

/dev/dummy Source Code (Un)Loading Routines

```
int
_init(void)  /* see _init(9e) */
{
    int e;

    if ((e = ddi_soft_state_init(&skel_state,
        sizeof (skel_devstate_t), 1)) != 0) {
        return (e);
    }

    if ((e = mod_install(&modlinkage)) != 0)  {
        ddi_soft_state_fini(&skel_state);
    }

    return (e);
}
```

/dev/dummy Source Code (Un)Loading Routines Continued

```
int
_fini(void)  /* see _fini(9e) */
{
    int e;

    if ((e = mod_remove(&modlinkage)) != 0)  {
        return (e);
    }
    ddi_soft_state_fini(&skel_state);
    return (e);
}

int
_info(struct modinfo *modinfop)  /* _info(9e) */
{
    return (mod_info(&modlinkage, modinfop));
}
```

/dev/dummy Source Code Attach Routine

```
static int  /* called for each device instance, see attach(9e) */
skel_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    skel_devstate_t *rsp;

    switch (cmd) {
    case DDI_ATTACH:

        instance = ddi_get_instance(dip);

        if (ddi_soft_state_zalloc(skel_state, instance)
            != DDI_SUCCESS) {
            cmn_err(CE_CONT, "%s%d: can't allocate state\n",
                ddi_get_name(dip), instance);
            return (DDI_FAILURE);
```

/dev/dummy Source Code Attach Routine Continued

```
    } else
        rsp = ddi_get_soft_state(skel_state, instance);

    if (ddi_create_minor_node(dip, "skel", S_IFCHR,
        instance, DDI_PSEUDO, 0) == DDI_FAILURE) {
            ddi_remove_minor_node(dip, NULL);
            goto attach_failed;
    }

    rsp->dip = dip;
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

default:
    return (DDI_FAILURE);
}
```

/dev/dummy Source Code Attach Routine Continued

```
attach_failed:
    (void) skel_detach(dip, DDI_DETACH);
    return (DDI_FAILURE);
}
```

/dev/dummy Source Code Detach Routine

```
static int  /* see detach(9e) */
skel_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int instance;
    register skel_devstate_t *rsp;

    switch (cmd) {
    case DDI_DETACH:
        ddi_prop_remove_all(dip);
        instance = ddi_get_instance(dip);
        rsp = ddi_get_soft_state(skel_state, instance);
        ddi_remove_minor_node(dip, NULL);
        ddi_soft_state_free(skel_state, instance);
        return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
    }
```

/dev/dummy Source Code Open Routine

```
/*ARGSUSED*/
static int  /* called on open(2), see open(9e) */
skel_open(dev_t *devp, int flag, int otyp, cred_t *cred)
{
    if (otyp != OTYP_BLK && otyp != OTYP_CHR)
        return (EINVAL);

    if (ddi_get_soft_state(skel_state, getminor(*devp)) == NULL)
        return (ENXIO);

    return (0);
}
```

/dev/dummy Source Code Read/Write Routines

```
static int
skel_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int instance = getminor(dev);
    skel_devstate_t *rsp = ddi_get_soft_state(skel_state, instance);
    return(0);

}

/*ARGSUSED*/
static int
skel_write(dev_t dev, register struct uio *uiop, cred_t *credp)
{
    int instance = getminor(dev);
    skel_devstate_t *rsp = ddi_get_soft_state(skel_state, instance);
    return(0);
}
```

/dev/dummy Source Code The *driver*.conf File

- Driver configuration file containing properties of the device
- Optional, not necessary for PCI devices

```
bash-3.00$ cat skeleton.conf

name="skeleton" parent="pseudo";
bash-3.00$
```

Compiling a Driver

- 32-bit x86, gcc

```
bash-3.00$ gcc -D_KERNEL -c -O foo.c bar.c
```

- 32-bit x86, SunStudio

```
bash-3.00$ cc -D_KERNEL -c -O foo.c bar.c
```

- 64-bit amd64, gcc

```
bash-3.00$ gcc -D_KERNEL -m64 -mcmodel=kernel -c  -O foo.c bar.c
```

- 64-bit amd64, SunStudio

```
bash-3.00$ cc -D_KERNEL -xarch=amd64 -xmodel=kernel -c -O foo.c bar.c
```

Creating a Driver Kernel Module

- Create a re-locatable object

```
bash-3.00$ ld -r -o foobar foo.o bar.o
```

- If needed, specify dependencies on other kernel modules using the “-N” option to ld(1), multiple -N options are allowed
  - For instance, if your driver is using GLDv3:

```
bash-3.00$ ld -r -dy -Nmisc/mac -o foobar foo.o bar.o
```

Installing the Driver

- Copy the *driver*.conf file, if needed, to /usr/kernel/drv

bash-3.00$ cp foobar.conf /usr/kernel/drv

- Copy the driver module to /usr/kernel/drv on 32-bit x86, /usr/kernel/drv/amd64 on amd64, or /usr/kernel/drv/sparcv9 on SPARC

bash-3.00$ cp foobar /usr/kernel/drv/amd64/foobar

- During development, (specifically when you are not sure your _init() and attach() functions are correct):

bash-3.00$ cp foobar /tmp/foobar
bash-3.00$ ln -s /tmp/foobar /usr/kernel/drv/amd64/foobar

Installing the Driver (Continued)

- Use the add_drv(1M) command

```
bash-3.00$ add_drv foobar
bash-3.00$
```

- Note that any output from add_drv indicates a problem
  - Check /var/adm/messages for error output
  - Once you identify the problem, you'll need to rem_drv foobar before running add_drv again
- Check for the device file in the /devices tree

Testing the Driver

- Repeated add_drv/rem_drv commands should work (and not cause memory leaks)
- You'll need to write a program to test ioctl, mmap, and poll calls
- You should be able to use existing tools to test read/write

```
bash-3.00$ while true
> do
> add_drv foobar
> echo hello > /devices/...      <- your device file
> cat < /devices/... > /dev/null
> done
```

- Make sure your driver handles multiple device instances correctly
- You can add instances for pseudo devices by adding lines to the *driver*.conf file

```
name="ramdisk" parent="pseudo" instance=0;
name="ramdisk" parent="pseudo" instance=1 disk-size=512
```

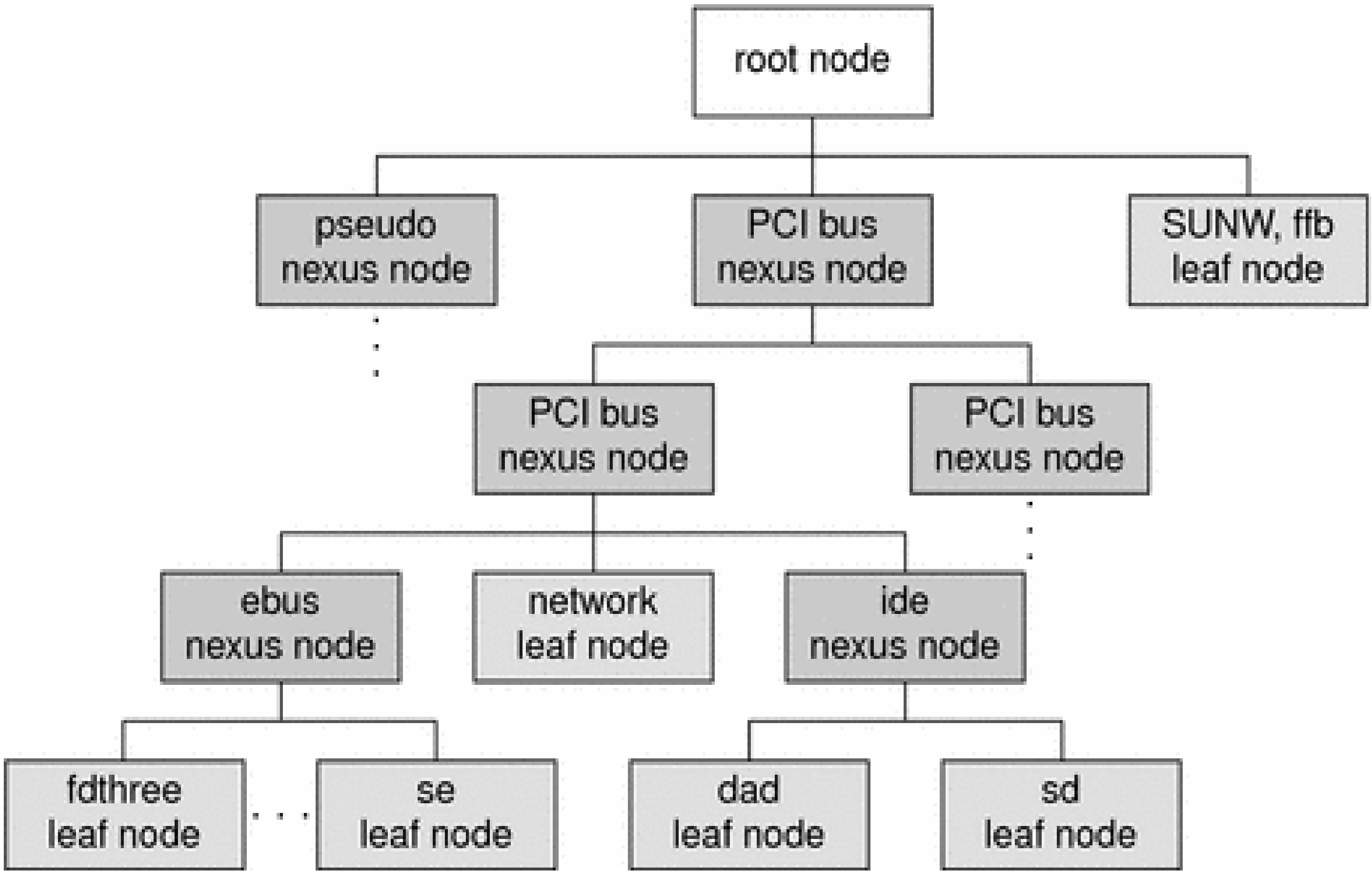
Exercise 1: Build, Install, and Test a Driver

0. Lab files are provided on CD, DVD, or available via NFS. Make a directory on your system and place the files in that directory. In the following, assume the material is in /export/home/drivers.
1. Cd exercises/lab1
2. Compile and link the skeleton driver provided.
3. Copy the skeleton module and skeleton.conf module to the kernel/drv directory
4. Run the add_drv command for the skeleton.
5. Check that the device file exists
6. Test using multiple add_drv, rem_drv, echo, and cat commands.

The Device Tree

- Geographical representation of devices on the system
- Built initially at boot time
- Each node on the tree represented internally by a struct dev_info
- Can be viewed by prtconf(1M) and by ls -lR /devices
  - libdevinfo(3LIB) provides a programmatic interface
- Can be modified at run time via dynamic reconfiguration
- Can also be viewed via mdb -k
  - ::walk devinfo | ::print struct dev_info
- If the device is not present in the device tree, the kernel can not "see" the device
- Kernel walks the tree at boot time trying to match each node on the tree with a device driver
- Intermediate nodes are called "nexus" nodes
- End nodes are "leaf" nodes
- Each node on the tree has a "name" property and optionally a "compatible" property used to determine which device driver to use for the device

The Device Tree – An Example



Choosing the Driver for a Device

- Every node in the device tree has a **name** property, and optionally, a **compatible** property
- The system first scans the list of names in the **compatible** property to find a driver with a matching name
- If none found, or there is no **compatible** property, the system tries to find a match on the **name** property
- If no match is found, there is no driver
- The system uses the **/etc/driver_aliases** file to find matching names
- Drivers are, by default, in **/platform/*xxx*/kernel/drv, /kernel/drv, and /usr/kernel/drv**
- For x86 and x64, *"xxx"* is **i86pc**, for SPARC, *"xxx"* is (typically) **sun4u**
- x64 drivers are in the **amd64** subdirectory, SPARC drivers are in the **sparcv9** subdirectory
- **driver.conf** files are in the **drv** directory

Choosing the Driver for a Device – An Example

```
# prtconf -vp
...
   pci, instance #0
...
      Node 0x000005  <-- device is on first PCI bus
         compatible: 'pci1002,5950.10cf.1301.1' + 'pci1002,5950.10cf.1301'
 + 'pci10cf,1301' + 'pci1002,5950.1' + 'pci1002,5950' + 'pciclass,060000'
 + 'pciclass,0600'  <-- compatible property
...
         name:  'pci10cf,1301'  <-- name property
...
# prtconf -D | grep 10cf,1301
      pci10cf,1301, instance #0 (driver name: skeleton1)
# grep skeleton /etc/driver_aliases
skeleton "pci10cf,1301"
skeleton1 "pci1002,5950.10cf.1301.1"
#
```

Exercise 2: Use the Skeleton Driver for an Existing Device

- For this lab, you will use the skeleton driver provided in exercise 1 as the driver for an existing PCI device on your system.
- You need a device which does not have a driver, or a device which has a driver, but which you are not currently using
- Note that the device will not work, but the driver should successfully install and attach.
- The **pci10cf,1301** device shown on the previous page is an example
- At the end, you should “see” the device file under **/devices/...** where “**...**” is some PCI subdirectory
- Note that the device should be a leaf node (or a nexus with nothing attached)
- Don't forget to back up **/etc/name_to_major, /etc/driver_aliases, /etc/driver_classes, and /etc/path_to_inst**
- You will use this driver with your device again in Exercise 3

Exercise 2: Lab Steps

1. Make copies of **/etc/name_to_major, /etc/driver_aliases, /etc/driver_classes, and /etc/path_to_inst**
2. Use **prtconf -D** to find a PCI device that does not have a driver. If you don't have a PCI device without a driver, pick a PCI device that you are not currently using (audio is often a good choice). If you don't have PCI on your system, see the instructor.
3. If you are replacing an existing driver, **rem_drv(1M)** the driver. Otherwise, skip this step. Note this may require reboot if the driver can not be unloaded.
4. Remove **/usr/kernel/drv/skeleton.conf**
5. Use **add_drv -i \\"*compatible_property_name*\\" skeleton**
   - Get compatible property from **prtconf -vp**
6. Assuming step 5 is successful, **prtconf -D | grep skeleton** should show the skeleton driver being used with your device
7. Use **ls -lR /devices | grep skel** to find your device and driver
8. **cat(1)** the device file
9. **rem_drv skeleton**

Working with Hardware

- Device Registers and Memory Space
  - Used to configure, retrieve status, program, and minimally start I/O
    - Hardware documentation should describe how to use the registers
  - Access is done via a *handle* retrieved by ddi_regs_map_setup(9F)
    - ddi_get/ddi_put routines retrieve/set values (unless space is memory mapped) See ddi_get8(9f)
- Interrupt Handling
  - Driver routines handle asynchronous events from the hardware
  - Driver registers interrupt handler(s) with system via ddi_intr_add_handler(9f)
  - Interrupt handler may be called when device is <u>not</u> interrupting, and when device is not fully initialized
  - Handler must acknowledge interrupt
- DMA – Driver sets up data transfer between memory and device, device does the transfer

Working with Hardware – PCI Address Spaces

- Devices attached to PCI buses have:
  - Config space
    - 256-byte area, first 64-bytes are well-defined (see sys/pci.h). Remaining bytes are vendor/device specific
    - May be memory-mapped or use I/O ports
    - See pci_config_setup(9f)
  - I/O Space
    - Cptional device specific area(s)
    - Use ddi_regs_map_setup(9f) to initialize access, ddi_get/ddi_put routines to access/modify (see ddi_get8(9f))
  - Memory Space
    - Optional device specific area(s)
    - Use ddi_regs_map_setup(9f) to initialize access
    - Use kernel virtual addresses (from ddi_regs_map_setup(9f)) to access/modify

Accessing Device Registers/Memory

```
#include <sys/pci.h>
/* assume device is PCI */
skel_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    ddi_device_acc_attr_t dev_attr;
    caddr_t regp;
    ddi_acc_handle_t regh;
    unsigned short x;
    /* specify device attributes */
    dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
    /* get a handle for register/space access */
   /* register set 0 (2nd arg) is PCI config space */
    ddi_regs_map_setup(dip, 0, &regp, 0, 0,  &dev_attr, &regh);
    /* retrieve value of 16-bit register */
    x = ddi_get16(regh,     (unsigned short *)(regp+PCI_CONF_VENID)));
```

Interrupt Handling

- Called when the device interrupts
  - Mechanism of how interrupt handler is called is not important (unless you are porting Solaris to new hardware!)
- Interrupts are assigned priorities
  - Defaults for PCI devices are based on value of **class-code** config space register
  - Interrupts above priority 10 are "high level" interrupts
    - Driver should have special handling for these
    - Mutexes initialized with high level priority are *spin locks*
  - Driver may specify priority via ddi_intr_set_pri(9f)
- Interrupts can be "fixed" or MSI
- Interrupt handlers must return either DDI_INTR_CLAIMED or DDI_INTR_UNCLAIMED
- Driver should also tell device that the interrupt has been handled
- Search for ddi_intr_add_handler in the source code for various examples

Interrupt Handling - Setup

```
/* foo_intr() is an interrupt handler */
/* setup is typically in attach(9f) */
extern uint_t foo_intr(caddr_t arg1, caddr_t arg2);
foo_state_t foo_state;

foo_attach(dev_info_t *dip, int cmd)
{
    int         avail, actual, intr_size, count = 0;
    int         i, flag, ret;
    ddi_intr_handle_t  *htable;  /* normally in driver state */
    int         intr_types;
    int         intr_type, intr_cnt, intr_cap;
    unsigned intintr_pri;
```

Interrupt Handling – Setup (Continued)

```
ddi_intr_get_supported_types(dip, &intr_types);

/* should check device support for this type */
if (intr_types & DDI_INTR_TYPE_MSIX) {
  intr_type = DDI_INTR_TYPE_MSIX;
  ddi_intr_get_nintrs(dip, intr_type, &count);
} else if (intr_types & DDI_INTR_TYPE_MSI) {
  intr_type = DDI_INTR_TYPE_MSI;
  ddi_intr_get_nintrs(dip, intr_type, &count);
} else {
  intr_type = DDI_INTR_TYPE_FIXED;
  ddi_intr_get_nintrs(dip, intr_type, &count);
}
```

Interrupt Handling – Setup (Continued)

```
/* Allocate an array of interrupt handles */
intr_size = count * sizeof (ddi_intr_handle_t);
htable = kmem_alloc(intr_size, KM_SLEEP);

/* Call ddi_intr_alloc() */
ret = ddi_intr_alloc(dip, htable, intr_type, 0,
    count, &actual, DDI_INTR_ALLOC_NORMAL);

/* Call ddi_intr_add_handler() */
for (i = 0; i < actual; i++)
  ddi_intr_add_handler(htable[i], foo_intr, (caddr_t)&foo_state,
              (caddr_t)(uintptr_t)i);
```

Interrupt Handling – Setup (Continued)

```
/* before enabling interrupts, there may be some */
/* device specific work to be done */
ddi_intr_get_cap(htable[0], &intr_cap);

    if (intr_cap & DDI_INTR_FLAG_BLOCK) {
        (void) ddi_intr_block_enable(htable, intr_cnt);
    } else {
        for (i = 0; i < intr_cnt; i++) {
            (void) ddi_intr_enable(htable[i]);
        }
    }
```

Interrupt Handling – Interrupt Handler

```
uint_t
foo_intr(caddr_t arg1, caddr_t arg2)
{
    uint_t result = DDI_INTR_UNCLAIMED;
    int status;

    /* the following is not needed for MSI interrupts */
    if (intr_type == DDI_INTR_TYPE_FIXED) {
        status = ddi_get32(reghandle,
            (uint32_t)(regp+STATUS_OFFSET));
        if (status != INTERRUPTING)
            return (result);
    }
    result = DDI_INTR_CLAIMED;
```

Interrupt Handling – Interrupt Handler (Continued)

```
    /*
     * Tell the chip that we're processing the interrupt
     */
    ddi_put32(reghandle, (uint32_t)(regp+CONTROL_OFFSET),
IPROCESSED);

    /* handle I/O completion, status changes, new I/O setup, etc. */
    ...
    return (result);
}
```

DMA (Direct Memory Access)

- Device is responsible for transfer of data between device and memory
- Driver must:
  - Setup/Teardown DMA Mappings
  - When needed, build *scatter/gather* list
  - Synchronize CPU and I/O caches
- Most of the work is done by the ddi_dma_*xxx* routines, which make appropriate calls into a nexus driver
  - There are over 40 routines documented in /usr/man/man9f
- Generally, drivers should not be concerned with physical/virtual addresses
- Proper use of the DDI routines should make driver source level compatible across platforms and OS versions, without the use of ifdefs

DMA Attributes

- To setup DMA, first define a ddi_dma_attr_t (see ddi_dma_attr(9S))

```
static ddi_dma_attr_t foo_dma_attr = {
    DMA_ATTR_V0,            /* dma_attr version */
    0x0000000000000000ull,  /* dma_attr_addr_lo   */
    0xFFFFFFFFFFFFFFFFull,  /* dma_attr_addr_hi   */
    0x00000000FFFFFFFFull,  /* dma_attr_count_max */
    0x0000000000000001ull,  /* dma_attr_align     */
    0x00000FFF,             /* dma_attr_burstsizes */
    0x00000001,             /* dma_attr_minxfer   */
    0x0000000000000FFFull,  /* dma_attr_maxxfer   */
    0xFFFFFFFFFFFFFFFFull,  /* dma_attr_seg       */
    1,                      /* dma_attr_sgllen    */
    0x00000001,             /* dma_attr_granular  */
    DDI_DMA_FLAGERR         /* dma_attr_flags */
};
```

DMA Setup – Allocate a DMA Handle

- Done in attach(9F) or before I/O transfer

```
ddi_dma_handle_t foo_dma_hdl;  /* typically in device state struct */

ddi_dma_alloc_handle(dip, &foo_dma_attr,
        DDI_DMA_DONTWAIT, NULL, &foo_dma_hdl);
```

- DDI_DMA_DONTWAIT
  - or DDI_DMA_SLEEP
  - or *callback* function to be called when resources may be available
- NULL – argument to callback function

DMA Setup – Allocate DMA Memory

- Done in attach(9F) or before I/O transfer
- Not needed with buf(9S) structures

```
    ddi_dma_handle_t dma_handle;/* returned by
ddi_dma_alloc_handle(9F) */
    size_t memsize;  /* number of bytes to be DMA-ed */
    ddi_device_acc_attr_t foo_acc_attr;  /* endianess, ordering */
    uint_t dma_flags;  /* consistent or streaming and cache attributes */
    caddr_t *kaddrp;  /* returned address */
    size_t real_length;  /* actual size returned */
    ddi_acc_handle_t  /* opaque for use with ddi_get/ddi_put if needed */

    err = ddi_dma_mem_alloc(dma_p->dma_hdl, memsize, attr_p,
        dma_flags, DDI_DMA_DONTWAIT, NULL, &va, &dma_p-
>alength,
        &dma_p->acc_hdl);
```

DMA Setup – Bind Handle to Memory

- Done in attach(9F) or before I/O transfer
- Use ddi_dma_buf_bind_handle(9F) with buf(9S) structures

```
   ddi_dma_handle_t dma_handle;/* returned by
ddi_dma_alloc_handle(9F) */
   caddr_t kaddrp;  /* returned address */
   size_t real_length;  /* number of bytes to be transferred */
   uint_t dma_flags;  /* direction, allow partial mappings, etc. */
   ddi_dma_cookie_t  dmac; /* returned by ddi_dma_addr_bind_handle */
   uint_t cookiecnt;  /* number of dma cookies returned */

   ddi_dma_addr_bind_handle(dma_handle, NULL,  /* NULL means
kernel */
      &kaddrp, real_length, dma_flags, DDI_DMA_DONTWAIT, NULL,
      &dmac, &cookiecnt);
```

DMA – Program the Device

- Done when DMA is to be performed
- See the Writing Device Driver Guide (http://docs.sun.com) for information on handling partial mappings

```
ddi_acc_handle_t regh;  /* retrieved from ddi_regs_map_setup */

for (i = cookiecnt; i != 0; i--) {
    ddi_put64(regh, (uint64_t)(regp+sglst[i]), dmac.dmac_laddress);
    ddi_put64(regh, (uint64_t)(regp+sglen[i]), dmac.dmac_size);
    if ( i > 1)
    ddi_dma_nextcookie(dma_handle, &dmac);
}
```

Exercise 3: Print Vendor/Device ID

- For this exercise, you will modify the attach routine in the skeleton driver from exercise 2 to map and print the vendor and device IDs from PCI config space for your chosen device.
- You should add a ddi_acc_attr_t structure, and calls to ddi_regs_map_setup(9F) to gain access to the config space registers
  - The config space registers are in register set 0
  - The header file, /usr/include/sys/pci.h has defines for the config space register offsets
- Then use ddi_get16(9f) to retrieve the Vendor ID and Device ID
- Alternatively, you can use pci_config_setup(9F) instead of ddi_regs_map_setup(9F)
- Use cmn_err(9F) to print the values to the console
- When the driver is attached, you should see the vendor ID and device ID in /var/adm/messages
- Use prtconf -vp to check your work

Ioctl Handling

- Ioctl(2)/ioctl(9F) provides a mechanism to allow application code to retrieve status or set configuration information for a device. In fact, ioctl can be used for just about anything, including implementing I/O.
- Drivers should be able to handle both 32-bit and 64-bit applications
  - Driver itself should be compilable for both 32-bit and 64-bit Oses
- Ioctls that are meant to be “public” interfaces should be documented in a man page for the driver.
  - See /usr/man/man7d/* for examples

Ioctl Handling – An Example

- The following should be in a header file that can be included by both the driver and any application wishing to use the ioctl.

```
/* set of commands handled by driver */
#define GETSTATUS (('f'<< 8)|01)
#define SETCTL    (('f'<< 8)|02)

/* structures for 3rd arg for each command */
struct foostatus {
  short status;
};

struct fooctl {
  short ctl;
};
```

Ioctl Handling – Example Driver Code

```
foo_ioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *crp, int *rvalp)
{
  short stat, ctl;
  switch(cmd) {
  case GETSTATUS:
    /* retrieve status of device */
    ddi_copyout(&stat, (void *)arg, sizeof(stat), mode);
    *rvalp = 0;
    return DDI_SUCCESS;
  case SETCTL:
    ddi_copyin(arg, &ctl, sizeof(ctl), mode);
    /* set contol info on device */
    *rvalp = 0;
    return DDI_SUCCESS;
  default:
    return EINVAL;
  }
}
```

Ioctl Handling – Application Code

```
#include "foo.h"  /* header file included by driver and app */
int
main(int argc, char *argv[])
{
  struct foostatus fs;
  int fd, rval;

  fd = open(argv[1], 0);
  rval = ioctl(fd, GETSTATUS, &fs);
  exit(0);
}
```

Exercise 4: ioctl

- For this exercise, you will modify the driver from exercise 3 to implement an ioctl. This ioctl should retrieve the values of the vendor and device IDs and pass them to the user. Use the following structure:

```
struct skel_arg {
    short vendorid;
    short deviceid;
};
```

- The ioctl should return the value of **lbolt** (see ddi_get_lbolt(9f)).
  - Note: this should be ok as long as your system has not been running more than 2**32 clock ticks.
- You need to write the driver ioctl routine, add it to the cb_ops structure, and rebuild/reinstall your driver.
- Then you need to write a test application.

Debugging Driver Problems

  - Useful for post-mortem analysis
  - Can be used also on live system
- kmdb(1)
  - Like mdb, but can set break/watchpoints and single-step in kernel
  - Good for analyzing hung systems (provided you can get to kmdb)
- dtrace(1M)
  - Function bound tracing with arguments and return values
  - Also useful as a code coverage tool for testing
- See Solaris Modular Debugging Guide at http://docs.sun.com/app/docs/doc/816-5041 for details on using mdb/kmdb
- See Solaris Dynamic Tracing Guide at http://docs.sun.com/app/docs/doc/817-6223 for dtrace
- The Writing Device Driver Guide at http://docs.sun.com/app/docs/doc/816-4854 also has useful information about debugging driver problems.
- See also Solaris internals and Crashdump Analysis on x86/x64 platforms http://opensolaris.org/os/community/documentation/files/book.pdf

Debugging Drivers – Crashes

- If your system crashes, 2 files are placed in /var/crash/*hostname*
  - vmcore.# - a (Kernel) memory image, # is 0 for the first dump, 1 for the second, etc.
  - unix.# - a symbol table listing.
  - Generally, you will get vmcore.0/unix.0, vmcore.1/unix.1, etc.
- Of course, your system has to reboot in order for you to get and access these files.
  - If the system doesn't reboot during these labs, ask for help.
  - If your system doesn't reboot at other times, you can ask for help on the web (assuming you can get there).
- For more complete coverage of tools, read the documentation, or take a course!
- The next several pages will cover a very small subset of the things you can dc with mdb.

Debugging Drivers – Crashes

- If your system crashes, 2 files are placed in /var/crash/*hostname*
  - vmcore.# - a (Kernel) memory image, # is 0 for the first dump, 1 for the second, etc.
  - unix.# - a symbol table listing.
  - Generally, you will get vmcore.0/unix.0, vmcore.1/unix.1, etc.
- Of course, your system has to reboot in order for you to get and access these files.
  - If the system doesn't reboot during these labs, ask for help.
  - If your system doesn't reboot at other times, you can ask for help on the web (assuming you can get there).
- For more complete coverage of tools, read the documentation, or take a course!
- The next several pages will cover about 5% of mdb you'll need to know in order to do 90% of the debugging you'll do.

Getting Started with mdb

- First, we'll look at running mdb on a kernel crash dump. This crash occurred while testing a solution for the fourth exercise.

```
# cd /var/crash/unknown
# ls
bounds    unix.0    vmcore.0
# mdb 0

> ::status
debugging crash dump vmcore.0 (64-bit) from unknown
operating system: 5.11 snv_55b (i86pc)
panic message:
BAD TRAP: type=d (#gp General protection) rp=fffffe8001261b90
addr=fef28420
dump content: kernel pages only
```

Stack Backtrace

- The **$c** command shows a stack backtrace for the thread running at the time of the crash.
  - Stack backtraces for other threads can be found using *stack_address*$c, or by the *thread_address*::findstack command
  - Note that some frames may be missing due to optimization

```
$c
ddi_get16()
cdev_ioctl+0x48(d100000000, 7301, 80472d8, 100001, ffffffff8488c918,
fffffe8001261e9c)
spec_ioctl+0x86(ffffffff81735840, 7301, 80472d8, 100001, ffffffff8488c918,
fffffe8001261e9c)
fop_ioctl+0x37(ffffffff81735840, 7301, 80472d8, 100001, ffffffff8488c918,
fffffe8001261e9c)
ioctl+0x16b(3, 7301, 80472d8)
sys_syscall32+0x101()
```

Stack Backtrace (Continued)

- The **$c** command shows cdev_ioctl() calling ddi_get16. The driver is not in the trace. Let's check this...

cdev_ioctl+48::dis -n 2 <- *disassemble 2 lines around this location*
cdev_ioctl+0x42: xorl %eax,%eax
cdev_ioctl+0x44: call *0x38(%r10) <- an indirect call
cdev_ioctl+0x48: leave
cdev_ioctl+0x49: ret
0xffffffffb962ada: nop

Stack Backtrace (Continued)

- A closer look at the stack. Here, the value of the stack pointer is used to examine the top 6 elements on the stack. Note that this would be **<esp** on 32-bit x86, and **<sp** on SPARC.

```
<rsp,6/apn
0xfffffe8001261c88:     skel_ioctl+0x72
0xfffffe8001261c90:     1
0xfffffe8001261c98:     0xffffffff83029438
0xfffffe8001261ca0:     0
0xfffffe8001261ca8:     0x67e
0xfffffe8001261cb0:     0xfffffe8001261e9c
```

Function Call Sequence

- On x86 platforms, arguments to functions are pushed on the stack, then the function is called. Calling the function pushes the return location on the stack.
- Typically, the called function pushes the frame pointer (**%ebp**) onto the stack and make the current stack pointer (**%esp**) the new frame pointer. Then the called function decrements the stack pointer by the number of bytes needed for local variables.
- On X64 platforms (for 64-bit kernels and applications), the first 6 arguments are placed into registers, then the function is called, which pushes the return location. The same steps are taken to save the frame pointer (**%rbp**) and move the stack pointer (**%rsp**) into the frame pointer.
- SPARC also puts arguments into registers. When a function is called, the location where the function is called from (not the return location) is pushed on the stack.
- Typically, the called function executes a **save** instruction to create a new stack frame.
- An optimization is to skip the new frame setup (by omitting the push of the frame pointer/store stack pointer into frame pointer, or omitting the **save**)

Where does the Panic Occur?

- The panic() code save the value of the registers at the time of the panic. The *program counter* (**%rip** on x64, **%eip** on x86, and **%pc** on SPARC) contains the address of the executing instruction when the BAD TRAP occurred. In the following, the system panics at the first instruction in **ddi_get16()**. Note that this is not a push of the frame pointer (or **save** on SPARC).

<rip/i <- disassemble 1 instruction at the current program counter
ddi_get16:
ddi_get16: movl 0x68(%rdi),%edx

- The only way a BAD TRAP can happen here is if the **%rdi** register contains an invalid address.

<rdi=K <- what is the (64-bit) value of %rdi?
6e65706f696e76
<rdi/K <- indirect through this value
mdb: failed to read data from target: no mapping for address
0x6e65706f696e76: <- not a valid address, looks like ascii?
<rdi=s <- show value of %rdi as a string
vniopen

What Calls ddi_get16?

- Since ddi_get16() does not set up a stack frame or modify the stack poiner (at least not in the first instruction), the return location to the calling function should be at the top of the stack.

```
<rsp/p  <- show contents at current stack pointer as symbol
0xfffffe8001261c88:          skel_ioctl+0x72
```

- So, skel_ioctl() calls ddi_get16()
- **%rdi** contains the first argument. This should be a **ddi_acc_handle_t**,but instead is a string.
- Examination of the code reveals that the **skel_attach()** function was allocating space, mapping the registers, printing some values, and then unmapping the registers and de-allocating the space. The **skel_ioctl()** function was using the freed space causing the bug.

Other Useful mdb Commands

```
::cpuinfo -v  <- show synopsis of each cpu
 ID ADDR            FLG NRUN BSPL PRI RNRN KRNRN SWITCH
THREAD          PROC
  0 fffffffffbc2f260  1b    6    0  59   no    no t-0    ffffffff83db3760
skelapp
              |    |
     RUNNING <--+    +-->  PRI THREAD          PROC
      READY              60 fffffe80014dcc80 sched
     EXISTS              58 ffffffff838c1b20 sh
     ENABLE               49 ffffffff84eb3100 xemacs-21.4.12
                        49 ffffffff84fa77e0 java
                        49 ffffffff847e80e0 java
                        49 ffffffff8460cb60 gnome-terminal

::log tlist  <- log mdb output to file "tlist"
mdb: logging to "tlist"
```

Other Useful mdb Commands

```
::threadlist -v  <- show synopsis of all kernel threads, useful for hangs
        ADDR             PROC              LWP CLS PRI            WCHAN
...
fffffff83db3760 fffffff84920a38 fffffff83de3830   2  59                0
  PC: panicsys+0x7b    CMD: ./skelapp /devices/pci@0,0/pci10cf,1301@0:skel
  stack pointer for thread fffffff83db3760: fffffe8001261600
    0xfffffff80cf6e88()
    i_ddi_prop_search+0x57()
    ddi_prop_search_common+0x22c()
    0xd()
    ddi_get16()
    cdev_ioctl+0x48()
    spec_ioctl+0x86()
    fop_ioctl+0x37()
    ioctl+0x16b()
    sys_syscall32+0x101()
```

Other Useful mdb Commands

```
!vi tlist  <- run a shell command

ffffffff84920a38::whatis  <- given an address, what is it?
ffffffff84920a38 is ffffffff84920a38+0, allocated from process_cache
ffffffff84920a38::print -t proc_t  <- print a data structure
{
   struct vnode *p_exec = 0xffffffff86102180
   struct as *p_as = 0xffffffff849cd7e8
...
::msgbuf  <- display console output
...
```

. Output of an mdb command can be piped to input of another mdb command.

```
ffffffff84920a38::print -t proc_t p_exec | ::print vnode_t v_path
v_path = 0xffffffff81b405a8 "
/export/home/max/courses/techdays/exercises/lab4/Solution/skelapp"
```

Exercise 5: Debug a Driver

- For this lab, you will install a driver that has a bug that causes the system to panic. You should examine the resulting core file (and then source file) to debug the problem.
- Here are the steps to take:

1. cd /var/crash/*hostname*
2. ls -l   <- should show you a vmcore and unix file, if more than one, use the most recent
3. mdb # <- "#" should be the number of your vmcore/unix files
4. ::status <- see the panic message
5. ::cpuinfo -v <- see what was running
6. $c <- get a stack backtrace
7. Verify the stack trace against the source code
8. Using the program counter, locate the instruction where the panic occurred.
9. Find this instruction in the source code.
10. Identify problem and, if there is time, fix the problem.

http://www.opensolaris.org/