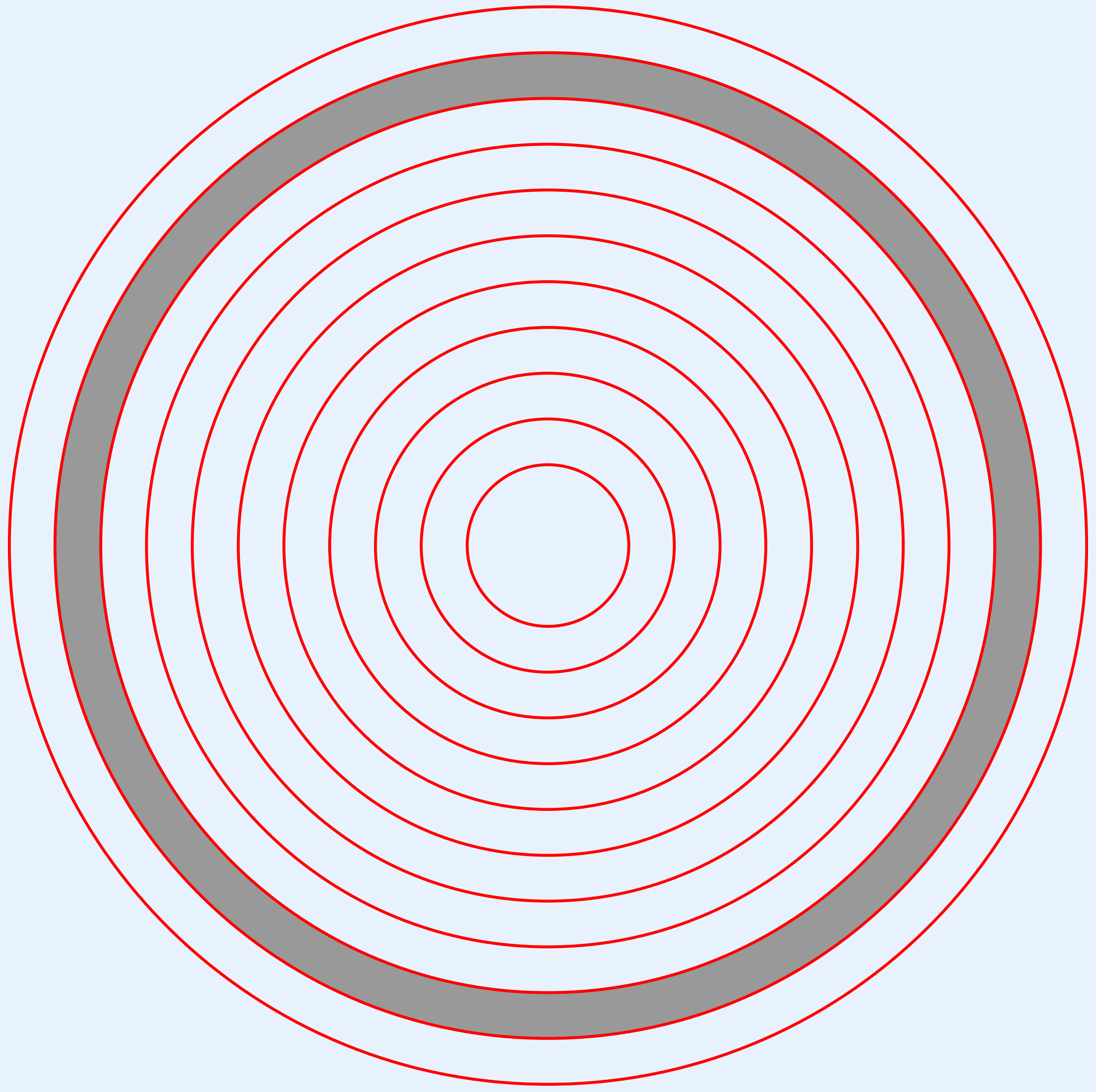
Module XXI

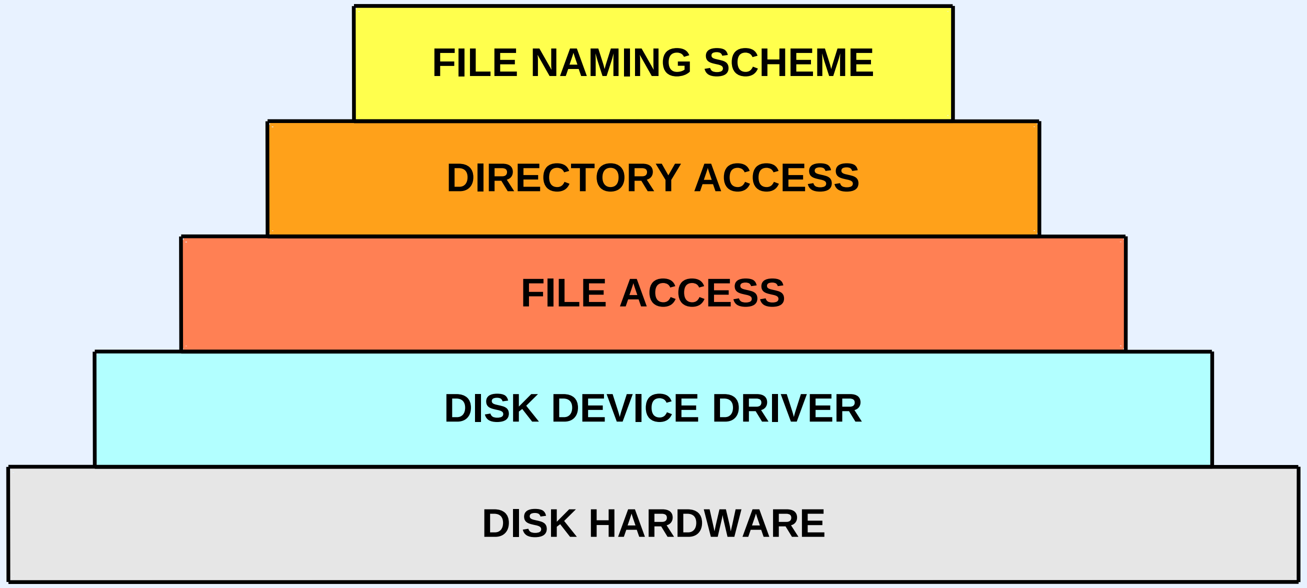
File Names And A Syntactic Namespace

Location Of The Namespace In The Hierarchy



Review

- We said that a file system has three conceptual layers



- We have already considered directory and file access mechanisms
- What about naming?

Identifiers

- Many modules in an operating system use the term *identifier* (*ID*) to designate an object identifier
- Processes use the IDs to identify objects when operating on them
- Examples
  - Each semaphore is given an ID that processes use when they call *wait* and *signal*
  - Each process has an ID that is used when invoking process management functions, such as *suspend*, *resume*, *send*, *ready*, and *kill*
  - Each device has an ID used in device manager functions, such as *read* and *write*

Identifier Choice And Efficient Object Identification

- We have seen that
  - The identifiers used in an operating system consist of integers
  - Choosing values 0, 1, 2, ... for identifiers means that the mapping from an identifier to an actual object is extremely efficient
- To achieve high efficiency, early programming languages required programmers to use numerical identifiers (e.g., FORTRAN specified that when a programmer called *write*, identifier *6* designated a printer)
- However

Although using numerical identifiers makes mapping efficient, humans find such identifiers difficult to understand and remember.

Solving The Problem

- Tension exists between humans and machines
  - Computers work best with numeric values
  - Humans prefer identifiers that convey meaning
- How can we resolve the tension?
- A general approach is used to provide the advantages of each
  - Allow humans to use meaningful symbolic identifiers
  - Perform early binding to convert the symbolic identifiers into an internal numeric form (because early binding increases efficiency)
  - Once the binding has been done, use the numeric form

An Example Of Binding A Name To An Integer Identifier

- Recall that in a Unix file system
  - The file system uses an inode number to identify a file
  - Humans use file names
- The directory system provides a binding from names to files
- A process uses a path name when opening a file:

```
desc = open("path", omode, mode);
```

- The *open* function searches the directory system, and maps the name to an i-node number
- The file system used the i-node number internally

Identifiers, Mappings, And Vulnerability

- Facts
  - Symbolic identifiers work best for humans
  - Operating systems contain mechanisms that map symbolic identifiers into efficient, internal identifiers
- Unfortunately, revealing the mapping to users has a potential security downside: malevolent users (or malware) may be able to guess how internal identifiers are assigned, and then use the information to access other system resources
- An example:
  - The Internet standards specify protocol port numbers used for each application
  - An attacker can use the protocol port numbers in attempts to access services on a computer, even if the owner does not advertise the services

The Principle Of Transparency

- To prevent outsiders from misusing information about the internal representation, operating system designers follow a rule known as the *principle of transparency*:

> **Whenever possible, applications should remain unaware of implementation details such as the location of an object or its representation.**

Protecting Against Attack

- It may seem that because it reveals a mapping between identifiers and values used internally, early binding always introduces a vulnerability
- However, knowing the internal value may not give others access
- As an example, consider file descriptors in Unix
  - A descriptor is only meaningful within one process
  - Even if process 20 learns that process 27 is using descriptor 4 to access file X, process 20 will not be able to access the file because when process 20 references descriptor 4, the reference will be interpreted with respect to process 20’s descriptor table
- Safety: it is safe to use early binding and to reveal how identifiers are mapped to internal values provided that additional protections are employed to prevent the knowledge from being exploited

Transparency And Functionality

- At first glance, the principle of transparency seems both reasonable and innocuous
- However ... true transparency has consequences for functionality
- Consider an operating system that offers access to both a local file system and a remote file system
  - Suppose the functionality of the local and remote file systems differ (a common situation)
  - To keep local and remote file access completely transparent, the operating system must keep the interface identical
  - The effect: transparency means the set of operations that applications can use are limited to the intersection of the operations available on the two file systems
- Another consequence: if the interface is truly transparent, an application will *not* be able to find the actual location of an object, even if doing so is important

File Namespace

- We use the term *file namespace* to refer to the set of all valid file names
- Note that a namespace includes all possible names, not just the names of existing files
- Items in a namespace are bound by both syntactic and semantic restrictions
- Examples
  - Most file systems place a bound on the maximum length of a file name
  - Some file systems prohibit unprintable characters in file names or prohibit “separator” characters (e.g., Unix prohibits the slash character from appearing in file names)

A Namespace For Hierarchical Directories

- Systems such as Multics and Unix provide a hierarchical directory structure in which a directory can contain files and other directories
- A namespace for such a system usually
  - Includes names for directories as well as files
  - Gives uniform names for all files
  - Two forms are used: absolute names and relative names

An Example Of Absolute And Relative Names (Unix)

- In Unix, an *absolute name* begins with a slash, and gives a path downward from the root of the file system
- Examples of absolute names
  - /usr/bin/awk
  - /var/lib/vim/addons
  - /dev/null
- In Unix, a *relative name* gives a path starting from the current directory
- Examples
  - myfile
  - bin/awk
  - ../../lib/java/runtime

Heterogeneous File Names

- A variety of file naming schemes have been created
  - MS-DOS *Device: file*
  - V-System *[context] name*
  - BSD Unix *machine: path*
- Unfortunately
  - The Internet means that when referring to a file on a remote computer, the form of the file name may differ completely from the form of the file name used on the user's local computer
  - No single naming scheme is best, which means that it is unlikely a single scheme will ever be adopted by all systems

Gluing Together Many File Systems

- Can an operating system hide differences in names and provide users with a single, uniform view?
- One approach consists of building a single, large file system out of multiple pieces by inserting an extra level on top of the file system software hierarchy
- The extra level is arranged to
  - Present a uniform interface to users and application programmers
  - Hide the details of the underlying file systems
  - Map unified file system names to names for specific underlying file systems
  - Map generic file operations to appropriate operations on the underlying file systems

An Example

- Suppose a computer has two disks and two separate Unix file systems, *F1* and *F2*
- To unite them into a single giant file system
  - Create a new root directory that is “above” the two file systems
  - Add two entries to the new root, one for *F1* and one for *F2*
- In the new system, a name of the form */F1/path_1* will refer to a file in file system *F1*, and a name of the form */F2/path_2* will refer to a file in file system *F2*
- Examples
  - The name */F1/var/mail/smith* will be interpreted as a reference to */var/mail/smith* in file system *F1*
  - The name */F2/usr/bin/awk* will be interpreted as a reference to */usr/bin/awk* in file system *F2*

Unix File System Mounting

- Unix provides a unification mechanism similar to the one described above
- The idea is straightforward: make a file system appear to be one of the directories in the main file system
- The procedure is
  - Start with one file system as the root
  - Create an empty directory at any point in the directory, call it *X* hierarchy
  - Use the *mount* command to specify that another file system should attach in place of directory *X*
- Once the new file system is mounted, its root directory appears in place of directory *X*
- Note: mounting only affects the cached copy of an i-node in memory; the two file systems remain independent on disk

Using Names

Compound Names And Their Parts

- Consider the Unix file name

  /var/lib/vim/addons

- Because we know the meaning of items in a Unix file name, we tend to think of the name as three directories (*var*, *lib*, and *vim*) plus one file (*addons*)
- Syntactically, we think of the name as the four items separated by slash characters

Hierarchies, Strings, And Prefixes

- Instead of thinking of a file name as specifying a sequence of directories, think of it merely as a string of characters
- For example, think of */var/lib/vim/addons* as a string of nineteen characters
- Observe that
  - The one-character prefix / specifies what we think of as the top level directory
  - The four-character prefix */var* specifies what we think of as a second-level directory
  - The eight-character prefix */var/lib* specifies what we think of as a third-level directory, and so on
- The point is that some prefixes of the string specify directories in the directory hierarchy

The Prefix Property

- We use the term *prefix property* to describe the relationship between prefixes and the directory hierarchy
- Longer prefixes refer to items further down the directory hierarchy
- Of course, an arbitrary length prefix may not correspond to a directory
- In the example string */var/lib/vim/addons*
  - The two-character prefix */v* does not name a directory
  - The seven-character prefix */var/li* does not name a directory
  - The seventeen-character prefix */var/lib/vim/addo* does not name a directory
- Conclusion: a prefix length must be chosen carefully or the prefix will not correspond to a directory or a file

Using A Prefix To Identify A File System

- As an example of how we can use the prefix property to build a namespace, imagine a computer with two file systems, a local file system that uses names of the form *C:file* and a remote file system that supports Unix names of the form *file* or *relative_path*
- Now imagine creating a file namespace that includes both file systems
- We could choose the prefix */local/* for the local file system and */remote/* for the remote file system
- If a user specifies the name */remote/a/b/c*, the system will remove the prefix, and use the name *a/b/c* as a path name on the remote file system
- The local file system is slightly more complex because the prefix must be replaced instead of removed: if a user specifies the name */local/X*, the system must replace the prefix */local/* with the string *C:*, and then use the name *C:X* as the name of a file on the local file system

Generalized Prefix Mapping

- Instead of building code that uses *if* statements to check for */local/* and */remote/*, consider a generalized system
- The generalized system will use a table of prefixes
- The *prefix table* will contain one entry for each possible prefix
- Each entry in the table will be a 3-tuple that specifies
  - A prefix to be matched
  - A replacement string (e.g., the prefix “/local/” might be replaced by “C:”)
  - A file system to be used once the file name has been modified

Using A Prefix Table

- When a user specifies a file name, the prefix mapping code searches the prefix table
- When it finds a prefix in the table that matches the prefix of the name the user specified, the code
  - Modifies the user’s file name by substituting the replacement string in the entry in place of the prefix
  - Uses the modified file name as the file name for the file system specified in the entry
- A prefix table has several advantages over using conditional statements in the code
  - Nothing is hard-wired
  - The mappings can be changed at any time, even while the operating system continues to run

A Prefix Table For Our Example

- Consider the local and remote file systems described on previous slides
- Assume the local file system is *local_fs* and the remote file system is *remote_fs*
- A prefix table that captures the needed mappings contains two entries

Prefix	Replacement	File System
"/local /"	"C:"	local_fs
"/remote /"	" "	remote_fs

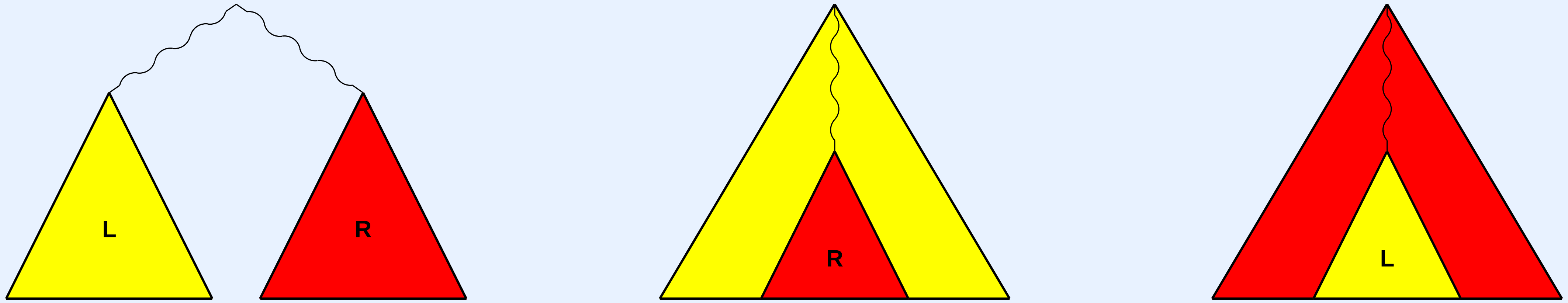
- Because the entry for the remote file system has a replacement set to the null string, the prefix */remote/* will be removed from the name, and no further modifications will occur

Names, Prefixes, And Subdirectories

- Treating each name as a string has several implications
- No meaning: if a name is merely a string, no special meaning is assigned to any character
  - Therefore, “x/y/z” is merely a string of five characters
  - Humans might think of *y* as a subdirectory of *x*, but code that operates on strings does not
- Consequence: treating names as character strings means
  - The segments in a name may not match the levels in the directory hierarchy
  - Applications cannot infer semantic meaning

Possible Hierarchies

- Using prefixes allows us to create a unified abstract namespace that includes multiple file systems
- We have seen, for example, that an abstract namespace can include both local and remote file systems
- Interestingly, various arrangements of a hierarchy are possible: local and remote can be located at the same level, remote can be a subdirectory of the local file system, or local can be a subdirectory of the remote file system, as illustrated



A Xinu Namespace

The Xinu Syntactic Namespace

- Xinu uses a single namespace to unify multiple file systems into a single abstract naming scheme
- The characteristics are
  - Syntactic: the Xinu namespace uses prefix mapping
  - Optional: applications can choose to bypass the namespace and access a specific file system directly
  - Dynamic: the namespace mappings can be changed at run time

Xinu Namespace Details

- In Xinu, everything is a device; so the namespace is a device (actually a pseudo-device because there is no real hardware)
- The namespace pseudo-device is configured to the name *NAMESPACE*
- An application
  – Calls *open* on the *NAMESPACE* pseudo-device
  – Supplies a file name
- The namespace *open* function
  – Uses a prefix table to find an entry in the prefix table where the prefix matches the file name
  – Modifies the file name by replacing the prefix with the specified replacement string
  – Calls *open* on the file system device in the entry

An Example Of Passing An Open Call To A File System

- Consider a Xinu namespace that has been configured to have an entry in the prefix table with the following values

  ("/remote/", " ", RFILESYS)

- Suppose a process calls

  open(*NAMESPACE*, "/remote/xyz", "r");

- The namespace driver will
  - Map “/remote/xyz” into “xyz”
  - Call *open* on the RFILESYS device with the mode argument the user specified

  open(RFILESYS, "xyz", "r");

Summary Of The Xinu Syntactic Namespace Operation

- The Xinu namespace uses a table to hold a set of 3-tuples

  (*prefix, replacement, device*)

  where *prefix* and *replacement* are strings of characters

- Given a file name, the namespace
  - Checks the name against each entry in the table
  - If a prefix in the table matches the beginning of the name
    - Rewrite the file name by substituting the replacement string for the prefix
    - *Open* the resulting file name on the device specified in the table entry, and return the result to the caller

- If none of the prefixes in the table match the name, return SYSERR to caller

A Default Prefix Mapping

- We said that if no prefix in the table matches a name, *open* returns *SYSERR*
- To prevent errors, one can install a *default* entry in the table (i.e., an entry that will be used if none of the other entries match)
- Prefix matching means a default entry can be added without modifying the lookup code and without any special cases
- To insert a default entry, specify a prefix of the null string
  - The null string is considered to be a prefix of all other strings
  - Therefore, an entry in which the prefix is null will always match
- We will see examples of how a default entry can be used

Adding Entries To The Prefix Table

- Xinu uses the same approach as Unix, a *mount* function
- In Xinu, *mount* merely adds an entry to the prefix table
- As expected, Xinu's *mount* function takes three arguments
  - A string that specifies a prefix
  - A string that specifies a replacement
  - A device descriptor for a file system pseudo-device
- Example:

```
mount ( "/remote /", "", RFILESYS );
```

Namespace Initialization

- The driver for the *NAMESPACE* pseudo-device includes initialization function *naminit*
- After Xinu boots, it calls the *init* function for each device, which means it calls *naminit* for the *NAMESPACE* device
- *Naminit*
  – Fills in initial values in the prefix table
  – Automatically creates an entry for each device
  – Uses a name of the form */dev/x* for a device named X
- Example
  – *Naminit* creates an entry */dev/console* for the *CONSOLE* device
  – An *open* on */dev/console* will be mapped to an *open* on *CONSOLE*
- To create additional entries, calls to *mount* can be added to *naminit*

An Excerpt From Naminit

- An example of *mount* calls used to build a namespace (from *naminit*)

```
mount("/dev/null",      "",          NULLDEV);
mount("/remote/",       NULLSTR,     RFILESYS);
mount("/local/",        NULLSTR,     LFILESYS);
mount("/tmp/",          "tmp-",      LFILESYS);
mount("/dev/",          NULLSTR,     SYSERR);
mount("~/",             NULLSTR,     RFILESYS);
mount("/",              "root:",     RFILESYS);
mount("",               NULLSTR,     LFILESYS);
```

- Observe
  - The last entry uses a null prefix to provide a default (the local file system)
  - File names that start with ~/ are also mapped to the remote file system
  - Names that begin with a slash are mapped to the remote file system, and the prefix root: is added to the name

Declarations For The Namespace

```
/* name.h */

/* Constants that define the namespace mapping table sizes */

#define NM_PRELEN       64              /* Max size of a prefix string  */
#define NM_REPLLEN      96              /* Maximum size of a replacement*/
#define NM_MAXLEN       256             /* Maximum size of a file name  */
#define NNAMES          128             /* Number of prefix definitions */

/* Definition of the name prefix table that defines all name mappings */

struct  nmentry {                       /* Definition of prefix table   */
        char    nprefix[NM_PRELEN];     /* Null-terminated prefix       */
        char    nreplace[NM_REPLLEN];   /* Null-terminated replacement  */
        did32   ndevice;                /* Device descriptor for prefix */
};

extern  struct  nmentry nametab[];      /* Table of name mappings       */
extern  int32   nnames;                 /* Number of entries allocated  */
```

Open Function For The Namespace

```
/* namopen.c - namopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  namopen  -  Open a file or device based on the name
 *------------------------------------------------------------------------
 */
devcall namopen(
          struct dentry *devptr,        /* Entry in device switch table */
          char   *name,                 /* Name to open                 */
          char   *mode                  /* Mode argument                */
        )
{
        char    newname[NM_MAXLEN];     /* Name with prefix replaced    */
        did32   newdev;                 /* Device ID after mapping      */

        /* Use namespace to map name to a new name and new descriptor */

        newdev = nammap(name, newname, devptr->dvnum);

        if (newdev == SYSERR) {
                return SYSERR;
        }

        /* Open underlying device and return status */

        return  open(newdev, newname, mode);
}
```

Mapping Function For The Namespace (Part 1)

```
/* nammap.c - nammap, namrepl, namcpy */

#include <xinu.h>

status  namcpy(char *, char *, int32);
did32   namrepl(char *, char[]);

/*------------------------------------------------------------------------
 *  nammap  -  Using namespace, map name to new name and new device
 *------------------------------------------------------------------------
 */
devcall nammap(
         char   *name,                  /* The name to map              */
         char   newname[NM_MAXLEN],     /* Buffer for mapped name       */
         did32 namdev                   /* ID of the namespace device   */
        )
{
        did32   newdev;                 /* Device descriptor to return  */
        char    tmpname[NM_MAXLEN];     /* Temporary buffer for name    */
        int32   iter;                   /* Number of iterations         */

        /* Place original name in temporary buffer and null terminate */

        if (namcpy(tmpname, name, NM_MAXLEN) == SYSERR) {
                return SYSERR;
        }
```

Mapping Function For The Namespace (Part 2)

```
        /* Repeatedly substitute the name prefix until a non-namespace  */
        /*   device is reached or an iteration limit is exceeded        */

        for (iter=0; iter<nnames ; iter++) {
                newdev = namrepl(tmpname, newname);
                if (newdev != namdev) {
                        return newdev;  /* Either valid ID or SYSERR    */
                }
                namcpy(tmpname, newname, NM_MAXLEN);
        }
        return SYSERR;
}
```

Mapping Function For The Namespace (Part 3)

```
/*------------------------------------------------------------------------
 *  namrepl  -  Use the name table to perform prefix substitution
 *------------------------------------------------------------------------
 */
did32   namrepl(
          char    *name,                  /* Original name                */
          char    newname[NM_MAXLEN]      /* Buffer for mapped name       */
        )
{
        int32   i;                        /* Iterate through name table   */
        char    *pptr;                    /* Walks through a prefix       */
        char    *rptr;                    /* Walks through a replacement  */
        char    *optr;                    /* Walks through original name  */
        char    *nptr;                    /* Walks through new name       */
        char    olen;                     /* Length of original name      */
                                          /*   including the NULL byte    */
        int32   plen;                     /* Length of a prefix string    */
                                          /*   *not* including NULL byte  */
        int32   rlen;                     /* Length of replacment string  */
        int32   remain;                   /* Bytes in name beyond prefix  */
        struct  nmentry *namptr;          /* Pointer to a table entry     */
```

Mapping Function For The Namespace (Part 4)

```
/* Search name table for first prefix that matches */

for (i=0; i<nnames; i++) {
        namptr = &nametab[i];
        optr = name;            /* Start at beginning of name   */
        pptr = namptr->nprefix; /* Start at beginning of prefix */

        /* Compare prefix to string and count prefix size */

        for (plen=0; *pptr != NULLCH ; plen++) {
                if (*pptr != *optr) {
                        break;
                }
                pptr++;
                optr++;
        }
        if (*pptr != NULLCH) {  /* Prefix does not match */
                continue;
        }

        /* Found a match - check that replacement string plus   */
        /*  bytes remaining at the end of the original name will*/
        /*  fit into new buffer.  Ignore null on replacement    */
        /*  string, but keep null on remainder of name.         */
```

Mapping Function For The Namespace (Part 5)

```
                olen = namlen(name ,NM_MAXLEN);
                rlen = namlen(namptr->nreplace,NM_MAXLEN) - 1;
                remain = olen - plen;
                if ( (rlen + remain) > NM_MAXLEN) {
                        return (did32)SYSERR;
                }

                /* Place replacement string followed by remainder of    */
                /*   original name (and null) into the new name buffer   */


                nptr = newname;
                rptr = namptr->nreplace;
                for (; rlen>0 ; rlen--) {
                        *nptr++ = *rptr++;
                }
                for (; remain>0 ; remain--) {
                        *nptr++ = *optr++;
                }
                return namptr->ndevice;
        }
        return (did32)SYSERR;
}
```

Mapping Function For The Namespace (Part 6)

```
/*------------------------------------------------------------------------
 *  namcpy  -  Copy a name from one buffer to another, checking length
 *------------------------------------------------------------------------
 */
status  namcpy(
         char           *newname,       /* Buffer to hold copy          */
         char           *oldname,       /* Buffer containing name       */
         int32          buflen          /* Size of buffer for copy      */
        )
{
        char    *nptr;                  /* Point to new name            */
        char    *optr;                  /* Point to old name            */
        int32   cnt;                    /* Count of characters copied   */

        nptr = newname;
        optr = oldname;

        for (cnt=0; cnt<buflen; cnt++) {
                if ( (*nptr++ = *optr++) == NULLCH) {
                        return OK;
                }
        }
        return SYSERR;          /* Buffer filled before copy completed  */
}
```

Problems With A Syntactic Approach

- Infinite name expansion
- Infinite recursion
- A short prefix hides a longer one

Problem #1: Infinite Name Expansion

- Suppose the namespace contains

```
mount("a", "this_a", NAMESPACE);
mount("t", "and_t", NAMESPACE);
```

- Consider

```
open(NAMESPACE, "and_that", "r");
```

- Repeated replacement keeps building a longer and longer name

```
this_and_that
and_this_and_that
this_and_this_and_that
and_this_and_this_and_that
     {and so on...}
```

- Solution: check the size of the expanded name, stop and return (SYSERR) if the size exceeds the maximum name length

Problem #2: Infinite Recursion

- Suppose the namespace contains

```
mount("cs_", "ece_", NAMESPACE);
mount("ece_", "cs_", NAMESPACE);
```

- Consider

```
open(NAMESPACE, "cs_is_best", "r");
```

- Repeated substitution goes on forever, alternating between *cs_is_best* and *ece_is_best*
- Solution: limit the number of repeated substitutions to *nnames*

Problem #3: A Short Prefix Hides A Longer One

- Suppose the namespace contains two entries in the following order

```
mount("l", "l", RFILESYS);
mount("local/", "", LFILESYS);
```

- Consider

```
open(NAMESPACE, "local/x", "r");
```

- The first entry prevents the second from ever getting used. so the open always goes to *RFILESYS* even though the second entry appears to direct it to *LFILESYS*

- Solution: order entries in the table with longest prefix first (and prohibit duplicate prefixes)

Generalized Patterns

- Although our example uses a fixed string as a pattern, more sophisticated patterns are possible, for example:

Character	Meaning
^	match beginning of string
$	match end of string
.	match any single character
*	repeat 0 or more of a pattern
\	take next character in pattern literally
other	self match as in a fixed string

Summary

- It is possible to build a naming hierarchy separate from the underlying file systems
- When the naming hierarchy is viewed syntactically, prefixes define each piece of the hierarchy
- A prefix table that includes replacement can be used to create a fairly general hierarchy
- In Xinu, the syntactic namespace is implemented as the *NAMESPACE* pseudo-device
- Opening the *NAMESPACE* device causes a file name to be mapped according to the prefix table and then passed to *open* on a specific file system
- Using the null string as a prefix creates a default entry in the prefix table that is guaranteed to match any file name

Questions?