Illinois Institute of Technology **Homework #6**
CS 450 - Operating Systems **October 21, 2002**

Homework # 7 – DUE: 11:59pm November 15, 2002
NO EXTENSIONS WILL BE GIVEN

1. **Overview**

   In this assignment you will implement that `FILES` module of *OSP*. This is an in depth assignment and probably more complex than the last one. You will need to start preparing immediately.

2. **Getting Set Up**

   Now that you've gotten a little more used to how *OSP* works, you should have a decent idea how to start. Create a directory off your home directory called `as6`. Into this directory copy the following files for the assignment:

   ```
   ~osp/asg6.sparc64/Makefile
   ~osp/asg6.sparc64/files.c
   ~osp/asg6.sparc64/dialog.c
   ```

3. **What To Do**

   You will implement the missing functions at the end of the file `files.c`. These functions are: `files_init`, which should perform any required initializations to your data structures, `openf`, which is called by a process requesting access to a file, `closef`, which is called by a process when it is done with the file, `readf` to get data from an open file, `writef` to put data into a file, and `notify_files`, which is called by routines in the `DEVICES` module to let the `FILES` module know that an `I/O` request has been completed.

   You will also have to implement internal routines for file creation and deletion. These routines are not part of the interface, but are necessary for the implementation. Read section 1.8 in the *OSP* manual carefully, and follow the guidelines below.

4. **Grading**

   For your program, credit will be given for *style*, *correctness*, and *completeness*. Elements important to good style include:

   - discussion - see below,
   - documentation - especially function headers,
   - modularity - a function should not (usually) extend beyond a single page, and
   - readability - the various functions should be visually separated.

   At the end of your program, you should create two sections, as per the last assignment. One called "Implementation and Design Analysis" where you explain the design choices made in the assignment and another called "Statistic Explanation" where you comment on how the statistics differ for `par.low` and `par.high`.

   The correctness of your implementation reflects the ability of your program to run to completion without errors on both parameter files. There is no "right" solution, however solutions with extremely poor performance relative to that of the `OSP.demo` program will be penalized accordingly.

The completeness of your implementation is based on whether your solution implements merely a simple array, or a more advanced method for finding files, such as a hash or tree (see below).

To cover the question I know will be asked, this assignment will be worth an amount of points N where N will be determined at a future point in time and will be same value of N that was use for the previous assignment; it is dependent on how many projects will be assigned.

A solution that does not conform to these guidelines will not receive full credit.

5. **Hashing and Search Trees**

   To get full points for completeness you must use some load balancing scheme for the devices as mentioned in the last paragraph of page 39 in the *OSP* manual. You should review the discussion of `dev_entry_node` on page 33.

   The *OSP* manual suggests, on page 38, that the *file directory* be organized as either a *search tree* or a *hash table*. For this assignment you will do the following. Both of these will result in higher performance that a standard array. What you implement is up to you.

   Hashing You may choose any hashing strategy you like with the following restriction. You **must** implement a strategy that goes beyond simple hashing. You may choose to implement a collision resolution strategy, an extensible hashing strategy, or tree structured "buckets." You should explain clearly your choice and the strategy you used to implement it.

   Balanced Trees Any strategy that *guarantees* a $log(n)$ look-up time will suffice, but keep in mind that a simple tree implementation can become skewed, so some means of height balancing must be used.

6. **A Few Random Notes**

   - I expect you may have some trouble with the above data structures, so I recommend that you get your implementation working with a simple directory structure first, and work on the directory data structure *after* you have debugged the rest of you code.
   - Any good data structures book will discuss both hashing and balanced trees, and I recommend that you review that material before you start coding. The CS430 textbook (by Cormen, Lieserson and Rivest) is a wonderful resource for these.
   - You may be uncertain about the appropriateness of a particular choice of data structure. Is it enough? Is it too much? If you are unsure check with the TA before you begin coding.
   - Explain clearly the choices you've made in your implementation. Do not assume that the TA can read your mind (or your code).
   - I am working on a version of this for Windows. It may or may not be released in time for it to be of use to you. If it does come out, your software MUST still work under Unix. The Windows software is intended for remote students who are severely hindered by high ping times to the CS450 machine.
   - Hopefully you all learned a valuable lesson about not waiting until a day or two before the assignment is due to complete it. It may not be possible to do so many extensions next time, and the TA's may not be available.
   - Remember what was said about reading the manual? Many students never did. If you read the manual it explains very well everything that needs to be done for the project.

7. **Handing In Your Assignment**

   There are two simulation runs that are used with this set of programs. When you submit your program using the `hand_in` script, you should use these files (in this order):

   ```
   ~osp/asg6.sparc64/par.high
   ~osp/asg6.sparc64/par.low
   ```

   The file `par.low` is a simulation with a low frequency of I/O events and `par.high` is a file with a high frequency of I/O events. If you wish to compare your results with that of the standard file system see:

   ```
   ~osp/asg6.sparc64/run.low
   ~osp/asg6.sparc64/run.high
   ```

   Make sure that your *name* and *login name* are included in your source file. Also include the descriptions mentioned previously as a block of comments. Use the `hand_in` procedure to post your work to the submissions directory by 11:59:59pm of the due date. Make sure to save copies of your program for your own reference. If we need to get more information from you, we will contact you.

NOTICE:

Unauthorized collaboration or assistance is prohibited. If you have a question, ask the teaching assistants. Do not copy code from a neighbor. Students who participate in copying will receive an 'E' grade for the course. The same penalty applies for students who fail to adequately protect their accounts.

Appendix

Data Structures

I've included this section as helpful information to get you started on the project. The first thing is that you need to get familiar with the following structures:

```
typedef struct file_dir_entry_node FILE_DIR_ENTRY;
struct file_dir_node {
    char *filename;
    INODE *inode;
    int *hook;
};
```

This is a simple element that correlates filenames to their appropriate inode. This is the structure that after you get it working as a simple array you will want to optimize into some sort of tree or hash to minimize lookup time from `filename` to `inode`. Note, because of the typedef, you can refer to this as just `FILE_DIR_ENTRY` (ie `FILE_DIR_ENTRY *myFile;`).

```
typedef struct inode_node INODE;
struct inode_node {
    int inode_id;    /* set by programmer (optional)   */
```

```
    int dev_id;      /* device id; index into Dev_Tbl  */
    int filesize;
    int count;       /* # of files assoc'd with i-node */
    int allocated_blocks[MAX_BLOCK];
                     /* info on where file is stored   */
    int *hook;
};
```

It should be clear here that you don't need to have inode_id's for inodes. It's completely at your discretion. The filesystem is recreated each and every time, so the inodes for files will change from run to run. You might also want to refer to Dev_Tbl, which is on page 34 of the *OSP* manual, but I've included it here.

```
typedef struct dev_entry_node DEV_ENTRY;
struct dev_entry_node {
    int dev_id;     /* device id - index into Dev_Tbl; */
                    /* set by the simulator            */
    BOOL busy;      /* the busy flag; true if busy     */
    BOOL free_blocks[MAX_BLOCK];
                    /* block i is free if and only if  */
                    /* free_blocks[i] == true          */
    IORB *iorb;     /* iorb serviced by this device    */
    int *dev_queue  /* optional ptr to device queue    */
    int hook;
};
DEV_ENTRY Dev_Tbl[MAX_DEV];
```

I'm not going to real far into depth here. But you can imagine that you'll have to check the free_block to see if you can write to a block. Also you need to set all the blocks to free when you start it up.

```
typedef struct ofile_node OFILE;
                      /* entry in the table of open files */
struct ofile_node {
    int ofile_id;     /* used by the trace facility;      */
                      /* set by the simulator             */
    int dev_id;       /* device where file resides        */
    int iorb_count;   /* # of this file's pending iorb's  */
    INDOE *inode;     /* pointer to this files i-node     */
    int *hook;
};
```

This will form your table of open files. It contains information on what device the file is on and also the INODE that it resides in.

Procedure Descriptions

This section will give brief overviews of what each function should do. They are not the only way to do this assignment. I would suggest trying your own way, then trying to figure out these ways.

openf

`openf()` takes two parameters, *filename* which is your standard C character string, and a pointer to an `OFILE` that is allocated by the simulator. `openf()` must fill in all the proper values for this. First it searches the directory for *filename*, if its not there, you have to create it. Because no routines outside of the `files.c` call your create file routine, it can be private.

The `OFILE` template is initialized by setting the *inode* field to the location where the file exists and then setting the *iorb_count* to 0. You will need to copy the device id from inode to to the `OFILE`, this is a bit of redundancy, but pays off for performance. After this the *count* field of the inode is incremented. There cannot be a case where a file cannot be opened because it operates a little like Unix where if a file doesn't exist, it is created.

closef

`closef()` simply closes a file and removes its entry from the list. See page 40 for more details. It is important that the failure conditions are checked.

files_init

`files_init()` creates all of the data structures you need. You should iterate over all of the devices (from 0 to MAX_DEV-1) and set their free blocks to `MAX_BLOCK`. For each block on that device you need to set it free. You can set an individual block *j* on device *i* via `Dev_Tbl[i].free_blocks[j] = free;`. Then you need to iterate over `free_files[]` from 0 to MAX_OPENFILE-1 and set each entry to true. Finally iterate over each `free_inode[]` from 0 to MAX_OPENFILE-1 and set each entry to true.

You may choose to use different structures than `free_inode[]` and `free_files[]`. That is your choice however you do it. In any case you need an easy way to tell if a slot is free or not. Arrays are good for that sort of thing because access is always indexed.

readf and writef

These functions start out pretty similar. An `OFILE` template is passed in. This information must be filled in. The first thing that you need to do is to locate the `OFILE` that matches the start of the template that is passed in. I have a little helper routine called `locate_ofile` that iterates through my list of open files and returns `open_files_tbl[i]` if it equals the `OFILE` that is passed in. This function will not work right out of the box, but it should give you an idea how to index stuff.

```
PRIVATE
OFILE *locate_ofile(file)
  OFILE *file;
{
    int i;

    i = 0;
    while((i < MAX_OPENFILE) && (open_files_tbl[i] != file))
        i++;

    if (i == MAX_OPENFILE)
        return(NULL);
    return open_files_tbl[i];
}
```

If for some reason we get a `NULL` when that function returns, we've got something weird going on and need to call `osp_abort()` so we can stop it.

Next for reads we need to see if the `position` is less than 0 or if it's greater than the end of the file. On a write we need to check if it is less than 0. This is because we can't read where there is no data and we can't write to a negative offset. If this is the case, we set `iorb->dev_id = -1;` and return with `fail` as the argument.

Now I'll explain the rest of read and write separately with read first.

Next you check to see if the physical block of the file is empty or not (via something like `find_block_read(position,file->inode)`) and if it is, we need to complain also because we are trying to read a file that doesn't exist.

If we've made it this far, we can now copy the stuff over from iorb and set it up to forward to the interrupt driver. Here is a snippet that may help here:

```
iorb->dev_id = file->inode->dev_id;
iorb->block_id = phys_block;
iorb->pcb = PTBR->pcb;
iorb->action = read;
iorb->page_id = page_id;
iorb->file = file;
iorb->hook = NULL;

/* set event happened to false */
iorb->event->happened = false;
/* increment the pending I/O */
file->iorb_count += 1;

/* set the interrupt vector to indicate a read interrupt */
Int_Vector.event = iorb->event;
Int_Vector.iorb = iorb;
Int_Vector.cause = iosvc;
gen_int_handler();
```

Now for the `writef()` function. After you make sure it's not a negative offset, you need to figure out if we need a new block for the data. I do this in a little bit of a round about way. You can see some of the code here:

```
fb = file->inode->filesize == 0 ? -1 :
     (file->inode->filesize-1)/PAGE_SIZE;
rb = (position)/PAGE_SIZE;

if (rb > fb) {
    new_block_num = rb - fb;
    if (allocate_block(new_block_num,file->inode)==EMPTY) {
        iorb->dev_id = -1;
        return(fail);
}
```

If we've made it this far, you need to see if the position is beyond the end of the file, if it is, increment the file size. Then you need to assemble the iorb. Here is the code that might help you do that:

```
iorb->dev_id = file->inode->dev_id;
```

```
    iorb->block_id = file->inode->allocated_blocks(rb);
    iorb->pcb = PTBR->pcb;
    iorb->action = write;
    iorb->page_id = page_id;
    iorb->file = file;
    iorb->hook - NULL;

    /* set event happened to false */
    iorb->event->happened = false;

    /* increment number of pending I/O */
    file->iorb_count += 1;

    /* set interrupt vector to indicate a write */
    Int_Vector.event = iorb->event;
    Int_Vector.iorb = iorb;
    Int_Vector.cause = iosvc;
    gen_int_handler();
```

notify_files

This is probably one of the shortest functions you'll write here. It's pretty straight forward. An `IORB` is passed in. You set get the `OFILE` from that and locate the full record of it using the `locate_ofile` routine previously mentioned. If there is a record, then decrease `file->iorb_count` by one. Otherwise if there isn't a record, we've got something wrong in our system and need to abort immediately.