

Program 2: Shared Memory Media Server (mserv2)

CS 4210 Advanced Operating Systems * Fall 99 Hutto

Groups of 3!

DUE:

design review: (in class) Tuesday 19 October 99

final turn: 3pm (classtime) Thursday 4 November 99

Questions? Office hours, email, newsgroup...

For your **second programming assignment** of the semester, you will **implement a media server with multiple clients and suppliers, each implemented as a separate process**. The server and customer processes are multithreaded (using Pthreads) and processes interact using System V shared memory and cross-process synchronization variables. In this assignment, the server will actually manage a directory of media files (AUDIO, GRAPHICS and VIDEO) and supply these to customers on request. Customer processes launch individual (existing) media players (audioplay, xv, mpeg_play) to consume the media. Suppliers add items to the server's media store as the program executes.

You may work in teams of 3 for this project. You have four weeks to complete this non-trivial assignment. There will be a detailed design check on Tuesday the 19th during class. Each team will handin a detailed design document (about 5 pages) and do a brief (10 to 15 minute) class presentation.

We will not go over the System V shared memory primitives in detail in class so you need to read the fine documentation handed out in class!

Server

The server process manages a directory of media files and maintains an in memory "database" of these files. Media files are provided by suppliers and requested (checked out) by customers. Media file bits are moved around in shared memory segments. Suppliers and customers enqueue requests in a bounded buffer (the request queue) that is implemented as a well-known shared memory segment mapped by all the processes in the system. The synchronization variables used to implement the bounded buffer also reside in the same shared memory segment. Replies are placed in a per-process shared memory "reply" segment. Because of a severe limitation on the number of segments that any one process can have "attached" at any given time (6 under Solaris), these reply segments must be attached and detached dynamically for each server transaction.

The server is multithreaded and maintains a pool of worker threads (nworkers specified in the init file) as in the first assignment. Note that the server will be "throttled" by the number of shared segments that can be attached (6). Since all processes must attach (map) the request queue, only 5 requests can be handled simultaneously.

Each process in the system will log transactions to a per-process log file in the current directory. All log files will have type ".log". Log file names will begin with "server", "customer" and "supplier" as appropriate. Client log files (customers and suppliers) will be distinguished by adding the client id to the filename. For example: customer123.log.

The server will also be the “main” process for your system and will spawn the number of suppliers and customers specified in the init file (nsuppliers, ncustomers). When the server receives a SHUTDOWN request, it will signal the clients to terminate, and print the final database.

Requests

There are basically only 6 requests that the server knows about. Suppliers and customers are distinguished by unique integer ids specified in a configuration file. All requests include the client id. Requests from unknown clients are rejected.

```
Customers:
    *LIST( id, type )
    *START( id, title )
    STOP( id, title )

Suppliers:
    ADD( id, title, type, ncopies )
    *SUPPLY( id, title )

Anyone:
    SHUTDOWN
```

Customers can get a LIST of managed titles of the specified type (ANY, AUDIO, GRAPHICS, VIDEO) and START and STOP a transfer of a particular title. Suppliers can attempt to ADD a specified title to the database. If the ADD is successful, the media bits are provided in a separate transaction. Finally, for convenience, the server responds to a SHUTDOWN request and gracefully terminates.

Request Details

There are some important details about the request types. System V shared memory segments also have a length restriction (about 1 meg on our systems) so we can't assume that replies will always “fit” into a single reply segment. Therefore, requests that move bits around may involve several “exchanges” between the server and client. You can think of this as a simplified “bound buffer” problem where the buffer has only one “slot”. The client and server pass control of the shared reply segment back and forth, exchanging “chunks” of data until the entire transaction is complete. The requests that may require exchanges are marked with an asterisk above. You will need a mechanism to signal the “end” of the reply sequence (perhaps include a “bytes remaining” field in the reply segment).

The START command is asynchronous. The customer process will create a separate thread for each transfer. The STOP command allows the customer to terminate a transfer (thread) that is in progress and serves as a “return” command. Titles are maintained as strings and must be distinct. There are two LIMITS that the system must respect. First there is a maximum number of media files that the server can maintain (SMAX). This value will be specified in a configuration file (mserv2.init). Attempts by suppliers to add titles after this number has been reached will fail. Similarly, there is a maximum number of titles that a customer can have “checked out” at any given time (CMAX). Finally, there is a number of copies associated with each media item.

Note that the request descriptions above show the “logical parameters”. Your actual implementation of each request will need to supply additional implementation specific details such as an identifier of the shared reply segment and the reply segment size (“chunk size”).

Errors

There are several possible errors and corresponding error codes and you may need to introduce additional codes:

- 1 ENoSuchTitle
- 2 ENotCheckedOut
- 3 EOverLimit
- 4 ENotAvailable
- 5 EUnknownClientID
- 6 EUnknownType
- 7 EInvalidNCopies
- 8 ENoMoreSpace

Customers

Each customer process will receive its id as a command line argument. Recall that customer processes are spawned by the main server process after it reads `mserv2.init`. Customer processes will be multithreaded. Clients will NOT use randomly generated delays as in the first project. Rather, clients will be driven by commands in the file `mserv2.cmds`. Each customer will have a main thread that is responsible for reading `mserv.cmds` and “executing” commands for that customer (distinguished by customer id). Possible commands include: LIST, START, STOP, SHUTDOWN and PAUSE. The main thread will execute LIST, STOP, SHUTDOWN and PAUSE commands directly. START commands will spawn a thread to handle the media transfer and “play” the resulting media. Recall that there is a per-process reply segment so concurrent threads will contend for access to this segment. **(NOTE: This will be a bit tricky... Need to talk about this some more. – pwh)** Here is a sample format for the `mserv.cmds` file:

```
c123 list audio
c123 list any
c123 pause 12
c123 start graphics "zebra.jpg"
c123 start audio "zoom.au"
c123 pause 10
c123 stop "zebra.jpg"
c123 pause 5
c123 stop "zoom.au"
c123 pause 20
c123 shutdown
```

Note that in the actual `mserv2.cmds` file, commands for different clients will be interleaved. The leading “c” indicates customer. Supplier ids start with “s”.

Customers will log transactions to a per-customer log file. If the customer id is 123 then the log file should be `customer123.log` in the current directory.

As mentioned before, the main thread reads the commands file and executes commands prefixed with the appropriate customer id. Executing a command involves creating and enqueueing a request in the server request queue. This will, of course, involve some synchronization operations. The PAUSE command is special and does not generate a request to the server. Rather, the client thread simply pauses for the specified amount of time. The START command is also a bit special. The main thread will simply spawn a thread to handle the START

request. This involves enqueueing a request to the server, copying the media bits to a temporary file (by passing the reply segment back and forth as many times as necessary) and then spawning a separate process to “play” the media. When the main thread reads the corresponding STOP command, it will issue the STOP request to the server, terminate the worker thread, remove the temporary file and do any additional cleanup required.

Note that this model will have concurrent media players “popping up” on the display console.

Initialization File (mserv2.init)

```
qkey 3145          # well-known “key” for request key shared memory segment (created by server)
nworkers 5         # number of server “worker” threads
ncustomers 3      # number of customer processes
c1 c2 c3          # valid customer ids
nsuppliers 2
s1 s2
smax 10           # max number of media files “managed” by server
cmax 5            # max number of files that can be concurrently “checked out” by a customer
```

main()

This space for rent.

Client-Server Communication (System V Shared Memory)

This space for rent.

Packaging

Place your code in a directory along with a README file describing how to compile and run your program and indicating any special capabilities you have implemented. Your directory should contain a sample mserv.init. You can package your code as a single file (mserv.c) or break it up into pieces like (main.c workers.c clients.c). Be sure you properly extern shared data if you break your program into separate source files!

System V Shared-Memory Resources

Several chapters (3, 12-14) from:

W. Richard Stevens
“Unix Network Programming: Volume 2 Second Edition Interprocess Communication”
Prentice-Hall 1999

will be your bible for this assignment. You can also access online documentation through man pages and docs.sun.com.

Turnin / Late Policy

To turn in your assignment, create a tarball containing the project directory and email to pwh@cc.gatech.edu with the Subject line: "CS4210 P2 SUBMISSION". Only one member of the group should submit the project. Make sure your submission includes the names of all group members. The standard late policy applies. Projects can be submitted up to five school days late. A 5% penalty will be applied for each day late. Projects will not be accepted more than 5 days late.

-- Good luck!! Have fun