Unix System Programming - Chapter 2, part b

Neal Nelson

The Evergreen State College

Mar 23, 2010

USP Chapter 2.7 to 2.13

- Thread-safe functions
- Static objects in C
- A list object implementation
- Process environment
- Process termination
- Assignment for Week 1, part b, Message Logging

Reentrant functions

- Reentrant is an older term that came about before the now more common term thread-safe.
- Reentrant generally means
  - Execution can re-enter code before it has finished (eg, recursion)
  - Code is reentrant if it has no side effects (pure functional; no state)
  - Generally this works if parameters and variables are either constant (read only) or automatic storage class (and thus on the stack and not shared)
  - Special care must be taken with reference objects (like strings)
  - Static or allocated storage classes can make code non-reentrant
  - Reentrant means recursion-safe, that is, safe to use in recursive algorithms if used properly
  - Reentrant is one step toward thread-safe and may have been used in the earlier days to mean thread-safe.

Thread-safe functions

- Thread-safe is a more general term that means different threads of execution using either shared code or shared data can execute concurrently without corruption. We an acheive this by having
  - code has no side effects (pure functional, no non-local state) or,
  - code is reentrant with proper management of shared parameters and no shared (static) variables, or
  - static shared variables are properly protected using synchronization tools (locks, mutual exclusion, critical regions, semaphores, etc).
- Thread-safe includes dealing with signal interrupts which are concurrent threads

strtok is not thread-safe (p39)

- `strtok` second argument is a const string of allowed token delimiters - thread safe
- `strtok` first parameter is the string to scan
- Each successive call to `strtok`
  - scans for a token from the current scan position,
  - uses an internal static variable to keep track of the current scan position - not thread-safe,
  - inserts a 0 at the end of the token in the input string, and
  - returns a pointer to the start of the token just scanned.
- When using `strtok` pass NULL for parameter one after the first call
- `strtok` changes your input string in place (violating guideline 7) - can lead to not thread-safe
- Copy your string before passing it to strtok to be thread-safe

A thread-safe strtok_r (p39)

- POSIX `strtok_r` uses an additional parameter provided by the caller that is used to store the current scan position
- `strtok_r` returns NULL when it reaches the end of the scan string
- Multiple threads calling `strtok_r` each have their own scan position, so concurrent scanning is now possible
- The callers are responsible for providing the storage for scan position
- The actual scan position state is maintained by `strtok_r`
- Callers must not modify the scan pointer they use with `strtok_r`

A thread-safe strtok_r (cont)

- Question: Is the input string in `strtok_r` still modified with a 0 at the end of each token?
  - It should not be in order to be more easily used in a thread-safe way
  - It might be in order to be compatable with `strtok`
  - I cannot see where the man page says one way or the other; experimental verification is in order here
  - If it is modified, then does `strtok_r` treat 0 inserted by other threads another implicit delimiter?

Writing thread-safe functions

- Thread-safe is all about shared data in an object state. But sharing can be very difficult to predict in non-functional languages. Here are some string thread-safe guidelines
  - Use only automatic (stack) storage
  - Make caller keep track of state between calls
  - Make caller provide buffer allocations for copy of return data
  - Do not modify caller input data or else make copies
  - Make all local mallocs match local free in all code paths in your code

Writing thread-safe functions

- The previous guidelines for writing thread-safe functions pretty much say to write pure functional code; no side effects. That may be unrealistic to achieve.
- Somewhat less restrictive, we can make code recursion-safe (reentrant code) and then carefully attend to thread-safe parameter passing and static shared variables.
- Analyze potential concurrency carefully.
- Explicitly design code around shared data using concurrency tools as needed.

Static objects in C

- A little OOP using static storage
  - Global variables in a file outside a block are static storage
  - Global variables in a file can be single-object instance data
  - `static` keyword makes them private to the file
  - static functions in the file are private methods on the object data
  - non-static functions in the file are public methods on the object data
- Again, the objects in this scheme are single-instance, one-per-file.

Use of Static Variables - Bubblesort exchange count

- Refer to the program 2.5 p41 bubblesort.c listing
- `static int count` is instance data that holds internal state information between function calls (in onepass)
- This program illustrates encapsulation and implementation hiding plus simple object state (count).
- `static int onepass` is private method operating on count
- The other functions are public methods
- The code is clearly not intended to be thread-safe
- Take a moment to answer storage class and linkage questions in Exercise 2.20 p42
  - `onepass` has internal linkage. The other functions have external linkage
  - The `count` variable has internal linkage and static storage
  - All other variables are block-local; they have no linkage and automatic storage

Signal counter exercise

- Consider a multi-threaded example similar to the bubblesort counter
- A static variable is used by a signal handler to keep track of the a count of signal interrupts (sound familiar?)
- The signal counter setting is now multi-threaded
- Think about how you can organize the program to use implementation hiding like the bubblesort example
- Think about how you can be sure the signal counter thread-safe

A list object implementation in program 2.6, 2.7 pp42-46

- Refer to the program 2.6 listlib.h and program 2.7 listlib.c listings
- List items are (time,string) pairs
- The listlib program implements a single list structure
- The list can be added-to but not otherwise changed
- The list can be traversed
- The program keeps track of multiple simltaneous traversals
- The program is useful for logging.
- Can the traversal functions be used in a recursive setting, that is, are all the functions reentrant? (ie, is the code recursion-safe?)
- Is the program thread-safe? Argue why or why not. How do you organize your argument?

listlib program design

- Make sure callers cannot modify the list
- list elements are (time, string), where time is a value but string is a reference object.
- Reference objects must be treated as values and copied rather than shared state in order to maintain thread-safety
  - Don't give callers pointers into the list or into list elements
  - When the user passes a string element to add, make a copy to put in the list
  - When the user retrieves a list element, return a copy of the list element string to pass back.
  - Caller is responsible for freeing memory of the list element copies returned on retrieval

listlib program design (cont)

- Provide a way to support multiple simultaneous traversals
  - Could use the `strtok_r` strategy and require the caller to provide the storage for traversal state and pass it as an extra parameter.
  - The textbook says no, that would give the caller internal access to the list because the traversal state is stored as a pointer into the list
  - Instead, the code maintains traversal states on behalf of callers by keeping a key-indexed table of traversal states and assigning traversal keys to callers. Callers must pass their traversal keys to the traversal functions
  - Note - the `strtok_r` strategy could be used if the traversal state were a list index instead of a list pointer. But keeping traversal state in the caller storage space is asking for trouble. Better to keep traversal state internally.

listlib.h interface program 2.6

- The listlib.h header file has the public interface to the listlib functions

```
typedef struct data_struct
{   time_t time;
    char *string; /* only pointer allocated */
} data_t;

int accessdata(void); /* return traversal key */
int adddata(data_t data);
int freekey(int key);
int getdata(int key, data_t *datap);
```

- the string field of data_t list elements requires programmer management of allocation (malloc, free)

listlib implementation program 2.7

- `adddata` inserts a copy of the `data_t` item at the end of the list
- `accessdata` returns an integer key for traversing the data list. Each key produces an independent traversal starting from the beginning of the list.
- `getdata` returns a copy of the next `data_t` list element via the second parameter. The copy of the string field requires a malloc in getdata and the caller is responsible for the corresponding free.
- `getdata` returns NULL for the string field at the end of the list and automatically frees the key.
- `freekey` allows the caller to release a traversal key before reaching the end of the list.
- All the functions return -1 on error and set errno.
- If successful, `accessdata` returns a valid non-negative key and the other functions return 0.

listlib implementation program 2.7 (cont)

- Traversal keys are just indices into a fixed size array of traversal pointers. Note that the array is dynamically allocated and grows dynamically as needed (Mid-page 44 at the calloc and bottom third page at the realloc)
- Study the error handling in each of the functions as a model for your programs. USP p45 bottom and 46 top describes the failure cases.
- The program is recursion-safe, but not thread-safe (can you see why?)
- The program does not have fixed limits, but then it does have potential space leak problems if not used properly. Malloc and free are not in the same text module and there is no limit to the size of strings or traversal tables.

keeplog and keeploglib programs 2.8, 2.9 (pp47-48)

- keeplog (p47) is a program that uses listlib to keep a history of commands executed in a mini-shell.
- keeplog uses the listlib program to record the mini-shell commands typed by the user.
- keeplog runs in a loop reading commands, one per line, running and saving them in the history list until EOF.
- keeplog uses runproc in keeploglib (p48) to execute commands and save them in the history list.
- keeplog uses the `system` library function call to execute the commands
- keeplog watches specifically for the "history" command to display the history list to the standard output (stdout file).
- Study this program to improve your shell (later) and as a model for your message log assignment for this week.

Process Environment

- Processes begin with an environment list of (variable,value) pairs represented as an array of pointers to strings of the form *name* = *value*. Environment variables are like HOME, PATH, and others that supply system-specific or user-specific information in setting defaults within a program. Type `env` at the command line to see the environment variables of your command line shell.
- Recall the environment array and the argv command line array are stored in the high memory of the program image in main memory
- The end of the environment list array is NULL
- Processes typically inherit the environment list from the process just before `exec`. See the `exec` family of system calls for details.

The extern variable

- In C programming, the extern variable `environ` points to the process environment list when the process begins executing.
- The following program 2.22 p49 prints the environment list just like the command line env.

```
extern char **environ;

int main(void)
{  int i;
   printf("The environment list follows:\n");
   for (i = 0; environ[i] != NULL; i++)
      printf("environ[%d]: %s\n", i, environ[i]);
   return 0;
}
```

The `getenv` system call

- The system call `getenv` can be used in a C program to fetch the value of a named environment variable
- The function returns a pointer to the string for the environment variable value, or NULL if there is no such environment variable
- Copy the result string if you want to keep it around, getenv might be using a static buffer for the return string and another call will wipe out your string.
- The Exercise 2.24 program p51 is a very nice example of converting a PATH string to an argv array using getenv and then makeargv. This would be a good test for your makeargv.

Process Termination

- When a process terminates the operating system must, among other things
  - close open files
  - deallocate process resources, such as virtual memory, locks, etc.
  - update appropriate statistics on resource usage and record process status
  - cancel pending timers and signals,
  - notify the parent in response to a parent `wait`
- A process does not completely release its resources after termination until the parent waits for it
- If its parent is not waiting for it, the process becomes a *zombie*
- A zombie hangs around and holds its resources until a parent waits for it
- The init process collects orphaned zombies (parent dies before waiting) and periodically waits for them so zombies can release resources and rest peacefully.

Normal Process Termination

- A process may terminate normally or abnormally
- A normal process termination happens under the following conditions
  - A call to exit(k), where k is a small integer indicating exit status to the parent (or _Exit, _exit as well, apparently)
  - A `return k` from main, which is the same as exit(k).
  - Implicit return from main by falling off the end of main. This is the same as exit(0), where 0 is the status code for successful termination.
- A normal termination does the following in order
  - Calls user defined exit handlers that were registered by the `atexit` system call.
  - Flushes the buffers of any open streams and closes them.
  - Removes all temporary files
  - Terminates control

Exit status and exit handlers in normal process termination

- The exit status
  - This is a small integer *k* passed via `exit(k)` or `return k`
  - The exit status is programmer defined
  - Chapter 3 discusses how a parent process can determine the exit status of its children when it does a `wait`.
- User defined exit handlers on normal exit
  - The `atexit` system call installs a user-defined exit handler.
  - Multiple calls to `atexit` can be made to install multiple exit handlers
  - Exit handlers are invoked on normal exit in the *reverse order that they were installed.*
  - See Program 2.10 p53 for an example of registering an and invoking an exit handler.

Abnormal Process Termination

- An abormal termination can happen by
  - A call to `abort`
  - Processing of an external event signal like cntl C that causes termination
  - Processing an internal error such as a seg fault (attempt to access an illegal memory location).
- Abnormal terminations do not call user-installed exit handlers
- Abnormal terminations may produce core dumps

Assignment 1, part b

- Assignment 1, part b, Message Logging Exercise Section 2.13
- Assignment details are on the USP assignments page and in the texbook.
- You should be able to either use or modify the listlib function implemented in the text.
- Analyze your code according to the Robbins function-writing checklist
- Build your code in a single directory and use a makefile. Start by downloading the textbook examples and testing them.
- Write a test program for your message logger
- Include a README.txt file in your directory that explains what you did, how you tested your code, how to read your code, and how to run your code. See the guidelines for program handin on the class web page.

- All Done