CptS 360 (System Programming)
Unit 9: Process Environment

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- Processes are fundamental components of operating systems.
- Need to understand what data is kept and not kept for a process.
- Understanding threads requires an understanding of processes.

References

- Stevens & Rago Ch. 7
- *man* pages

The `main()` Function

- prototype:

  `int main(int argc, char *argv[], char *environ[])`

  (but you can leave off arguments you don't use)
- OS promises to call a function with this name first.
- little-known guarantee:

  `argv[argc] == NULL`
- handle command line arguments with *getopt(3)* and *getopt_long(3)* (upcoming lab)

(Run the `demos/d`*n*`_main_return` demo.)

Process Termination

- return status from `main()`
- *exit(3)*
  - normal exit function
  - closes all open files
  - frees all memory
  - accepts `EXIT_SUCCESS`, `EXIT_FAILURE`, or `status`
- *atexit(3)* and *on_exit(3)*
  - set *exit handlers*
  - *on_exit(3)* provides more functionality
- *_exit(2)* is like *exit(3)*, but doesn't call exit handlers
- *abort(3)*
  - exits the program "abnormally"
  - uses signal `SIGABRT` (later), which ...
  - may cause core dump

Environment Variables

- usually set in the shell (including `~/.profile` or `~/.bashrc`)

  `$ export` *`NAME=value`*
- handy for persistent, user-specific configuration variables
- inherited from parent process (which might be the shell)
- prototype:

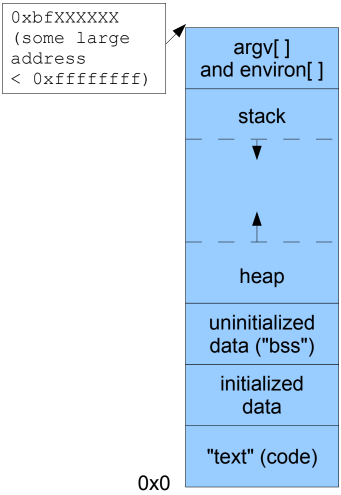
  ```
  extern char **environ;
  ```

  Points to NULL-terminated array of pointers to (nul-terminated) environment strings, each of the form "*`NAME=value`*".

  Better to use convenience functions:
  - *getenv(3)*
  - *putenv(3)*
  - *setenv(3)*
  - *unsetenv(3)*

(Run the `demos/dn_getenv` demo.)

Memory Layout of a Running Program



In the executable file:

- "text" (compiled code)
- initialized data
- *size* of uninitialized ("BSS") data
- a list of libraries to link in (for dynamic linking only)

Set up by OS at run time and dynamically resized:

- stack
- heap

see: *size(1)*

Shared (and Unshared) Libraries

- non-shared (“`.a`”) libraries
  - bound at link time
  - faster program starts
- shared (“`.so`”, “`.dll`”, or “`.dylib`”) libraries
  - bound at compile time
  - save on disk space
  - allow upgrades more easily

dlopen(3) Possibilities

These allow dynamic (your program controls) access of "`.so`" files for:

- customer-provided compiled code
  (provided your customer is trustworthy)
- selectively load pre-digested data at run time

(Run the `demos/d`*n*`_dlmath` demo.)
(also see original code from *dlopen(3)* man page)

Heap Memory Allocation

- *malloc(3)*
  allocate from heap (uninitialized, fast)
- *calloc(3)*
  like *malloc()*, but clears data (a bit slower)
- *realloc(3)*
  allocates or resizes an already-allocated heap area
- *free(3)*
  frees a previously-allocated area

How Heap Allocation Really Works

- deliberately removed from POSIX (but usually in API anyway)
  - *brk(2)*
    sets the heap limit
  - *sbrk(2)*
    increments the heap limit
- Don't use these unless you can design a better *malloc(3)*. (Good luck with that, but if you do, *don't* use *malloc(3)*.)

Stack Memory Allocation

alloca(3)

- It's non-POSIX, but seems to have been in UNIX 32V and similar.
- It allocates memory on its caller's stack.
  Q: On a MIPS machine, it's easy to allocate stack space. How?
- Do NOT call *free()* on this memory.
  Q: Why not?
- Memory is "freed" automatically when caller returns (so be careful!) *or* does a *longjmp(3)* (see below)
- Variable-length arrays are more elegant, if they work.
  (Run the `demos/dn_variable_stack_allocation` demo.)

Optimizing Memory Allocation

- can be a big win for high-performance systems
- especially useful if structures are small
- benchmarking critical
- earn big bucks!

read on...

Memory Allocation Optimization Using *malloc(3)*

malloc() and *free()* whole arrays of same-sized structures at a time

- especially useful when you have lots of small structures
- keep reference counts on those structs or arrays
- free array when `max(refCounts) = 0`
- maintain your own *free list*
  - Your *free()* (change the name) puts `struct` on free list.
  - Your *malloc()* (also change the name) calls *malloc()* only when free list is exhausted.

setjmp(3) and *longjmp(3)* Functions

- These form a nonlocal "goto".
- Alternative to error handling via the calling stack.
- *setjmp(3)*
  returns 0 if called directly, nonzero if returning from a *longjmp()*.
- *longjmp(3)*
  returns to matching *setjmp()* environment with value *val*.
- Low-level way to handle non-fatal errors.

We'll cover these more in the lab.

setjmp()/*longjmp()* Cautions

- Have to be careful about automatic (stack) variables in calling frame.
  - Sometimes, they're "rolled back" (to the time of the *setjmp(3)* call.).
  - Sometimes, they're the same as at the time *longjmp(3)* was called, which is usually what you want.

  (Run the `demos/dn_jmp_demo` demo.)
- global and static variables are left unchanged
- To prevent rollback, use `volatile` storage attribute (meaning "Don't put this variable in a register").

Resource Limits

- *getrlimit(2)*
  This returns your process's soft and hard limits.
- *setrlimit(2)*
  This lets you set your soft and hard limits, if you have that capability.