CS631 - Advanced Programming in the UNIX Environment

Process Environment, Process Control

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/
Memory Layout of a C Program

memory-layout.c

high address

low address
Memory Layout of a C Program
Memory Layout of a C Program

memory-layout.c

high address

command-line arguments and environment variables

low address

text
Memory Layout of a C Program

memory-layout.c

- high address
- command-line arguments and environment variables
- initialized data
- text
- read from program file by exec
Memory Layout of a C Program

memory-layout.c

high address

command-line arguments and environment variables

uninitialized data (bss)

initialized data

low address

text

initialized to zero by exec

read from program file by exec
Memory Layout of a C Program

memory-layout.c
Memory Layout of a C Program

memory-layout.c
Memory Layout of a C Program

memory-layout.c
Memory Layout of a C Program

memory-layout.c

- high address
  - stack
  - heap
  - uninitialized data (bss)
  - initialized data
  - text

- command-line arguments and environment variables
  - initialized to zero by exec
  - read from program file by exec

Lecture 05: Process Environment, Process Control

October 6, 2019
Memory Layout of a C Program

See also:

- /proc/self/map
- pmap(1) and pmap(9)

Obligatory "Smashing The Stack For Fun And Profit" links:

https://insecure.org/stf/smashstack.html
The **main** function

```c
int main(int argc, char **argv);
```
The main function

```c
int main(int argc, char **argv);
```

- C program started by kernel (by one of the `exec` functions)
- special startup routine called by kernel which sets up things for `main`
  (or whatever entrypoint is defined)
- `argc` is a count of the number of command line arguments (including
  the command itself)
- `argv` is an array of pointers to the arguments
- it is guaranteed by both ANSI C and POSIX.1 that `argv[argc] == NULL`
Process Creation

On Linux:

$ cc -Wall entry.c
$ readelf -h a.out | more

ELF Header:

[...]

Entry point address: 0x400460
Start of program headers: 64 (bytes into file)
Start of section headers: 4432 (bytes into file)

$ objdump -d a.out

[...]

00000000000400460 <_start>:

400460: 31 ed xor %ebp,%ebp
400462: 49 89 d1 mov %rdx,%r9

[...]

$
Process Creation

Linux: glibc/sysdeps/x86_64/start.S

000000000000401058  <_start>:
  401058:  31 ed           xor  %ebp,%ebp
  40105a:  49 89 d1       mov  %rdx,%r9
  40105d:  5e           pop  %rsi
  40105e:  48 89 e2       mov  %rsp,%rdx
  401061:  48 83 e4 f0    and  $0xfffffffffffffff0,%rsp
  401065:  50           push %rax
  401066:  54           push %rsp
  401067:  49 c7 c0 e0 1a 40 00 mov  $0x401ae0,%r8
  40106e:  48 c7 c1 50 1a 40 00 mov  $0x401a50,%rcx
  401075:  48 c7 c7 91 11 40 00 mov  $0x401191,%rdi
  40107c:  e8 2f 01 00 00 00 callq 4011b0 <__libc_start_main>
  401081:  f4          hlt
  401082:  90           nop
  401083:  90           nop
**Process Creation**

Linux: glibc/csu/libc-start.c

```c
STATIC int
LIBC_START_MAIN (int (*main) (int, char **, char ** MAIN_AUXVECDECL),
    int argc, char **argv,
    _typeof (main) init,
    void (*fini) (void),
    void (*rtld_fini) (void), void *stack_end)
{
    [...]  
    result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);

    exit (result);
}
```
Process Creation

NetBSD: /usr/src/lib/csu/common/crt0-common.c

void___start(void (*cleanup)(void), /* from shared loader */
           const Obj_Entry *obj, /* from shared loader */
           struct ps_strings *ps_strings)
{
    ...
    atexit(_fini);
    _init();

    exit(main(ps_strings->ps_nargvstr, ps_strings->ps_argvstr, environ));
}
Process Creation

$ cc -Wall entry.c
$ readelf -h a.out | more
ELF Header:

 [...] 
  Entry point address: 0x400460
  Start of program headers: 64 (bytes into file)
  Start of section headers: 4432 (bytes into file)

$ objdump -d a.out

 [...] 
000000000000400460 <_start>:
  400460: 31 ed xor %ebp,%ebp
  400462: 49 89 d1 mov %rdx,%r9

 [...] 

http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html
Process Creation

$ cc -e foo entry.c
$ ./a.out
Foo for the win!
Memory fault
$ cc -e bar entry.c
$ ./a.out
bar rules!
$ echo $?
1
$ cc --std=c89 entry.c
$ ./a.out
Hooray main!
$ echo $?
13
$
Process Termination

There are 8 ways for a process to terminate.

Normal termination:

- return from `main`
- calling `exit`
- calling `_exit` (or `_Exit`)
- return of last thread from its start routine
- calling `pthread_exit` from last thread
Process Termination

There are 8 ways for a process to terminate.

Normal termination:
- return from main
- calling exit
- calling _exit (or _Exit)
- return of last thread from its start routine
- calling pthread_exit from last thread

Abnormal termination:
- calling abort
- terminated by a signal
- response of the last thread to a cancellation request
exit(3) and _exit(2)

```c
#include <stdlib.h>
void exit(int status);
void _Exit(int status);

#include <unistd.h>
void exit(int status);
void _exit(int status);
```

- _exit and _Exit
  - return to the kernel immediately
  - exit required by POSIX.1
  - _Exit required by ISO C99
  - synonymous on Unix

- exit does some cleanup and then returns
- both take integer argument, aka exit status
atexit(3)

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

- Registers a function with a signature of `void funcname(void)` to be called at exit
- Functions invoked in reverse order of registration
- Same function can be registered more than once
- Extremely useful for cleaning up open files, freeing certain resources, etc.

exit-handlers.c
Lifetime of a UNIX Process
Lifetime of a UNIX Process
Lifetime of a UNIX Process
Lifetime of a UNIX Process

- User functions
- Main function
- C start-up routine
- Exit function
- Kernel

Call and return arrows indicate the flow of execution.
Lifetime of a UNIX Process
Lifetime of a UNIX Process

- exit
- _Exit
- exit
- _Exit
- exit
- _Exit
- exec
- C start-up routine
- main function
- exit (does not return)
- exit (does not return)
- exit (does not return)
- exit
- exit
- return
- return
- return
- return
- call
- call
- call
- call
- call
- call
- call
- call
- call
- call
- exit handler
- exit handler
- ...
Exit codes

$ cc -Wall --std=c89 hw.c
hw.c: In function 'main':
hw.c:7: warning: control reaches end of non-void function
$ ./a.out
Hello World!
$ echo $? 
13
$
Exit codes

$ cc -Wall --std=c89 hw.c
hw.c: In function 'main':
hw.c:7: warning: control reaches end of non-void function
$ ./a.out
Hello World!
$ echo $?
13
$ cc -Wall --std=c99 hw.c
$ ./a.out
Hello World!
$ echo $?
0
$
Environment List

Environment variables are stored in a global array of pointers:

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

The list is **null terminated**.

These can also be accessed by:

```c
#include <stdlib.h>

char *getenv(const char *name);
int putenv(const char *string);
int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
```
Environment List

Environment variables are stored in a global array of pointers:

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

The list is null terminated.

These can also be accessed by:

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv(const char *string);
int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
```

```
int main(int argc, char **argv, char **anvp);
```
Memory Allocation

```c
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
void *alloca(size_t size);
void free(void *ptr);
```

- `malloc` – initial value is indeterminate.
- `calloc` – initial value set to all zeros.
- `realloc` – changes size of previously allocated area. Initial value of any additional space is indeterminate.
- `alloca` – allocates memory on stack

Now consider manipulation of the environment by your program...
Memory Layout of a C Program

- **text**: read from program file by `exec`
- **initialized data**: read from program file by `exec`
- **uninitialized data (bss)**: initialized to zero by `exec`
- **heap**: command-line arguments and environment variables
- **stack**: high address
Memory Layout of a C Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>7F7FFFA3624C</td>
</tr>
<tr>
<td>argv</td>
<td>7F7FFFA36240</td>
</tr>
<tr>
<td>extern char **environ</td>
<td>601C38</td>
</tr>
<tr>
<td>envp</td>
<td>7F7FFFA36238</td>
</tr>
<tr>
<td>_.=/a.out from environ</td>
<td>7F7FFFA40478</td>
</tr>
<tr>
<td>main (function)</td>
<td>400B6B</td>
</tr>
<tr>
<td>func (function)</td>
<td>400B40</td>
</tr>
<tr>
<td>func2 (function)</td>
<td>400F07</td>
</tr>
<tr>
<td>num (initialized global int)</td>
<td>601A58</td>
</tr>
<tr>
<td>num2 (uninitialized global int)</td>
<td>601C48</td>
</tr>
<tr>
<td>string (initialized global char *)</td>
<td>601A50</td>
</tr>
<tr>
<td>string2 (uninitialized global char *)</td>
<td>601C40</td>
</tr>
<tr>
<td>array[] (uninitialized, fixed-size char[] on BSS)</td>
<td>601C60</td>
</tr>
<tr>
<td>array[] ends at</td>
<td>60B8A0</td>
</tr>
<tr>
<td>func_array[] (like 'array[]', but on stack)</td>
<td>7F7FFFA36250</td>
</tr>
<tr>
<td>func_array[] ends at</td>
<td>7F7FFFA362E0</td>
</tr>
<tr>
<td>malloced area begins at</td>
<td>740460F02000</td>
</tr>
<tr>
<td>malloced area ends at</td>
<td>740460F1A6A0</td>
</tr>
<tr>
<td>*environ[0] itself at</td>
<td>7F7FFFA36E8</td>
</tr>
<tr>
<td>environ[0] (._=/a.out) at</td>
<td>7F7FFFA40478</td>
</tr>
<tr>
<td>getenv(&quot;_&quot;) at</td>
<td>7F7FFFA4047A</td>
</tr>
<tr>
<td>getenv(&quot;USER&quot;) = jschauma at</td>
<td>7F7FFFA40598</td>
</tr>
<tr>
<td>getenv(&quot;USER&quot;) = root, after setenv</td>
<td>740460F20A6</td>
</tr>
<tr>
<td>func: stack frame around</td>
<td>7F7FFFA3621C</td>
</tr>
<tr>
<td>func2 (from func): stack frame around</td>
<td>7F7FFFA361FC</td>
</tr>
<tr>
<td>func popped off</td>
<td></td>
</tr>
<tr>
<td>func2 (from main): stack frame around</td>
<td>7F7FFFA3621C</td>
</tr>
</tbody>
</table>
Memory Layout of a C Program

On NetBSD:

$ cc hw.c
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for NetBSD 5.0, not stripped
$ ldd a.out
a.out:
   -lc.12 => /usr/lib/libc.so.12
$ size a.out
    text   data   bss   dec   hex filename
    2301    552   120   2973  b9d a.out
$ objdump -d a.out > obj
$ wc -l obj
    271 obj
$
Memory Layout of a C Program

On Mac OS X:

$ cc hw.c
$ file a.out
a.out: Mach-O 64-bit executable x86_64
$ otool -L a.out
a.out: 
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 125.2.11)
$ size a.out
__TEXT __DATA __OBJC others dec hex 4096 4096 0 4294971392 4294979584 100003000
$ otool -t -v a.out > obj
$ wc -l obj
  32 obj
$
Memory Layout of a C Program

On Linux:

$ cc hw.c
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
$ ldd a.out
linux-gate.so.1 => (0x00c66000) libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x006b4000)
/lib/ld-linux.so.2 (0x005fe000)
$ size a.out
    text  data  bss   dec  hex filename
        918   264     8  1190   4a6  a.out
$ objdump -d a.out >obj
$ wc -l obj
225  obj
$
Memory Layout of a C Program

On NetBSD:

```bash
$ cc -static hw.c
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, for NetBSD 5.0, not stripped
$ ldd a.out
ldd: a.out: unrecognized file format [2 != 1]
$ size a.out
  text   data   bss   dec   hex filename
  151877  4416   16384  172677  2a285 a.out
$ size a.out.dyn
  text   data   bss   dec   hex filename
   2301    552    120   2973   b9d a.out
$ objdump -d a.out > obj
$ wc -l obj
  35029 obj
```
Memory Layout of a C Program

On Mac OS X:

```
$ cc -static hw.c
ld: library not found for -lcrt0.o
collect2: ld returned 1 exit status
$
```
Memory Layout of a C Program

On Linux:

$ cc -static hw.c
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, for GNU/Linux 2.6.15, not stripped
$ ldd a.out
/usr/bin/ldd: line 161: /lib64/ld-linux-x86-64.so.2: cannot execute binary file
not a dynamic executable
$ size a.out
    text  data  bss   dec   hex filename
510786  1928  7052  519766  7ee56 a.out
$ objdump -d a.out >obj
$ wc -l obj
114420 obj
$
Process limits

$ ulimit -a
 time(cpu-seconds) unlimited
 file(blocks) unlimited
 coredump(blocks) unlimited
 data(kbytes) 262144
 stack(kbytes) 2048
 lockedmem(kbytes) 249913
 memory(kbytes) 749740
 nofiles(descriptors) 128
 processes 160
 vmemory(kbytes) unlimited
 sbsize(bytes) unlimited
$
getrlimit(2) and setrlimit(2)

```c
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```

Changing resource limits follows these rules:

- A **soft limit** can be changed by any process to a value less than or equal to its hard limit.
- Any process can lower its **hard limit** greater than or equal to its soft limit.
- Only superuser can raise **hard limits**.
- Changes are per process only.
getrlimit(2) and setrlimit(2)

```c
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```

Changing resource limits follows these rules:

- a *soft limit* can be changed by any process to a value less than or equal to its hard limit
- any process can lower its *hard limit* greater than or equal to its soft limit
- only superuser can raise *hard limits*
- changes are per process only (which is why `ulimit` is a shell built-in)
Process Control

Review from our first class, the world’s simplest shell:

```c
int main(int argc, char **argv)
{
    char buf[1024];
    pid_t pid;
    int status;

    while (getinput(buf, sizeof(buf))) {
        buf[strlen(buf) - 1] = '\0';
        if((pid=fork()) == -1) {
            fprintf(stderr, "shell: can't fork: %s\n", strerror(errno));
            continue;
        } else if (pid == 0) {
            /* child */
            execlp(buf, buf, (char *)0);
            fprintf(stderr, "shell: couldn't exec %s\n", strerror(errno));
            exit(EX_DATAERR);
        }
        if ((pid=waitpid(pid, &status, 0)) < 0)
            fprintf(stderr, "shell: waitpid error: %s\n", strerror(errno));
    }
    exit(EX_OK);
}
```
Process Identifiers

```c
#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);
```

Process ID's are guaranteed to be unique and identify a particular executing process with a non-negative integer.

Certain processes have fixed, special identifiers. They are:

- **swapper**, process ID 0 – responsible for scheduling
- **init**, process ID 1 – bootstraps a Unix system, owns orphaned processes
- **pagedaemon**, process ID 2 – responsible for the VM system (some Unix systems)
fork(2) causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent’s descriptors.
- The child process’ resource utilizations are set to 0.

Note: no order of execution between child and parent is guaranteed!
fork(2)
fork(2)
fork(2)

$ cc -Wall forkflush.c
$ ./a.out
a write to stdout
before fork
pid = 12149, glob = 7, var = 89
pid = 12148, glob = 6, var = 88
$ ./a.out | cat
a write to stdout
before fork
pid = 12153, glob = 7, var = 89
before fork
pid = 12151, glob = 6, var = 88
$
The \texttt{exec(3)} functions

```c
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
int execv(const char *pathname, char * const argv[]);
int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
int execvp(const char *filename, char *const argv[]);

int execle(const char *pathname, const char *arg0, ... /* (char *) 0, char *const envp[] */);
int execve(const char *pathname, char * const argv[], char * const envp[]);

The \texttt{exec()} family of functions are used to completely replace a running process with a a new executable.

- if it has a \texttt{v} in its name, argv's are a vector: const * char argv[]
- if it has an \texttt{l} in its name, argv's are a list: const char *arg0, ... /* (char *) 0 */
- if it has an \texttt{e} in its name, it takes a char * const envp[] array of environment variables
- if it has a \texttt{p} in its name, it uses the PATH environment variable to search for the file
```
wait(2) and waitpid(2)

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```

A parent that calls wait(2) or waitpid(2) can:

- block (if all of its children are still running)
- return immediately with the termination status of a child
- return immediately with an error
wait(2) and waitpid(2)

Differences between wait(2), wait3(2), wait4(2) and waitpid(2):

- wait(2) will block until the process terminates, waitpid(2) has an option to prevent it from blocking
- waitpid(2) can wait for a specific process to finish
- wait3(2) and wait4(2) allow you to get detailed resource utilization statistics
- wait3(2) is the same as wait4(2) with a wpid value of -1
wait(2) and waitpid(2)

Once we get a termination status back in status, we’d like to be able to determined how a child died. We do this with the following macros:

- **WIFEXITED(status)** – true if the child terminated normally. Then execute **WEXITSTATUS(status)** to get the exit status.

- **WIFSIGNALED(status)** – true if child terminated abnormally (by receiving a signal it didn’t catch). The we call:
  - **WTERMSIG(status)** to retrieve the signal number
  - **WCOREDUMP(status)** to see if the child left a core image

- **WIFSTOPPED(status)** – true if the child is currently stopped. Call **WSTOPSIG(status)** to determine the signal that caused this.

Additionally, waitpid’s behavior can be modified by supplying **WNOHANG** as an option, which says that if the requested pid has not terminated, return immediately instead of blocking.
What if we don't `wait(2)`?
What if we don’t `wait(2)`?
What if we don’t \texttt{wait(2)}? 

\begin{verbatim}
$ cc -Wall zombies.c
$ ./a.out
Let’s create some zombies!

15603 s003 S+ 0:00.00 ./a.out
15604 s003 Z+ 0:00.00 (a.out)

15603 s003 S+ 0:00.00 ./a.out
15604 s003 Z+ 0:00.00 (a.out)
15608 s003 Z+ 0:00.00 (a.out)

15603 s003 S+ 0:00.00 ./a.out
15604 s003 Z+ 0:00.00 (a.out)
15612 s003 Z+ 0:00.00 (a.out)

15603 s003 S+ 0:00.00 ./a.out
15604 s003 Z+ 0:00.00 (a.out)
15612 s003 Z+ 0:00.00 (a.out)
15616 s003 Z+ 0:00.00 (a.out)
\end{verbatim}
Notes and Homework

Reading:
- Stevens, Chapter 7 and 8

Thinking:
- trace process start through the source in NetBSD
- compare return codes on NetBSD of `printf(3)` vs `write(2)`; explain the difference
- Can you overflow the stack by setting many environment variables?
- What is the maximum size an environment variable can be? What happens when you set one larger than that (by e.g. manipulating `envp` directly)?

Other:
- update `memory-layout.c` to match the output on slide 35; play with `envp`, `environ`, `setenv(3)` etc. and see where things end up
- work on your midterm project!