CSCI 4061: Processes and Environment

Chris Kauffman

Last Updated:
Mon Feb 8 03:54:59 PM CST 2021
Logistics

Reading
- Stevens/Rago, Ch 7-8 (Procs / Env)
- Stevens/Rago Ch 3, 4, 5, 6 (I/O + Files)

Goals Today
- Process Lifecycle
- Killing programs
- Process memory layout
- Command Line Args
- Environment Variables
- Start I/O discussion

Labs/HWs
- Lab02 / HW02 due Mon
- Lab03 on Mon, realloc() function,
- HW03: on Mon WNOHANG and parents

Project 1
- Up now
- Due Mon 2/22 11:59pm
- Partners allowed
- Will create Piazza post for finding partners
Process: A “Running” Program

- Most OS’s provide a **Process** abstraction
- Hardware like the CPU just sees a stream of instructions, bits stored, bytes on disk
- OS presents notion of
  - “These instructions are for this running program”
  - “This running program owns this part of memory”
  - “This file was opened by this running program”
- One stored program can create many Processes
- OS is responsible for managing the lives of Processes with fairness and security
Process Life Cycle

- **Processes** (running programs) can be in one of several states
- OS tracks these states and manages transitions between them
- OS uses some internal data structure to track process state, can report states via utilities like `top` and `ps`
**ps and top show running process status**

These shell commands show a STAT or S columns corresponding loosely to process states.

<table>
<thead>
<tr>
<th>STAT</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Common</strong></td>
</tr>
<tr>
<td>R</td>
<td>running or runnable (on run queue)</td>
</tr>
<tr>
<td>S</td>
<td>interruptible sleep (waiting for an event to complete)</td>
</tr>
<tr>
<td>T</td>
<td>stopped, either by a job control signal or being traced.</td>
</tr>
<tr>
<td>Z</td>
<td>defunct (“zombie”) process, terminated but not reaped by parent.</td>
</tr>
<tr>
<td>I</td>
<td>idle (kernel process/thread only)</td>
</tr>
<tr>
<td></td>
<td><strong>Less Common</strong></td>
</tr>
<tr>
<td>D</td>
<td>uninterruptible sleep (usually IO)</td>
</tr>
<tr>
<td>W</td>
<td>paging (not valid since the 2.6.xx kernel)</td>
</tr>
<tr>
<td>X</td>
<td>dead (should never be seen)</td>
</tr>
</tbody>
</table>

Source: man page for ps

We'll continue to discuss Specifics of Zombines and Orphans
Handy Commands

- **top**: interactively observe top running processes, usually sorted by CPU usage
- **ps**: snapshot of running processes filtered on various criteria
- **watch**: repeatedly run a command showing its output on the screen

Interactively observe all processes sorting by top CPU usage

```
> top
press q to quit
Watch processes with command name yes refreshing every 0.1 seconds showing u-ser relevant information on the processes
```

```
> watch -n 0.1 'ps u -C yes'
Press Ctrl-c to end the watch
```
Terminal: Foreground/Background Processes

- Type a program into the terminal, press enter
- Stars a process in the **foreground** of the terminal
  - Input from user typing, output to terminal screen
- Jobs can be run in the **background** as well
  - Usually input must come from somewhere aside from user typing, output should go into a file or it will pollute the terminal

<table>
<thead>
<tr>
<th>Key/Cmd</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-z</td>
<td>Stop/Suspend foreground process, gets prompt back</td>
</tr>
<tr>
<td>Ctrl-c</td>
<td>Terminate foreground process (usually)</td>
</tr>
<tr>
<td>ls &amp;</td>
<td>Run program in background, gets prompt immediately</td>
</tr>
<tr>
<td>bg %2</td>
<td>Moves stopped Job 2 to background and continues it</td>
</tr>
<tr>
<td>fg %4</td>
<td>Moves background Job 4 to foreground</td>
</tr>
<tr>
<td>jobs</td>
<td>List jobs under the control of the terminal</td>
</tr>
<tr>
<td>kill %3</td>
<td>End job 3 nicely</td>
</tr>
<tr>
<td>kill -9 %3</td>
<td>End job 3 unequivocally</td>
</tr>
</tbody>
</table>
Exercise: Basic Job Control

Give a sequence of commands / keystrokes to...

Misbehaving

▸ Compile no_interruptions.c to a program named invincible
▸ Run invincible
▸ Try to end the process by sending it the interrupt signal
▸ In a separate terminal, end the invincible program

Edit / Build Seq

▸ Edit a source file like collatz_funcs.c with vi 😞
▸ Suspend vi (don’t quit it)
▸ Re-build program and run automated tests
▸ Terminate before completing tests
▸ Bring back vi to edit codes
Murdering Processes

Keystrokes to Remember

Ctrl-c  Send the interrupt signal, kills most processes
Ctrl-z  Send the stop signal, puts process to sleep

Easy to Kill

▶ yes spits output to the screen continuously
▶ End it from the terminal it started in
▶ Suspend it then, end it
▶ Kill it from a different terminal

Harder to Kill

▶ Consider the program no_interruptions.c
▶ Ignores some common signals
▶ Need to use the big stick for this one:
  kill -9 1234 OR
  pkill -9 a.out
States of a Living Process

- Note inclusion of Kernel/OS here
- **Interrupt and Sys Calls** start running code in the operating system
- Interrupt/Signal can come from software or hardware
- **Context switch** starts running another process, only happens when one process is safely tucked in and put to **sleep**

Source: *Design of the Unix Operating System* by Maurice Bach
Recall: Program Memory

- What are the 4 memory areas to a C program we’ve discussed OR that you know from previous courses?
- Give an example of how one creates variables/values in each area of memory
Answers: Program Memory

What are the 4 memory areas to a C program we’ve discussed OR that you know from previous courses?

1. Stack: automatic, push/pop with function calls
2. Heap: malloc() and free()
3. Global: variables outside functions, static vars
4. Text: Assembly instructions

Give an example of how one creates variables/values in each area of memory

```c
#include <stdlib.h>

int glob1 = 2; // global var
int func(int *a){
    int b = 2 * (*a); // local stack var
    return b; // de-allocate locals in func()
}

int main(){
    int x = 5; // local stack var
    int c = func(&x); // local stack var that points into heap
    int *p = malloc(sizeof(int)); // local stack var that points into heap
    *p = 10; // modify heap memory
    glob1 = func(p); // allocate func() locals and run code
    free(p); // deallocate heap mem pointed to p
    return 0; // deallocate locals in main()
}

// all executable code is in the .text memory area as assembly instructions
```
More Detailed Process Memory

Source: Unix Systems Programming, Robbins & Robbins
Yet more detailed view (Link)

A detailed picture of the virtual memory image, by Wolf Holzman

### Memory Layout (Virtual address space of a C process)

<table>
<thead>
<tr>
<th>STACK</th>
<th>HIGH MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfp − frame pointer (for main)</td>
<td></td>
</tr>
<tr>
<td>stack pointer (grows downward if func() calls another function)</td>
<td></td>
</tr>
</tbody>
</table>

- **STACK**
  - System
    - env
    - argv
    - argc
    - auto variables for main()
    - auto variables for func()
  - available for stack growth
    - malloc.o (lib*.so)
    - printf.o (lib*.so)
  - available for heap growth

- **SHAREDMEMORY**
  - malloc.o (lib*.a)
  - printf.o (lib*.a)

- **DATA**
  - global variables
  - "...%d...

- **TEXT**
  - malloc.o (lib*.a)
  - printf.o (lib*.a)
  - file.o
  - main.o func(72,73)
  - crfl0.o (startup routine)

- **TEXT**
  - compiled code (a.out)

- **SHAREDMEMORY**
  - malloc.o (lib*.a)
  - printf.o (lib*.a)

- **DATA**
  - global variables
  - "...%d..."

### Expanded view of the stack

#### Stack

<table>
<thead>
<tr>
<th>Offset from current frame pointer (for func())</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() auto variables</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>ra</td>
<td>return address</td>
</tr>
<tr>
<td>stack pointer (top of stack)</td>
<td></td>
</tr>
<tr>
<td>frame pointer points here</td>
<td></td>
</tr>
<tr>
<td>frame pointer</td>
<td></td>
</tr>
<tr>
<td>brk point</td>
<td></td>
</tr>
<tr>
<td>brk point</td>
<td></td>
</tr>
<tr>
<td>stack pointer (top of stack)</td>
<td></td>
</tr>
</tbody>
</table>

- **STACK**
  - mfp − frame pointer (for main)
  - stack pointer (grows downward if func() calls another function)

- **MEMORY**
  - high
  - main.o
  - file.o
  - malloc.o (lib*.a)
  - malloc.o (lib*.so)
  - printf.o (lib*.a)
  - printf.o (lib*.so)

- **TEXT**
  - compiled code (a.out)

- **SHAREDMEMORY**
  - malloc.o (lib*.a)
  - printf.o (lib*.a)

- **DATA**
  - global variables
  - "...%d..."

- **TEXT**
  - malloc.o (lib*.a)
  - printf.o (lib*.a)

- **SHAREDMEMORY**
  - malloc.o (lib*.a)
  - printf.o (lib*.a)

- **DATA**
  - global variables
  - "...%d...

### Low memory

- Stack illustrated after the call func(72,73) called from main(), assuming func defined by:
  ```c
  func(int x, int y) {
    int a;
    int b[3];
    /* no other auto variables */
  }
  ```
  Assumes int = long = char * of size 4 and assumes stack at high address and descending down.

- Expanded view of the stack

- All auto variables and parameters are referenced via offsets from the frame pointer.
- The frame pointer and stack pointer are in registers (for fast access).
- When func returns, the return value is stored in a register. The stack pointer moves to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller’s frame pointer). The caller moves the return value to the right place.
Unix Processes In Memory

- Separate Memory Image for Each Process
- OS + Hardware keeps processes inside their own address space
- Consequence for program dynamic memory allocation?
- Problems with running system calls?

This picture should bother you
Shows a gross simplification but will suffice until later when we discuss **Virtual Memory** system which is maintained by the OS.
Exercise: Memory Problems in C Programs

What you’re up against

- Stack problems: References to stack variables that go away
- Segmentation Faults: Access memory out of bounds for whole program, via heap or via stack
- Null pointers dereference: Often results in a segfault as NULL translates to address 0x0000 which is off limits
- Use of uninitialized: variables don’t have values by default, assign or get something random
- Memory Leaks: malloc() memory that is not used but never free()’d, program gobbles more and more memory
- Examine results of running overflow.c, EXPLAIN OUTPUT

Solutions

- Don’t program in C
- Use a tool to help identify and fix problems
- Valgrind → FREE for Linux Programs
Code for overflow.c

// overflow.c: program traverses memory that it really ought not to by
// walking off the end of an array into parts unknown.

#include <stdio.h>

int main(int argc, char *argv[]){
    char a[3] = {'A','B','C'};
    int i = 0;
    while(1){
        printf("%c",a[i]);
        i++;
        if(i%40 == 0){
            printf("\n");
        }
    }
    return 0;
}

// ## COMPILE AND RUN
// > gcc overflow.c
// > ./a.out
// ABC..^@....E......*V^@^@ ...^?^@^@X.^?^@^@^@^@^@^@^@^@^@
// ^@^@^@9..*V^@^@.....^?^@^@^@^@^@^@^@^@^@^@..K..|
// V^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
// ......M.....
Valgrind: Memory Tool on Linux and Mac

- Valgrind catches most memory errors
  - Use of uninitialized memory
  - Reading/writing memory after it has been free’d
  - Reading/writing off the end of malloc’d blocks
  - Memory leaks
- Source line of problem happened (but not cause)
- Super easy to use, installed on lab machines
- Slows execution of program way down
- Usually install on Linux via
  
  > sudo apt install valgrind

> gcc -g badmemory.c
> ./a.out
-714833203
0
1
4
9
16
0
1
...
5
6
7
8
Segmentation fault (core dumped)
# what now??
Valgrind on Common Problems in badmemory.c

> valgrind ./a.out
==2913308== Memcheck, a memory error detector
==2913308== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2913308== Using Valgrind-3.15.0 and LibVEX; rerun with --help for copyright info
==2913308== Command: ./a.out
==2913308== Conditional jump or move depends on uninitialised value(s)
==2913308== at 0x109189: main (badmemory.c:6)
==2913308==
==2913308== Invalid write of size 4
==2913308== at 0x1091D2: main (badmemory.c:11)
==2913308== Address 0x4a43050 is 0 bytes after a block of size 16 alloc'd
==2913308== at 0x483877F: malloc (vg_replace_malloc.c:309)
==2913308== by 0x1091AA: main (badmemory.c:9)
...
==2913308== Invalid read of size 4
==2913308== at 0x10924E: main (badmemory.c:20)
==2913308== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==2913308== Process terminating with default action of signal 11 (SIGSEGV):
==2913308== dumping core
Debuggers

- There comes a day when `printf` just isn’t enough
- On that day you will start compiling with `-g` to turn on the debugger
- Then you will run `gdb myprog`, set some breakpoints, and get to the root of the problem
- Debuggers are covered in earlier CSCI courses (like CSCI 2021); refer to those materials to review / refresh [https://www-users.cs.umn.edu/~kauffman/2021/gdb](https://www-users.cs.umn.edu/~kauffman/2021/gdb)
Communicating Information to Programs

- Often programs need info from the outside world
  - What file to read/write, # of iterations to run, verbose/quiet output, report immediately, shutdown gracefully etc.
- A variety of mechanisms exist to convey such info to a program
  1. Command Line Arguments
  2. Environment Variables
  3. Signals
  4. Input/Output system calls and libraries
- Will now discuss 1 & 2 which are often used at program startup
- Alluded to Signals (#3) earlier (SIGKILL, SIGSTOP); Will discuss Signals in more detail later
- I/O calls (#4) will come soon (next lecture)
Exercise: Command Line Arguments

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

2-arg version of main() will be set up to have number of arguments and array of strings in it by whatever started it

> cat print13.c
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("%s\n",argv[1]);
    printf("%s\n",argv[3]);
}
> gcc -o mine print13.c
> ./mine -c 10 2.0
-c
2.0

Print Args

Write a quick C program which prints ALL of its argv elements as strings. Print a special message if an arg is string --verbose
Answers: Command Line Arguments

File: 04-process-environment-code/print_args.c

```c
1 // Print all the arguments in the argv array. Prints a special message
2 // if option is --verbose.
3
4 #include <stdio.h>
5 #include <string.h>
6
7 int main(int argc, char *argv[]){
8     printf("%d args received\n",argc);
9     for(int i=0; i<argc; i++){
10        printf("%d: %s\n",i,argv[i]);
11        if( strcmp(argv[i],"--verbose") == 0){
12           printf("Turning on VERBOSE output\n");
13        }
14     }
15     return 0;
16 }
```
All programs can access **environment** variables, name/value pairs used to communicate and alter behavior.

### Shell show/set variables

Done with `echo $VARNAME`

```
> echo $PAGER
less
> PAGER=cat
> echo $PAGER
cat
> echo "'$PS1'"
>'
> PS1='wicked$'
wicked$
> export x=1234  # in env
> y=5678    # not
```

### Shell `env`

Show _all_ environment

```
> env
JAVA8_HOME=/usr/lib/jvm/java-8-openjdk
PAGER=less
PWD=/home/kauffman/4061-F2017/lectures/04-process-environment-code
HOME=/home/kauffman
BROWSER=chromium
COLUMNS=79
MAIL=/var/spool/mail/kauffman
MANPATH=:/home/kauffman/local/man:/home/kauffman/local/usr/share/man::/man:/home/kauffman/local/man:/home/kauffman/local/usr/share/man::/man
PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/lib/jvm/default/bin:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/cor...
PS1=>
x=1234
...
Global variable `char **environ` provides array of environment variables in form `VARNAME=VALUE`, null terminated.

- NOT suggested to use `environ` directly,
- Instead use library functions `getenv()` / `setenv()` to check/change
C Library for Environment Vars

The C Library Provides standard library functions for manipulating environment variables.

#include <stdlib.h>

char *getenv(const char *name);
// returns pointer to value associated with name, NULL if not found

int setenv(const char *name, const char *value, int rewrite);
// sets name to value. If name already exists in the environment, then
// (a) if rewrite is nonzero, the existing definition for name is
// first removed; or (b) if rewrite is 0, an existing definition for
// name is not removed, name is not set to the new value, and no error
// occurs. return: 0 if OK, -1 on error

int unsetenv(const char *name);
// removes any definition of name. It is not an error if such a
// definition does not exist. return: 0 if OK, -1 on error

int putenv(char *str);
// str is of form NAME=VALUE, alters environment accordingly. If name
// already exists, its old definition is first removed. Don't use with
// stack strings. Returns: 0 if OK, nonzero on error.
Exercise: Manipulate Environment Vars

Write a short C program which behaves as indicated in the demo

▶ Prints ROCK and VOLUME environment variables

▶ If ROCK is set to anything, change VOLUME to “11”

Use these functions

```c
char *getenv(const char *name);
// NULL if name not set
// otherwise pointer to value

int setenv(const char *name,
           const char *value,
           int rewrite);
// Change name value pair,
// if rewrite is 1,
// overwrite previous definitions
```

Note the use of export to ensure child processes see the environment variables

```shell
> unset ROCK
> unset VOLUME
> gcc environment_vars.c
> a.out
ROCK not set
VOLUME is not set
> export VOLUME=7
> a.out
ROCK not set
VOLUME is 7
> export ROCK=yes
> a.out
ROCK is yes
Turning VOLUME to 11
VOLUME is 11
> echo $VOLUME
7
```

Note also that the program does not change the shell’s values for ROCK: no child can change a parent’s values (or mind)
Answers: Manipulate Environment Vars

See 04-process-environment-code/environment_vars.c

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    char *rock = getenv("ROCK");
    if(rock == NULL){
        printf("ROCK not set
");
    } else{
        printf("ROCK is %s
",rock);
        printf("Turning VOLUME to 11\n");
        int fail = setenv("VOLUME","11",1);
        if(fail){
            printf("Couldn't change VOLUME\n");
        }
    }
    char *volume = getenv("VOLUME");
    if(volume == NULL){
        volume = "not set";
    }
    printf("VOLUME is %s\n",volume);
    return 0;
} 
```