Recitation 00

C, parser, shredder

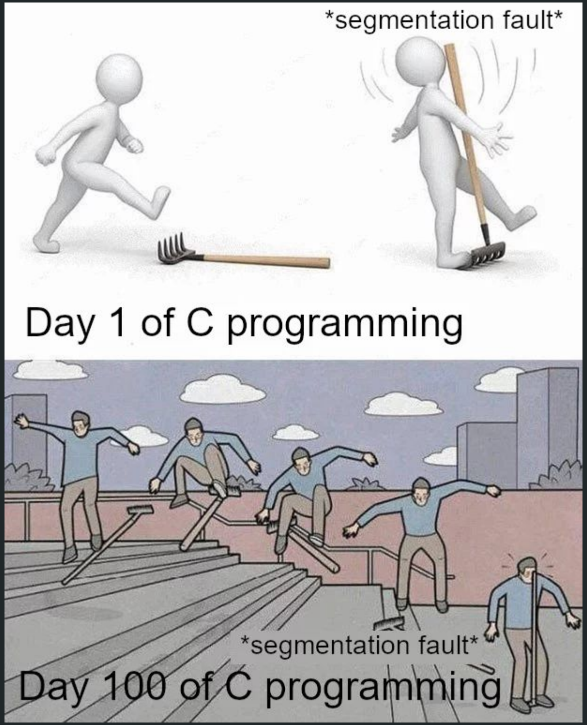


Table of Contents

1. Pointers
2. Memory management
3. Strings
4. penn-parser
5. fork, exec, wait, repeat!
6. Signal handling

Pointers

What is a pointer?

Pointers are another data type used frequently in C programming. Pointer variables store a memory address. Conceptually they “point” to a place in memory.

```
type * variable_name;
```

`type` is *any valid data type* in C and `*` indicates we are creating a pointer

Examples:

```
int * int_ptr;
```

```
struct parsed_command * command;
```

`char * text;` (Note: often referred to as C-strings, more later)

Pointer Operators

* Dereference operator: reads the address and “goes to” that location, accessing or modifying the data there

```
int x = *int_ptr; // reads data at address stored by int_ptr
 *int_ptr = x; // updates data at address stored by int_ptr
```

& Address of operator: gets the address of the operand, one way to assign pointers values

```
int y = 5;
int * int_ptr = &y;
```

Pointers, pointers, pointers!

Since we can create pointers to any valid data type, this means we can also create pointers to pointers! Or pointers to pointers to pointers! And so on :)

It follows that the dereference operator can be applied multiple times/chained.

This can be hard to conceptualize so we ***highly recommend*** drawing memory diagrams to help you visualize.

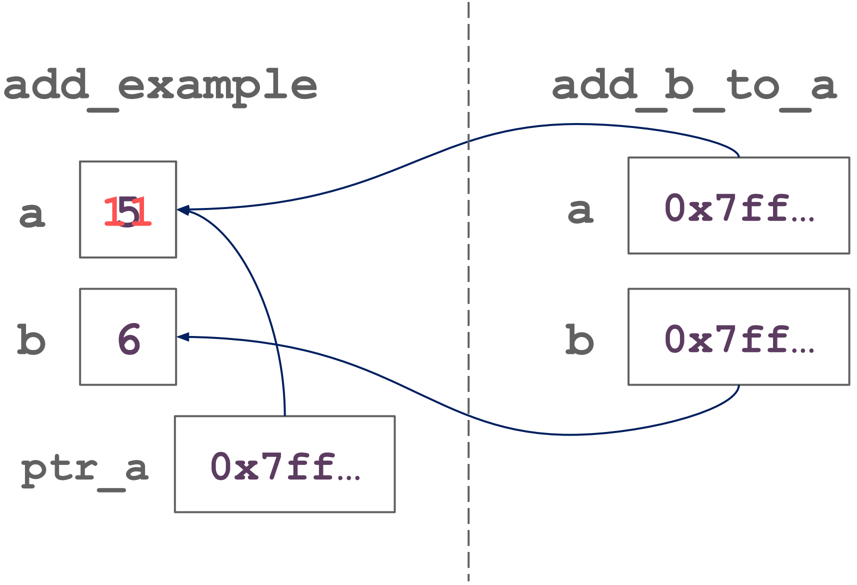
Example Code

```
void add_b_to_a(int * a, int * b) {
  *a = *a + *b;
}

void add_example() {
  int a = 5;
  printf(“The memory location of a
     is %p\n”, &a);

  int b = 6;
  int * ptr_a = &a;
  printf(“ptr_a value is %p\nptr_a
     dereferenced value is: %d\n”,
     ptr_a, *ptr_a);

  add_b_to_a(ptr_a, &b);
  printf(“a is %d\n”, a);
}
```



Output:

```
The memory location of a is 0x7fffffffd9a8
ptr_a value is 0x7fffffffd9a8
ptr_a dereferenced value is: 5
a is 11
```

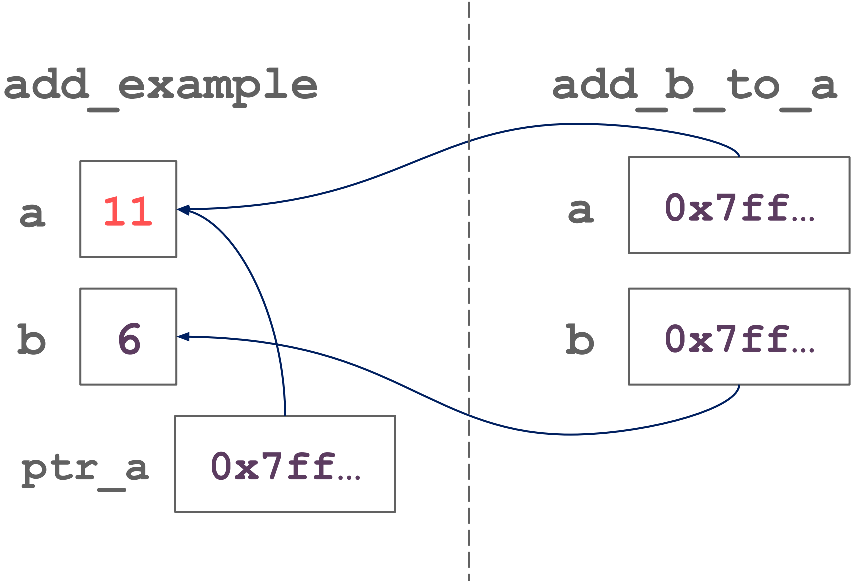
add_example

```
void add_b_to_a(int * a, int * b) {
  *a = *a + *b;
}

void add_example() {
  int a = 5;
  printf(“The memory location of a
    is %p\n”, &a);

  int b = 6;
  int * ptr_a = &a;
  printf(“ptr_a value is %p\nptr_a
    dereferenced value is: %d\n”,
    ptr_a, *ptr_a);

  add_b_to_a(ptr_a, &b);
  printf(“a is %d\n”, a);
}
```



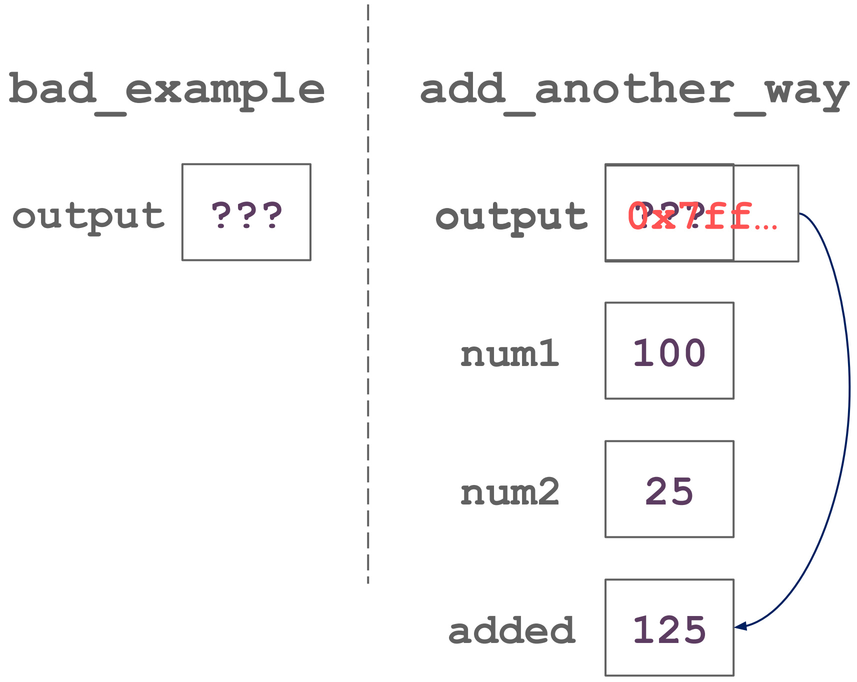
Output:

```
The memory location of a is 0x7ff...
ptr_a value is 0x7ff...
ptr_a dereferenced value is: 5
a is 11
```

bad_example

```
void add_another_way(int * output, int
      num1, int num2) {
   int added = num1 + num2;
   output = &added;
}

void bad_example() {
   int * output;
   add_another_way(output, 100, 25);
   printf(“output is: %d”, *output);
}
```

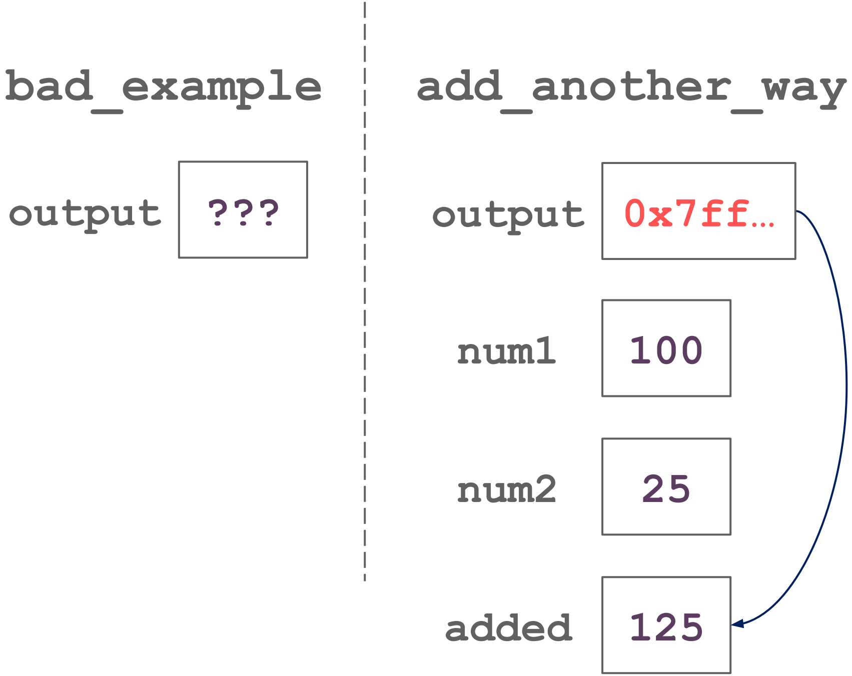


Output:
???

bad_example

```
void add_another_way(int * output, int
     num1, int num2) {
  int added = num1 + num2;
  output = &added;
}

void bad_example() {
  int * output;
  add_another_way(output, 100, 25);
  printf(“output is: %d”, *output);
}
```



bad_example

output ???

add_another_way

output 0x7ff…

num1 100

num2 25

added 125

Output:

???

What went wrong in this example?

We wanted **output** in the scope of **bad_example** to be set equal to **added**, but instead we only reassigned local variables in **add_another_way**. This is a common misuse of *output parameters*.

Fixed code:

```
void add_another_way(int * output, int num1, int num2) {
  int added = num1 + num2;
  *output = added;
}

void bad_example() {
  int output;
  add_another_way(&output, 100, 25);
  printf(“output is: %d”, *output);
```

Rule of Thumb for Output Parameters

When writing a method with output parameters:

- Dereference output parameter
- Assign value to pass out of method

```
void some_method(type * output) {

  // do some stuff

  *output = ...;

}
```

When calling a method with output parameters:

- Create variable for result; may initialize to a reasonable default value (i.e., NULL for a pointer type)
- In method call, pass address of variable to the output parameter

```
void another_method() {

  type result;

  some_method(&result);

}
```

Memory Management

The Stack and The Heap

The Stack

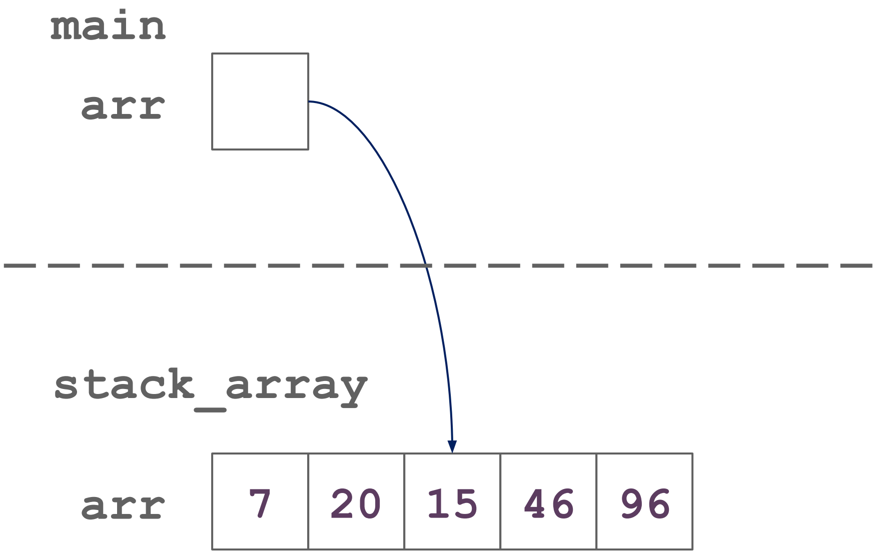
- Local scope (stack frames)
- Automatically allocated on function call
- Automatically deallocated on function return
- Limited size (stack overflow)

The Heap

- Program scope (dynamically allocated)
- Allocated with calls to `malloc`, `calloc`, `realloc`
- Deallocated with calls to `free`
- Large size

Local Scope Example

```
int * stack_array() {
  int arr[5] = {7, 20, 15, 46, 96};
  return arr;
}
int main(int argc, char ** argv) {
  int * arr = stack_array();
  printf(“arr[0]: %d\n”, arr[0]);
}
```



Unsafe memory usage! Should get a compiler warning like:

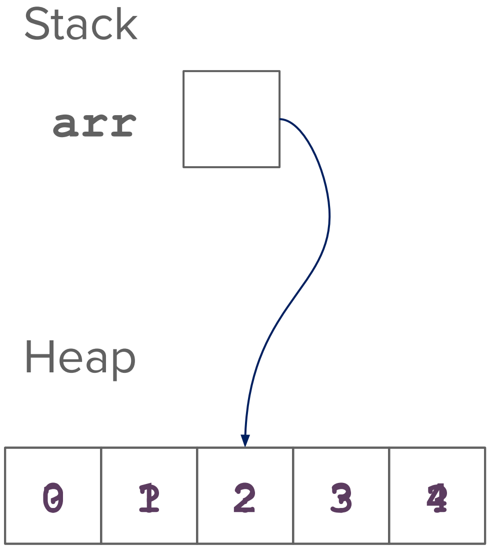
```
warning: address of stack memory associated with local
variable arr returned [-Wreturn-stack-address]
```

malloc Example

```
int main(int argc, char ** argv) {
  int * arr = (int *) malloc(5 * sizeof(int));
  printf(“arr[0]: %d\n”, arr[0]);

  for (int i = 0; i < 5; i++) {
    arr[i] = i;
  }

  for (int i = 0; i < 5; i++) {
    printf(“%d, ”, arr[i]);
  }
}
```



Line 2 produces a memory error for using uninitialized value(s)

Output: 0, 1, 2, 3, 4

Thinking Questions

How would we allocate memory for a struct?

How would we free such memory?

What we dynamically allocated memory for the fields of a struct?

Same as any other allocation/deallocation:

```
struct example * ex = (struct example *)
   malloc(sizeof(struct example));
```

```
free(ex);
```

If a field in our struct is dynamically allocated, you should free it before freeing the overall struct:

```
free(ex.allocated_field);
```

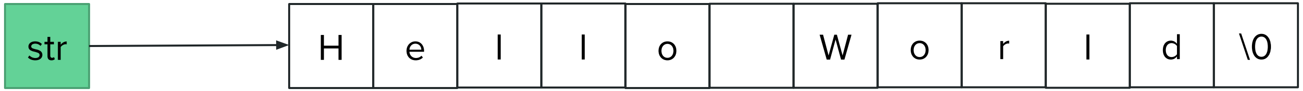
```
free(ex);
```

C Strings

String is a `char` array

```
char str[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ’, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’,
‘\0’};
```

```
char str[] = “Hello World”;
```



H	e	l	l	o		W	o	r	l	d	\0

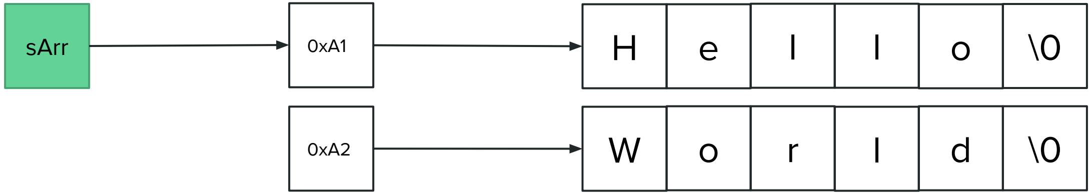
- By definition, string is NULL-terminated with the null character ‘\0’
- Null character tells C that this is the ‘end’ of the string
- A pointer to a string points to the first byte (character)

```
char* str = “Hello World”;
```

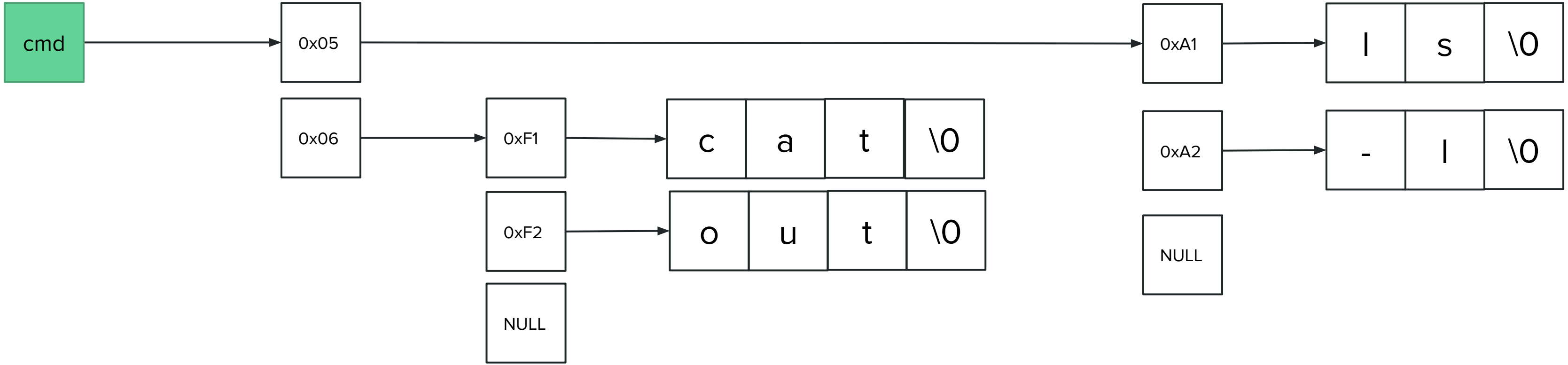
Strings and pointers

Note string in C: Null-terminated sequence of characters in memory

```
char** sArr = {“Hello”, “World”};
```



```
char*** cmd = { {“ls”, “-l”, NULL}, {“cat”, “out”, NULL} }
```



Penn-Parser

Tips on getting started

- Get the “easy” parses out of the way first
  - Background symbol “must” be at the end of the string
  - If the command string is empty, just return without doing anything
- Think about how you want to get rid of whitespace
  - strtok(3)
  - Pointer arithmetic?
- Think about how you want to separate each command on the pipe symbol ‘|’
- Read the man(ual) pages!
  - Search on google “[function] man”
  - On your docker terminal, type “man [function]”
  - realloc(3)

Points to Consider

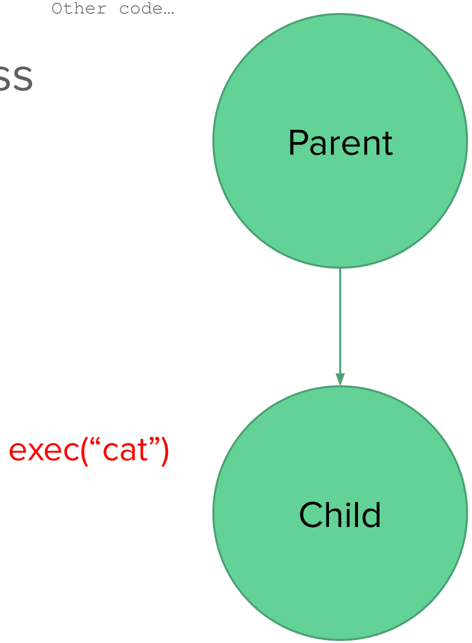
- If you are using strtok(3), how does it modify the original string?
- For every malloc(3), there MUST be a corresponding free(3)
    - Think of every case where you just exit(2) for error handling. Did you free all memory?
- How do you want to ‘dynamically’ allocate memory for the commands array?
    - Literally count
    - realloc(3)

fork(), exec(), wait(), repeat

Processes

- fork(2) system call creates an ‘identical’ child process
- exec(2) system call ‘replaces’ the child process with whatever the arguments to exec(2) are

```
fork();
for (int i = 0; i < 10; ++i) printf(“hi\n”);
wait(&status);
Other code…
```

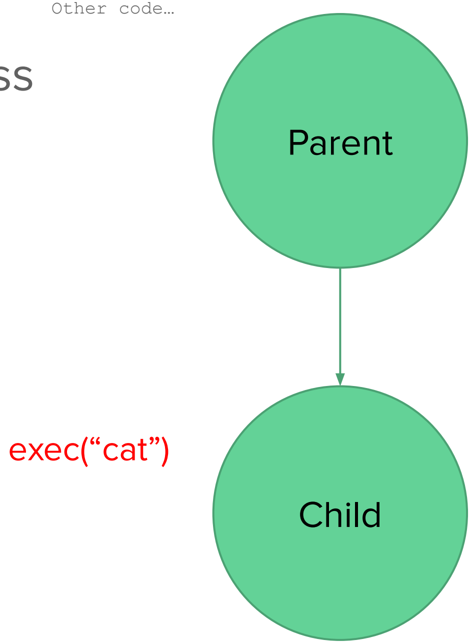


```
for (int i = 0; i < 10; ++i) printf(“hi\n”);
wait(&status);
Other code…
```

Processes

- fork(2) system call creates an ‘identical’ child process
- exec(2) system call ‘replaces’ the child process with whatever the arguments to exec(2) are

```
fork();
for (int i = 0; i < 10; ++i) printf(“hi\n”);
wait(&status);
Other code…
```

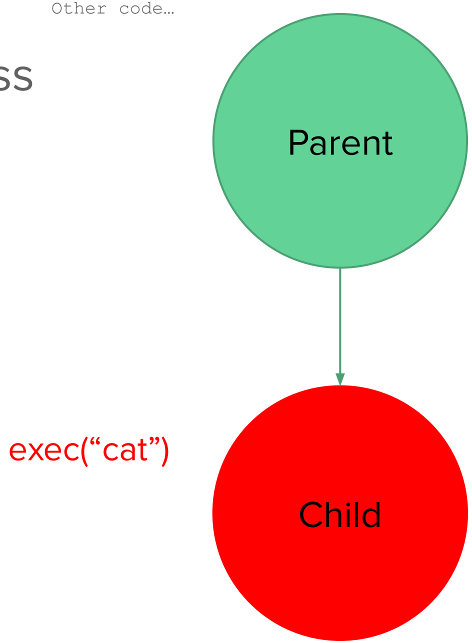


Processes

- fork(2) system call creates an ‘identical’ child process
- exec(2) system call ‘replaces’ the child process with whatever the arguments to exec(2) are

- Child exit(2) after exec(2)
- How does parent know when child exits?
  - wait(2)

```
fork();
for (int i = 0; i < 10; ++i) printf(“hi\n”);
wait(&status);
Other code…
```

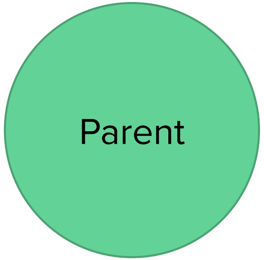


Processes

- fork(2) system call creates an ‘identical’ child process
- exec(2) system call ‘replaces’ the child process with whatever the arguments to exec(2) are

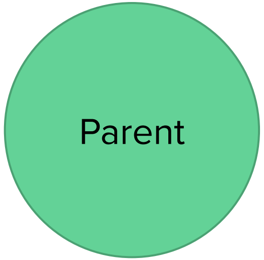
- Child exit(2) after exec(2)
- How does parent know when child exits?
    - wait(2)

```
fork();
for (int i = 0; i < 10; ++i) printf(“hi\n”);
wait(&status);
Other code…
```



Processes

```
fork();
for (int i = 0; i < 10; ++i) printf(“hi\n”);
wait(&status);
Other code…
```



- fork(2) system call creates an ‘identical’ child process
- exec(2) system call ‘replaces’ the child process with whatever the arguments to exec(2) are

- Child exit(2) after exec(2)
- How does parent know when child exits?
  - wait(2)

Penn-shredder basic flow

```
while(true) {
    write(prompt);
    char cmd_string[];
    int bytes = read(cmd_string);
    struct parsed_command cmd;
    parse_command(cmd_string, cmd);
    pid_t pid = fork();
    if (child){
        exec(cmd);
    }
    else {
        wait()
    }
}
```

Terminal

```
root# ./penn-shredder
```

Penn-shredder basic flow

```
while(true) {
    write(prompt);
    char cmd_string[];
    int bytes = read(cmd_string);
    struct parsed_command cmd;
    parse_command(cmd_string, cmd);
    pid_t pid = fork();
    if (child){
        exec(cmd);
    }
    else {
        wait()
    }
}
```

Terminal

```
root# ./penn-shredder
prompt>
```

Penn-shredder basic flow

```
while(true) {
    write(prompt);
    char cmd_string[];
    int bytes = read(cmd_string);
    struct parsed_command cmd;
    parse_command(cmd_string, cmd);
    pid_t pid = fork();
    if (child){
        exec(cmd);
    }
    else {
        wait()
    }
}
```

Terminal

```
root# ./penn-shredder
prompt> ls -l out.txt
```

Penn-shredder basic flow

```
while(true) {
    write(prompt);
    char cmd_string[];
    int bytes = read(cmd_string);
    struct parsed_command cmd;
    parse_command(cmd_string, cmd);
    pid_t pid = fork();
    if (child){
        exec(cmd);
    }
    else {
        wait()
    }
}
```

Terminal

```
root# ./penn-shredder
prompt> ls -l out.txt
```

Penn-shredder basic flow

```
while(true) {
    write(prompt);
    char cmd_string[];
    int bytes = read(cmd_string);
    struct parsed_command cmd;
    parse_command(cmd_string, cmd);
    pid_t pid = fork();
    if (child){
        exec(cmd);
    }
    else {
        wait()
    }
}
```

Terminal

```
root# ./penn-shredder
prompt> ls -l out.txt
-rw-r--r-- 1 root root 64 Jan 29
16:11 out.txt
```

Penn-shredder basic flow

```
while(true) {
    write(prompt);
    char cmd_string[];
    int bytes = read(cmd_string);
    struct parsed_command cmd;
    parse_command(cmd_string, cmd);
    pid_t pid = fork();
    if (child){
        exec(cmd);
    }
    else {
        wait()
    }
}
```

Terminal

```
root# ./penn-shredder
prompt> ls -l out.txt
-rw-r--r-- 1 root root 64 Jan 29 16:11
out.txt
prompt>
```

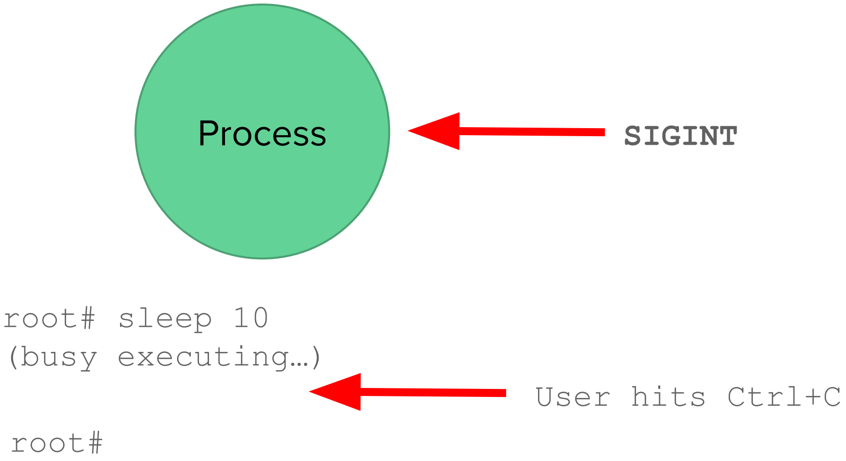
Some things to consider…

- What arguments does execve(2) take in?
  - int execve(const char *pathname, char *const _Nullable argv[], char *const _Nullable envp[]);
  - execve([cmd name], [array of command name and args/flags], {NULL});
  - How is this related to `struct parsed_command→commands` array?
- What is the difference between pressing enter after a command line and pressing Ctrl+D after a command line?
  - What character added at the end of command string when we press enter?
- What happens if we only press Ctrl+D on an empty line?
  - What is returned by read(2)?

Signal Handling

Terminology: Signals

- Asynchronous software notification to a process of an event
- “Software Interrupt” but can only be initiated by another process, not necessarily by the OS
- Simplest form of inter-process communication
- Each signal has a symbolic name
  - Starts with SIG*
  - Defined in <signals.h>



```
root# sleep 10
(busy executing…)

root#
```

Example signals

Useful for penn-shredder

Signal Name	Default Action	Description
SIGINT	Terminate	Terminal interrupt signal (Ctrl+C)
SIGKILL	Terminate	Immediate termination request
SIGALRM	Terminate	Terminate request after certain time
SIGTERM	Terminate	Graceful termination request
SIGSTOP	Stop	Stop process execution
SIGCONT	Continue	Continue process execution
SIGSEGV	Terminate (core dump)	Invalid memory access (segfault)
SIGUSER1 / SIGUSER2	Terminate	User defined signals
SIGFPE	Terminate (core dump)	Floating point exception

Signal Handlers

- We can define custom signal handler functions to alter behavior of signals
- Default behavior of signals ‘overwritten’ by the handler we attach using `signal(2)` system call
- Child processes also inherit these calls
- exec(2) system calls ‘reset’ the signal behaviors to DEFAULT behaviors

```
void handler(int signo) {
  if (signo == SIGINT) {
    printf(“Received SIGINT\n”);
  }
}

int main(void) {
  if (signal(SIGINT, handler) == SIGERR) {
    perror(“Unable to catch SIGINT”);
  }
  while(1);
  return 0;
}
```

Ctrl+C behavior of shredder

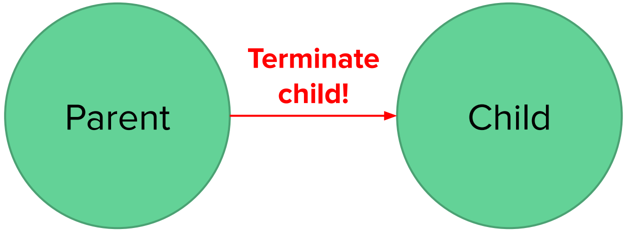
> “**Child processes** started from penn-shredder should respond to Ctrl-C by following their **normal behavior** on SIGINT, but **penn-shredder itself** should not exit (even when Ctrl-C is typed without a child process running). Instead, your shell must **catch SIGINT and re-prompt** (after printing a newline character if appropriate)”

```
$ penn-shredder
penn-shredder# /bin/cat
^C
penn-shredder# ^C
penn-shredder# sleep^C
penn-shredder# /bin/sleep 1
penn-shredder#
```

- Overwrite the behavior of SIGINT for shredder
- What can we do for the child?
  - Consider what the child code is doing and consider the changes (if any)
- What can we do to re-prompt?
  - Consider what happens at the start of each prompt-loop
  - Is there anything we should do in particular to reprompt?

Alarm

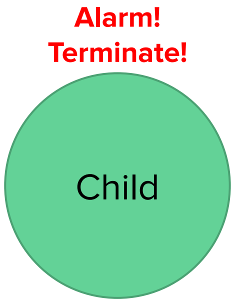
- penn-shredder takes in an extra argument [timeout]. Any child process execution that takes longer than [timeout] will be terminated
- How can we connect this to SIGALRM ?
  - Where can we set the alarm? Parent or the Child?
  - If parent, what can we do to handle SIGALRM's behavior when the parent code receives it?
    - Consider SIGKILL and kill(2) system call
  - If child, what can we do? (Extra Credit)



Parent

Terminate child!

Child



Alarm! Terminate!

Child

Any Questions?