Section 1

Wednesday, June 26, 2019 2:01 PM

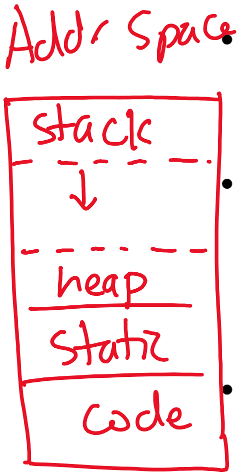
Section 1: Processes and Address Space

June 26, 2019

Contents

1 Vocabulary

- **process** - a process is an instance of a computer program that is being executed. It consists of an address space and one or more threads of control.



- **address space** - The address space for a process is the set of memory addresses that it can use. The address space for each process is private and cannot be accessed by other processes unless it is shared.

- **stack** - The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed.

- **heap** - The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. malloc, calloc free with free()

- **fork** - A C function that calls the fork syscall that creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process (except for a few details, read more in the man page). Both the newly created process and the parent process return from the call to fork. On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

int pid = fork() : pid > 0 succeeded, pid = child processes ID
pid = -1 failed, child not created
pid = 0 ⇒ inside child process

2 Warmup

2.1 Pointer and C Programming Practice

Write a function that places source inside of dest, starting at the offset position of dest. This is effectively swapping the tail-end of dest with the string contained in source (including the null terminator). Assume both are null-terminated and the programmer will never overflow dest. As a fun exercise to remember C tricks, try to see if you can shorten your code to as few lines as possible without using libraries or lines with multiple semicolons!

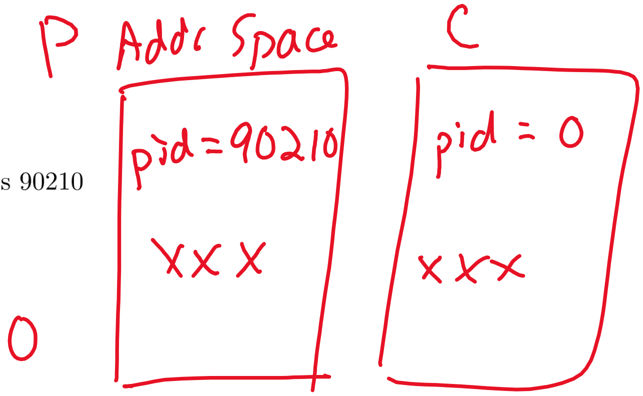
hello World \0 hi \0
↑ hel hi \0

```
void replace(char *dest, char *source, int offset)
{
    dest += offset;
    while ( *source != '\0' )
        *dest = *source;
        dest ++;
        source++;
    }
    *dest = *source;
}
```

while(*((dest++)+offset) = *source ++)

3 Problems

3.1 Hello World



What can C print in the below code? Assume the child's PID is 90210
(Hint: There is more than one correct answer)

```
int main() {
pid_t pid = fork();
printf("Hello World: %d\n", pid);
}
```

parent: 90210 Child: 0

Hello World: 0
" : 90210

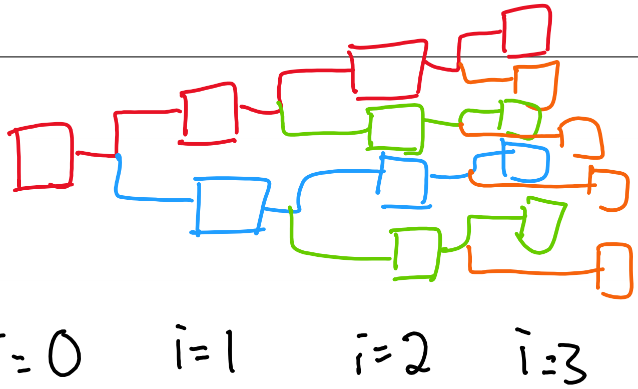
Hello World: 90210
" 0

Hello World: -1

3.2 Forks

How many new processes are created in the below program assuming calls to fork succeeds?

```
int main(void)
{
  for (int i = 0; i < 3; i++) {
      pid_t pid = fork();
  }
}
```

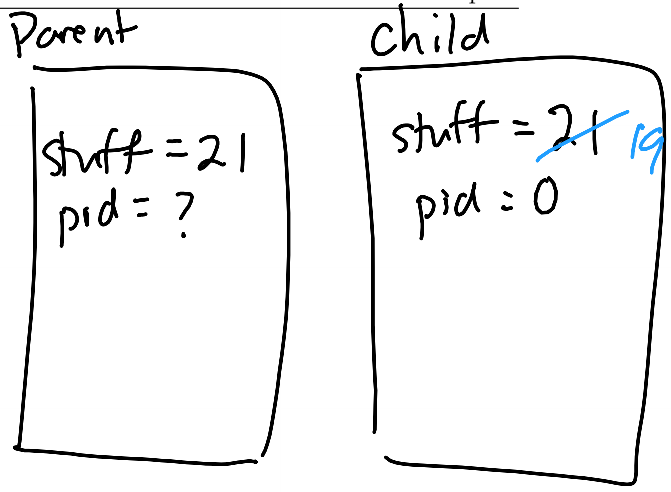


8. 7 created

3.3 Stack Allocation

What can C print?

```
int main(void)
{
    int stuff = 21;
    pid_t pid = fork();
    printf("9 plus 10 equals %d\n", stuff);
    if (pid == 0)
        stuff = 19;
}
```



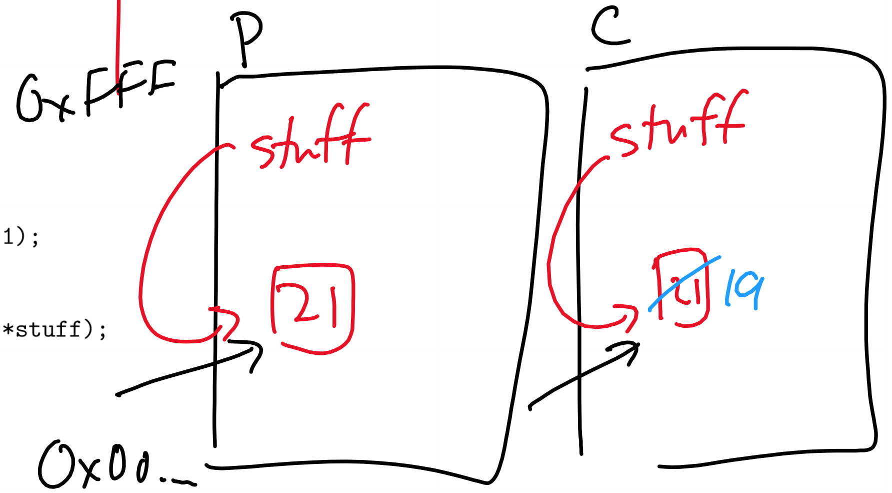
1) 9 plus 10 equals 21 9 plus 10 equals 21

'' 21

3.4 Heap Allocation

What can C print?

```
int main(void)
{
    int* stuff = malloc(sizeof(int)*1);
    *stuff = 21;
    pid_t pid = fork();
    printf("9 plus 10 equals %d\n", *stuff);
    if (pid == 0)
        *stuff = 19
}
```



Same as above

3.5 Slightly More Complex Heap Allocation

What does C print in this case ?

```
void printTenNumbers(int *arr)
{
    int i;
    printf("\n");
    for(i=0; i<10; i++) {
        printf("%d",arr[i]);
    }
    exit(0);
}

int main()
{
    int *arr, i;
    arr = (int *) malloc (sizeof(int));

    arr[0] = 0;
    for(i=1; i<10; i++) {
        arr = (int *) realloc( arr, (i+1) * sizeof(int));
        arr[i] = i;
        if (i == 7) {
            pid_t pid = fork();
            if (pid == 0) {
                printTenNumbers(arr);
            }
        }
    }
    printTenNumbers(arr);
}
```

Parent: 0 - 9
Child: 0 - 7 (possible segfault)