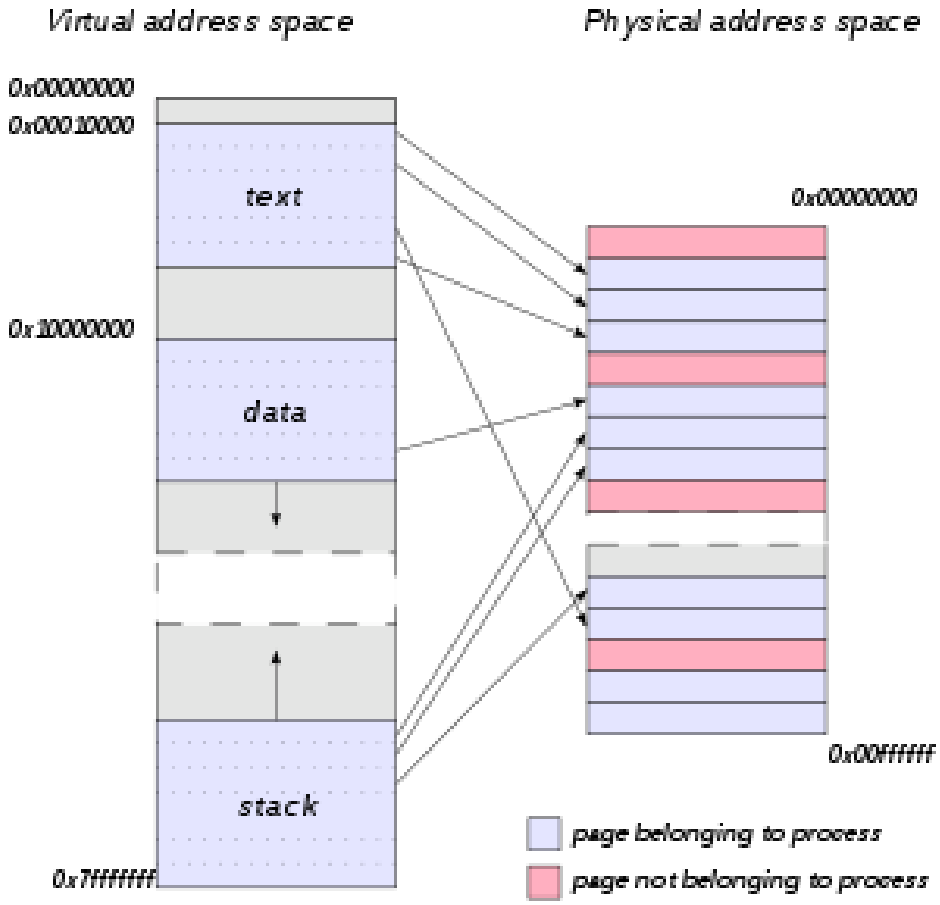
Pointers and addresses– what happens after fork()

```
*p = 14;
if (!fork()) {
        printf("CHILD: before changing the variable:\n");
        printf("address of p is %p and value is %d\n", (void*)&p, *p);
        *p = 25;
        printf("CHILD: after changing the variable:\n");
        printf("address of p is %p and value is %d\n", (void*)&p, *p);
        printf(" CHILD: exiting\n");
        exit(0);
} else {
        printf("PARENT: as is\n");
        printf("address of p is %p and value is %d\n", (void*)&p, *p);
        wait(NULL);
        printf("PARENT: after child exited\n");
        printf("address of p is %p and value is %d\n", (void*)&p, *p);
}
```

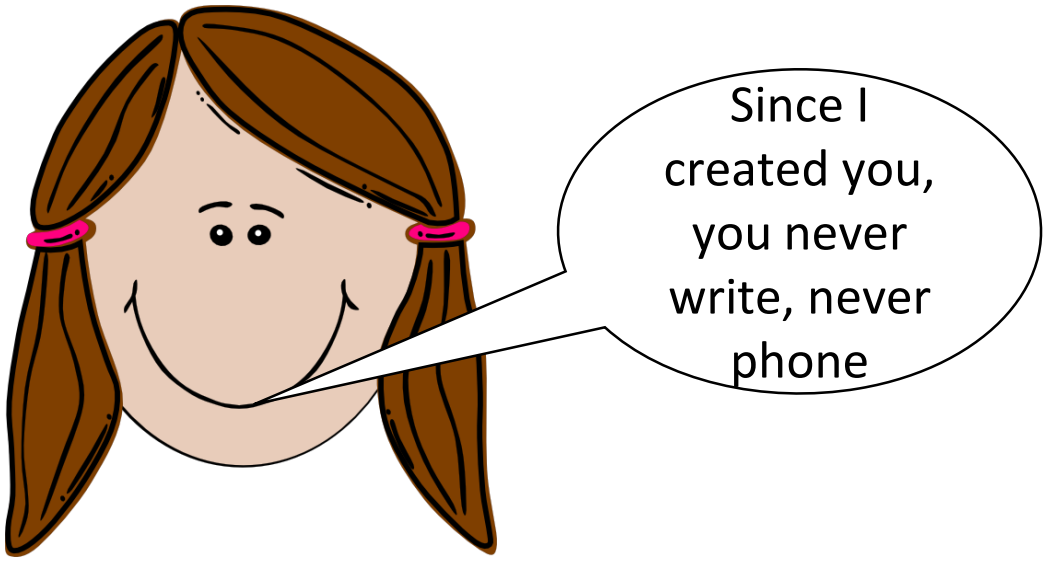
fork_test.c

Explanation: virtual address space

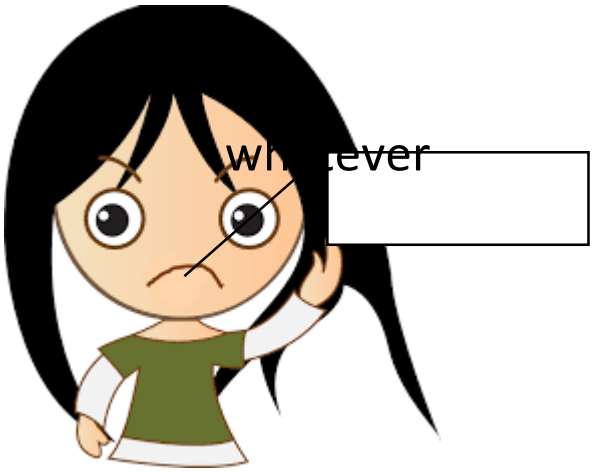
- Each process stores mapping from a virtual address to an actual physical memory address
- After fork() this virtual memory address is marked as read only
- So when child tries to change it, a new piece of physical memory is allocated – cannot modify read-only memory
- Now in child process virtual address is the same, but points to a different memory location



Staying in touch with your child



Parent process



Child process

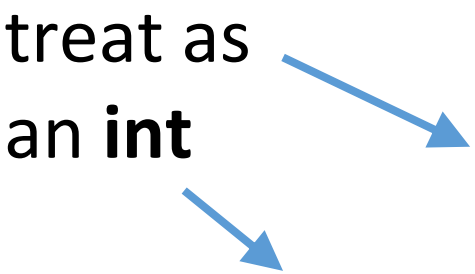
We need inter-process communication

Inter-process communication

- Wait for exit status (report when done)
- Pipe (always open for communication)
- Signals (send when you want, handle or ignore)
- Sockets (open connection with the world)

Fork and wait

Lecture 04.02




```
pid_t fork()
pid_t wait(int *status)
void exit(int status)

WIFEXITED(status)
WEXITSTATUS(status)
```

Is there something wrong with this code?

```
// fork a child and then in the parent do
int status;
wait(status);
```

Is there something wrong with this code?

```
// fork a child and then in the parent do
int *status;
wait(status);
```

Is there something wrong with this code?

```
// fork a child and then in the parent do
int status;
wait(&status);

printf("My child returned %d\n", status);
```

Doing it the right way

```
int status;
wait(&status);

if WIFEXITED(status) {
  printf("My child returned %d\n",
    WEXITSTATUS(status));
}
```

Example: fork_wait.c

```
int child_status;
wait (&child_status);
if (WIFEXITED (child_status))
      printf ("the child process exited normally,
            with exit code %d\n",  WEXITSTATUS (child_status));
else
      printf ("the child process exited abnormally\n");
```

Exercise

- Write a program that forks one child for each command line argument.
- The child computes the length of the command line argument and exits with that integer as the return value.
- The parent sums these return codes and reports the total length of all the command line arguments.

Solution: 1/4
declare any variables you need

```
int i, result;
int total_len =0; //to store total_len_of_args
```

Solution 2/4:
loop over command-line arguments and fork

```
for (i = 1; i < argc; i++) {
        int result = fork();
```

Solution 3/4: inside for loop

```
if (result < 0) {   // case: a system call error
        // handle the error          exit(1);
} else if (result == 0) {  // case: a child process
        int len = strlen(argv[i]);
        exit (len); //status returned is the length
 } else {
        // in the parent but before doing the next loop iteration
        // wait until a child terminates
        int ret_status;
        wait (&ret_status);
        total_len += WEXITSTATUS(ret_status);
}
```

Solution 4/4: outside for loop

```
// Only the parent gets here

printf("The length of all the args is %d\n", total_len);
```

Inter-process communication

- Wait for exit status (report when done)
- Pipe (always open for communication)
- Signals (send when you want, handle or ignore)
- Sockets (open connection with the world)