

CS333: Operating Systems Lab

Autumn 2022

Lab Quiz 2 (20 marks)

- **No internet. No phone. No friends** (strictly for the duration of the exam only!)
 - Test cases are in the `testcases` folder.
Note that different test cases may be used for grading.
 - This lab quiz has a total of 3 questions. You can attempt them in any order of your choice.
-

Q1. Implement the `waitpid` system call (6 marks)

Implement the following version of the `waitpid` system call.

A parent process waits on its child process which is specified by a `pid`;

the caller also specifies if the call is blocking or non-blocking. If the call is non-blocking, then the system call should return even if the child process is not exited.

The function signature for the system call is as follows:

```
int waitpid(int pid, int blocking)
```

The parameter `pid` specifies **pid** of the child process and `blocking` takes a value of 1 if the call is blocking, and 0 if it is non-blocking.

The function should ...

- return the `pid` of the child process if only if it is in the ZOMBIE state.
- return 0 if the call is non-blocking and the child process is not in the ZOMBIE state
parent can recheck status at a later point or default orphan processing take effect
- with a blocking call, block till the child process transitions to a ZOMBIE state and return `pid` of the child process
- return -1 in all other cases

Implementation Notes:

- An implementation of the `wait` system call exists in xv6, and can be the starting point for the `waitpid` implementation — a copy of the code for `wait` as a good starter for implementation of `waitpid`.
- `ptable` the list of PCBs maintained and used by xv6 for process metadata use a lock called the `ptable.lock` to protect against concurrent updates.
For the sake of this problem, observe that lock is acquired just before iterating over the `ptable`, and released before returning from the function. If you add other points of return in your function, make sure you release the lock before return.

Testing: Following are three test cases/programs for this question —

1. A test program `testcase-waitpid-1.c` tests your implementation for blocking calls of `waitpid`. The expected output should look like the one shown below:

```
$ testcase-waitpid-1
Parent process started
Child process started
Child process going to sleep for 3 seconds..
Child process woke up. Exiting..
Blocking Call 1: Valid Child process reaped by parent
Blocking Call 2: Returned -1 as expected
```

2. A test program `testcase-waitpid-2.c` tests your implementation for non-blocking calls of `waitpid`. The expected output should look like the one shown below:

```
$ testcase-waitpid-2
Parent process started
Non-Blocking Call 1: Returned 0 as expected
Parent process going to sleep for 4 seconds..
Child process started
Child process going to sleep for 3 seconds..
Child process woke up. Exiting..
Non-Blocking Call 2: Valid Child process reaped by parent
Non-Blocking Call 3: Returned -1 as expected
```

3. A test program `testcase-waitpid-3.c` tests both blocking non-blocking calls of `waitpid`. The expected output should look like the one shown below:

```
$ testcase-waitpid-3
Parent process started
Child process 1 started
Child process 1 going to sleep for 3 seconds..
Child process 1 woke up. Exiting..
Blocking Call 1: Valid Child process reaped by parent
Blocking Call 2: Returned -1 as expected
Non-Blocking Call 1: Returned 0 as expected
Parent process going to sleep for 4 seconds..
Child process 2 started
Child process 2 going to sleep for 3 seconds..
Child process 2 woke up. Exiting..
Non-Blocking Call 2: Valid Child process reaped by parent
Non-Blocking Call 3: Returned -1 as expected
```

Note that statements in your output need not appear in the same order as they appear above.

The functionalities being tested in each of the test cases carry partial credit.

Grading Note: Marks will be **deducted** from a positive score for this problem if a value of -1 is not returned by the system call for the appropriate situations.

Q2. List all the descendants of a process (8 marks)

Implement a system call to fetch the list of the **pids** of all the descendants of a process with a given `pid`. The `pids` are supposed to be populated in an array passed as an argument to the call. The function signature for the system call is as follows:

```
int getDescendants(int pid, int maxDescendants, int *descendants)
```

The system call implementation should...

- populate the `descendants` array with the pids of the descendants of the process with id `pid`, and
- return the total number of descendants.

The parameter `maxDescendants` is the maximum size of the `descendants` array.

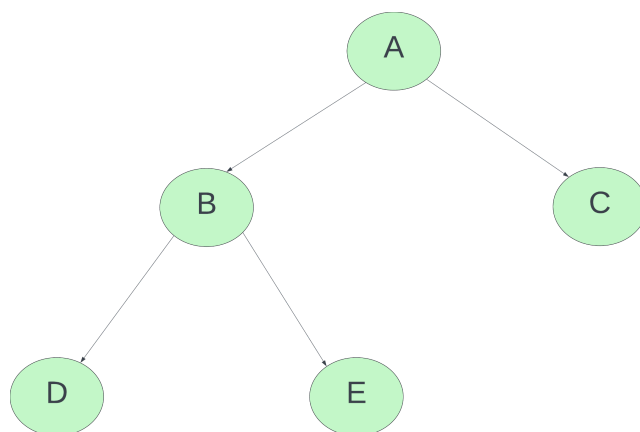
Assumptions:

1. Here, the definition of the descendants of a given process includes the process itself.
2. You are allowed to fill the `descendants` array in any arbitrary order, except that the `pids` of all the descendants must be present in the positions with indices in the range `[0, total-1]`, where `total` is the total number of descendants of the process under consideration.

Note that `total` is also the return value of the system call.

3. Return -1 if a process with the given `pid` does not exist.

E.g., consider the diagram below showing the parent-child relationship of processes— process A forks two children, B and C. B in turn forks two children of its own, D and E.



- The 'leaf processes' C, D and E have 1 descendant each (the descendant of a leaf process is the process itself)

- Process B has 3 descendants — B, D, and E

- Process A has 5 descendants — A, B, C, D, and E

Testing: Following are the three test cases/programs for this question —

1. A test program `testcase-getdescendants-1.c` tests the count of descendants returned for a tree of depth 1 (1 parent and its children).

```
$ testcase-getdescendants-1
Parent process with pid 3 started: Will have 4 descendants in total
Parent process going to sleep for 1 second..
Child process 1 with pid 4 started: Will have 1 descendants in total
Child process 2 with pid 5 started: Will have 1 descendants in total
Child process 3 with pid 6 started: Will have 1 descendants in total
getDescendants() for process with pid 3 returned 4
```

2. A test program `testcase-getdescendants-2.c` tests both the count and the list of descendants returned for a tree of depth 1 (parent and its children).

```
$ testcase-getdescendants-2
Parent process with pid 3 started: Will have 4 descendants in total
Parent process going to sleep for 1 second..
Child process 1 with pid 4 started: Will have 1 descendants in total
Child process 2 with pid 5 started: Will have 1 descendants in total
Child process 3 with pid 6 started: Will have 1 descendants in total
getDescendants() for process with pid 3 returned 4
PIDs of the descendants of process with pid 3 are: 3 4 5 6
```

3. A test program `testcase-getdescendants-3.c` tests both the count and the list of descendants returned by your implementation for a tree of depth greater than 1.

```
$ testcase-getdescendants-3
Parent process with pid 5 started: Will have 5 descendants in total
Parent process going to sleep for 1 second..
Child process 1 with pid 6 started: Will have 3 descendants in total
Child process 1_1 with pid 8 started: Will have 1 descendants in total
Child process 1_2 with pid 9 started: Will have 1 descendants in total
Child process 2 with pid 7 started: Will have 1 descendants in total
getDescendants() for process with pid 6 returned 3
PIDs of the descendants of process with pid 6 are: 6 8 9
getDescendants() for process with pid 5 returned 5
PIDs of the descendants of process with pid 5 are: 5 6 7 8 9
```

Note:

- The statements in your output need not appear in the same order as they appear above.
- The `pids` being printed in the above examples need not match with the `pids` in the output of your implementation.
- The `pids` listed by your implementation are not verified for their correctness. You can verify them manually.

The functionalities being tested in each of the test cases carry partial credit.

Grading Note: Marks will be **deducted** from a positive score for this problem if a value of -1 is not returned by the system call for the appropriate situations.

Q3. Implement a `heapSize` system call. (6 marks)

Implement a system call to return the total amount of heap memory allocated for a process.

The specification of the call is as follows: `int heapSize()`

The system call returns the **size** of the heap at the point of the call for the calling process. You can assume that the total heap size will never be negative at any point of the test case programs and only `sbrk` system calls will be made by the user program (no `malloc` or `free` system calls).

Hint: The heap allocation call `sbrk(int n)` is the entry point (of the system call) for heap allocations for a process.

Testing: Test cases have been provided inside the folder `testcases/q3`.

The expected output for each test case is given below.

1. `q3testcase1.c`
a program to check the system call with no `sbrk()` calls from the user process.
2. `q3testcase2.c`
a program to check the system call with one `sbrk()` call from the user process to grow the heap.
3. `q3testcase3.c`
a program to check the system call with multiple `sbrk()` calls from the user process to grow the heap.
4. `q3testcase4.c`
a program to check the system call with multiple `sbrk()` calls from the user process to grow and shrink the heap.

```
$ q3testcase1
Heap Size till now: 0 bytes
$
```

```
$ q3testcase2
Heap Size till now: 0 bytes
sbrk return address: 0x3000
Heap Size till now: 1024 bytes
$
```

```
$ q3testcase3
Heap Size till now: 0 bytes
sbrk return address: 0x3000
Heap Size till now: 1024 bytes
sbrk return address: 0x3400
Heap Size till now: 5120 bytes
sbrk return address: 0x4400
Heap Size till now: 6144 bytes
sbrk return address: 0x4800
Heap Size till now: 8192 bytes
sbrk return address: 0x5000
Heap Size till now: 12288 bytes
$
```

```
$ q3testcase4
Heap Size till now: 0 bytes
sbrk return address: 0x3000
Heap Size till now: 1024 bytes
sbrk return address: 0x3400
Heap Size till now: 5120 bytes
sbrk return address: 0x4400
Heap Size till now: 5120 bytes
sbrk return address: 0x4400
Heap Size till now: 4096 bytes
sbrk return address: 0x4000
Heap Size till now: 2048 bytes
$
```

The functionalities being tested in each of the test cases carry partial credit.

Submission Instructions

- All submissions via moodle. Name your submissions as:
`<rollnumber>_labquiz2.tar.gz`
- Auxiliary files are available in the `auxiliary_files` folder.
- The tar should contain the following files in the specified directory structure:

```
<rollnumber>_labquiz2/  
|__proc.h  
|__proc.c  
|__defs.h  
|__syscall.h  
|__syscall.c  
|__sysproc.c  
|__user.h  
|__usys.S  
|__ <any other source files that your change/add to xv6>  
|__Makefile
```

please adhere to this format strictly

- Shell command to tar your submission directory ...
`tar -zcvf <rollnumber>_labquiz2.tar.gz <rollnumber>_labquiz2`
- **Due date: 12th September 2022, 5.15 pm**