Jacob Little

ECE 565 (001)

Instructor: Dr. Michela Becchi

Project 1 Report

Introduction

This report discusses the implementation of three features to the Xinu operating system as well as answers to the questions presented in the project specification.

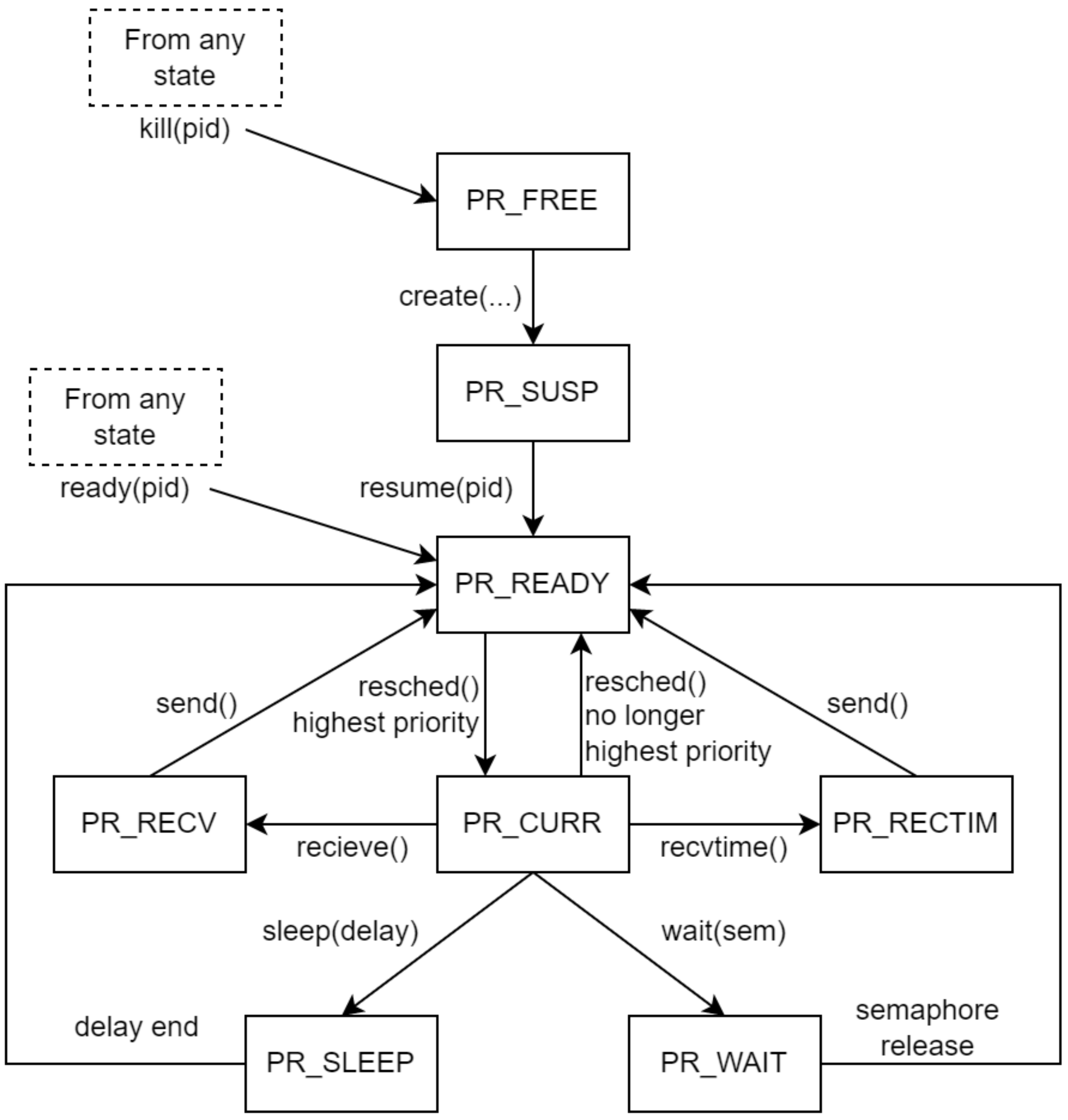
Question Answers

Q1. The maximum number of processes accepted by Xinu is 100 by default. This value is defined by `NPROC` in **config/Configuration**. The value defined in **config/Configuration** will overwrite the value defined in **config/conf.h** which will in turn overwrite the value of 8 set by **include/process.h**.

Q2. An inline definition in **include/process.h** called `isbadpid()` defines the criteria for an illegal (or "bad") PID. A PID is illegal if it is less than zero, greater than or equal to the maximum number of processes, or if it corresponds to a process table entry that is not currently being used (e.g. its state is `PR_FREE`).

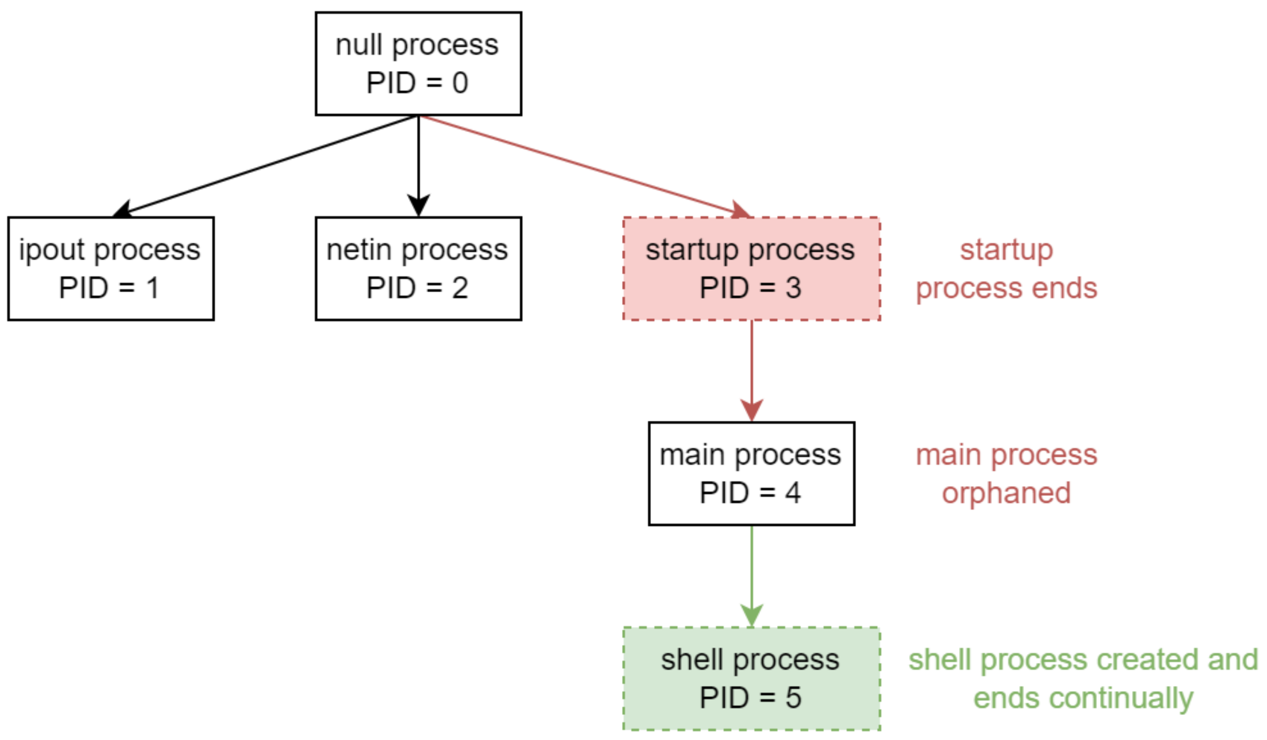
Q3. The default stack size for a process is 65536 bytes which is defined by `INITSTK` in **include/process.h**.

Q4. Process state diagram:



Q5. The shell process is created in **system/main.c** during the main process. It is created once first before the main process enters into a loop and continuously waits for the shell process to end so that it can recreate the shell process and start the cycle anew.

Q6. Process tree immediately after initialization:



Q7. `receive()` stalls the parent process until its child process ends execution. Without this, the parent and the child would execute concurrently and the output would be unpredictable.

P1. Timing

This problem involved modifying:

- **include/**
    - > **process.h** - Added time that process began execution to the process table entry data structure
    - > **clock.h** - Added `ctr1000` so that it could be used by other files
- **system/**
    - > **clkhandler.c** - Increment ctr1000 every 1 millisecond so that it can be used to track process time elapsed in milliseconds
    - > **create.c** - Capture value of ctr1000 at process creation and store it in its process table entry
    - > **initialize.c** - Capture value of ctr1000 at null process creation and store it in its process table entry
- **shell/**
    - > **xsh_ps.c** - Added additional column to ps command printout that shows time since process began execution

In order to keep time as accurate as possible, the value of `ctr1000` is captured at the beginning of the `xsh_ps` function and then used later on to calculate the difference between process creation (which is stored in the process table) and the captured time from `ctr1000`.

P2. Process Creation and Stack Handling

This problem involved modifying:

- **include/**
  - > **prototypes.h** - Added `syncprintf()` so that it could be used for debugging in **system/fork.c**. No instances of `syncprintf()` have been left in the final version of **system/fork.c**
- **system/**
  - > **fork.c** - Implemented process fork by copying and modifying **system/create.c**

`fork()` begins by setting up the process table entry for the child similarly to how it's done in `create()`. It then copies the parent stack into the child stack and recursively modifies the stack frame base pointers to point to the child instead of the parent by adding the offset between the two stacks to each stack frame base pointer in the child stack.

Finally, the address of the currently executing instruction (EIP) is stored into a pointer `child_eip`. This address is where the child begins execution. If the PID of the currently executing process is different than the parent PID, then the child must be executing so it can skip to the end of the function and return. Otherwise, the currently executing process must be the parent so it can continue setting up the partial context switch similarly to how it was done with `create()`.

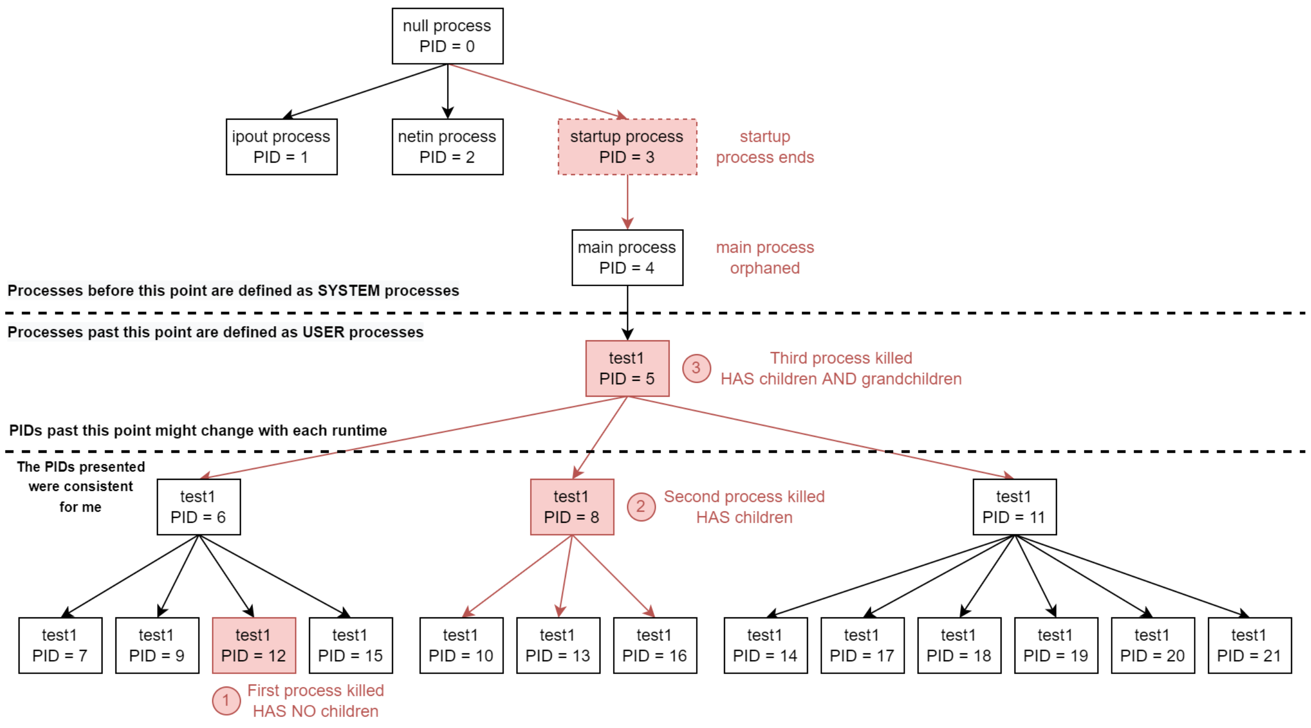
P3. Cascading Termination

This problem involved modifying:

- **include/**
  - > **process.h** - Added `user_process` flag for differentiating USER and SYSTEM processes
- **system/**
  - > **create.c** - Set the `user_process` flag to USER if parent is the main process. Otherwise set `user_process` flag the same as parent
  - > **fork.c** - Set the `user_process` flag to SYSTEM so that it does not conflict with cascading termination
  - > **initialize.c** - Set the `user_process` flag to SYSTEM for the null process
  - > **kill.c** - If process to be killed is a USER process, check if it has any children. If so, call the kill function again on each of those child processes
  - > **main.term** - Create a test case to demonstrate cascading termination by spawning a process tree and outputting the process tree before and after each termination

The test case spawns a process tree of 17 processes. First, a process with no children is killed, then a process with some children is killed, and last a process with children and grandchildren is killed for demonstration.

Process tree before killing:


null process PID = 0
ipout process PID = 1
netin process PID = 2
startup process PID = 3
startup process ends
main process PID = 4
main process orphaned
Processes before this point are defined as SYSTEM processes
Processes past this point are defined as USER processes
test1 PID = 5
3
Third process killed HAS children AND grandchildren
PIDs past this point might change with each runtime
The PIDs presented were consistent for me
test1 PID = 6
test1 PID = 8
2
Second process killed HAS children
test1 PID = 11
test1 PID = 7
test1 PID = 9
test1 PID = 12
test1 PID = 15
test1 PID = 10
test1 PID = 13
test1 PID = 16
test1 PID = 14
test1 PID = 17
test1 PID = 18
test1 PID = 19
test1 PID = 20
test1 PID = 21
1
First process killed HAS NO children