Software Requirements Specification

for

PID Manager

Version 3.0 approved

Prepared by Ricky Chon

CISC 3320 - Operating Systems

September 18th, 2020

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Ricky Chon	10/01/2020	Adding support for multi-threading	2.0
Ricky Chon	10/28/2020	Adding mutex locking to PID manager to protect against race conditions	3.0

1. Introduction

1.1 Purpose

The goal is to implement a basic PID manager that is capable of allocating and releasing PIDs from a PID map. It is for the purpose of learning how PID managers work in operating systems (Linux-based for this particular scope of the software).

The next iteration of this software supports multi-threading, which will allow the user to allocate and release PIDs on multiple threads.

The third iteration of this software implements mutex locking, to prevent race conditions and protects shared resources, such as the values in a PID map.

1.2 Document Conventions

This document follows the IEEE standard for software requirements specification.

1.3 Intended Audience and Reading Suggestions

Developers and project managers are the intended audience of this document. It is recommended to read this document from top to bottom, to go from more generic descriptions to more specific implementations.

1.4 Product Scope

The software is intended to show how a basic PID manager works, and allow developers to learn how to implement the basic functionalities of a PID manager. It is also used to demonstrate the PID manager’s support for multi-threading and how it is implemented. There are also tests to demonstrate mutex locking and the speed of processes with and without it (mutex is faster than without since it eliminates unnecessary access to shared resources).

1.5 References

2. Overall Description

2.1 Product Perspective

This product is a standalone PID manager, but should be built with scalability in mind for future iterations.

2.2 Product Functions

The PID manager must be able to do the following:
- Allocate a map for representing PIDs
- Allocate PIDs
- Release PIDs

An API should be available for the user to call functions to perform the above actions.

2.3 User Classes and Characteristics

Student Developers: other students that have this same assignment; they should be able to learn how to install the development tools to run the software, if they don't have them already

Faculty Members: professors that view this assignment and test out the software; more experience and technical expertise, should have the necessary development tools to run the software

2.4 Operating Environment

Any of the major operating systems with gcc/g++ support, as well as support for GNU Make: Windows 10, Mac OS X, Linux. Due to the vast operating environment, multi-threading support should be implemented using the newer C++11 `<thread>` library, as opposed to just the UNIX/Linux `<pthread.h>` or Windows' `<windows.h>`. The `<mutex>` library should also be used for the same reasons.

2.5 Design and Implementation Constraints

Since the goal is to create software that would be used in an operating system (PID manager), it is recommended to write this software in C/C++, due to its high performance. The current iteration of this software uses threads and future iterations may include optimizations, so it would be best to utilize C libraries. The software's code should conform to C++17 standards.

2.6 User Documentation

Provide a README markdown file containing details and instructions on running the PID manager for end-users, and an INSTRUCTIONS markdown file containing implementation instructions for developers. Developers can use this document alongside the instructions file for reference.

2.7 Assumptions and Dependencies

For the purpose of this assignment, assume that the software will run on any of the major operating systems with gcc/g++ support, as well as support for GNU Make: Windows 10, Mac OS X, Linux

3. External Interface Requirements

3.1 User Interfaces

A standard UNIX terminal, or any terminal that is able to run gcc/g++ and GNU Make would suffice as a user interface to test out the PID manager's features. The software will take in user input to select which tests to run. There is also an option for the user to run the executable with a specific test number as an argument to skip the prompts.

3.2 Hardware Interfaces

Any decent computer, laptop, remote server that can run GNU Make and compile C++ code.

3.3 Software Interfaces

One strategy the software can adopt is the way Linux handles the availability of process IDs, by having a bitmap where a value of **0** at position **i** would indicate an available process ID, and a value of **1** would indicate a process ID that is currently in use.

3.4 Communications Interfaces

4. System Features

4.1 API - Map Allocation

4.1.1 Description and Priority
- Creates and initializes a data structure for representing PIDs
- High Priority

4.1.2 Stimulus/Response Sequences

- User calls the function `int allocate_map()`
- If map allocation is successful, return 1
- If map allocation is unsuccessful, return -1

4.1.3 Functional Requirements

- REQ-1: range of possible PID values is between 300 and 5000 inclusive
    - `#define MIN_PID 300`
    - `#define MAX_PID 5000`
    - Since we are dealing with a bitmap of integers, an array of integers of fixed size equal to the difference of `MAX_PID` and `MIN_PID` is a better option for performance compared to a dynamic vector of integers

4.2 API - PID Allocation

4.2.1 Description and Priority

- Allocates and returns a PID
- High Priority

4.2.2 Stimulus/Response Sequences

- User calls the function `int allocate_pid()`
- If unable to allocate a PID (when all PIDs have been allocated), return -1
- If PID allocation is successful, return the PID

4.2.3 Functional Requirements

- REQ-1: range of possible PID values is between 300 and 5000 inclusive

4.3 API - PID Release

4.3.1 Description and Priority

- Releases a PID
- High Priority

4.3.2 Stimulus/Response Sequences

- User calls the function `void release_pid(int pid)`
- If the PID is valid, release it (set its value from **1** to **0**)
- If the PID is out of range of `MAX_PID` and `MIN_PID`, do nothing

4.3.3 Functional Requirements

- REQ-1: range of possible PID values is between 300 and 5000 inclusive

4.4 Optional Diagnostics Tool

4.4.1 Description and Priority

- A tool that outputs useful messages based on two parameters: a test value and the action conducted (Map Allocation, PID Allocation, PID Release)
- Medium Priority

4.4.2 Stimulus/Response Sequences

- User calls the function `void output_result(PidAction action, int testValue)`
- If the action is PidAllocation (enum), the function will test if the test value is -1
- If the testValue is -1, print an error letting the user know that PID allocation has been unsuccessful, then return with exit code 1
- Otherwise, print a message letting the user know that PID allocation has been successful, along with the PID number that was allocated
- Similar sequences should occur with the other actions

4.4.3 Functional Requirements

- REQ-1: the above features must be correctly implemented for the tool to display correct results, since all it does is test the parameters/return values of each function

4.5 Multi-Threading Support

4.5.1 Description and Priority

- Ability to create and run individual processes on threads concurrently irregardless of the operating environment
- Medium Priority

4.5.2 Stimulus/Response Sequences (Multi-Thread Testing)

- Initialize an array of threads
- Allocate PID map for use
- Create and execute the following process on each thread
    - Request PID
    - Sleep for random amount of time
    - Release PID
- Synchronize and join all threads

4.5.3 Functional Requirements

- REQ-1: reliable thread library for the specified language used (in this case C/C++)
- REQ-2: threads can't seem to execute functions from an external class, so both the process and thread should ideally be located in the same file

4.6 Mutex Locking

4.6.1 Description and Priority

- Protect PID map from race conditions by using mutex locks
- Medium Priority

4.6.2 Stimulus/Response Sequences (Mutex lock tests)

- Counting to 1k without mutex
    - Initialize an array of threads
    - Each thread executes a function that will count to 1k and print each number
    - Print elapsed time
- Counting to 1k with mutex
    - Initialize an array of threads

- Each thread executes a function that will count to 1k and print each number
  - Acquire lock
  - Increment count
  - Release lock
- Print elapsed time (should be shorter than test without mutex)

4.6.3 Functional Requirements

- REQ-1: reliable mutex library for the specified language used (in this case C/C++)

5. Other Nonfunctional Requirements

5.1 Performance Requirements

5.2 Safety Requirements

5.3 Security Requirements

5.4 Software Quality Attributes

Maintainability: easy to understand, repair, and optimize without any breaking changes
Scalability: easy to add new features or deprecate old ones
Reliability: make sure the implemented features always work as intended
Testability: easy to test features without extra hassle
Ease of use: easy for developers to pick up and use
Well documented: the software API should be well documented to eliminate any confusion on each feature

5.5 Business Rules

6. Other Requirements

The software should be licensed under the MIT License.

Appendix A: Glossary

API: Application Programming Interface
IEEE: Institute of Electrical and Electronics Engineers
PID: Process ID

Appendix B: Analysis Models

Appendix C: To Be Determined List