

M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics  
Faculty of Electrical Engineering and Informatics  
Department of Telecommunications and Media Informatics

# Reliability analysis of process operations

Author  
BALÁZS VALYON

Budapest, 2014.

---

Consultants: DR.TAMÁS MAROSITS, BME Dept. of Telecomm. and Media Inf.  
DR.ÁBEL SINKOVICS, Morgan Stanley

# Nyilatkozat

Alulírott Valyon Balázs, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, December, 2014.

# Kivonat

Napjainkban számítógépek és szerverek ezreinek távoli menedzselése már hétköznapi feladat. Ennek jelentősége és gyakorisága feltehetőleg a közeljövőben úgy nő meg, ahogyan a cloud-technológia terjed.

Bármely programnak a leállítása az egyik legfontosabb alkalmazása a távoli hozzáférésnek. Ha az alkalmazás csak egyetlen egy folyamatból áll, akkor ez a művelet általában fennakadás nélkül végrehajtható. De ha ezek az alkalmazások amelyeket felügyelni és vezérelni szeretnénk, összetettek, tehát bonyolult folyamatfákat hozhatnak létre a futtatásuk során, továbbá ha ezek valamilyen ön- és rokon-felügyelő algoritmust is implementáltak, akkor ezeknek a folyamatoknak és fáknek a leállítása problémássá válhat.

A dolgozatomban megvizsgálom a fenti problémára a Windows és UNIX-alapú operációs rendszereken használt megoldásokat. Ismertetem, hogy a platformok milyen eszközöket biztosítanak az önálló folyamatok vagy az összetett folyamatfák kezelésére. Ezen eszközök megbízhatóságának méréséhez számos tesztalkalmazást készítettem el különböző nyelveken, úgy mint Java, C# valamint Python.

A mérés után összehasonlítom az eszközök és megoldások képességeit valamint megbízhatóságát, továbbá bemutatom a gyenge pontjaikat.

# Abstract

Nowadays the remote administration of thousands of computers and servers is an everyday task. Over and above, the importance and frequency of these actions will definitely increase in the near future as the cloud-technology spreads.

The termination of any application is one of the most important part of the remote management. If the application consist of only one process, this action usually can be done without any disruption. But if those applications which have to be monitored and controlled are composite, which means that they can create a complex process tree during their execution, furthermore if these have some self- and relative-monitoring algorithm implemented, the termination of these processes and trees can be really difficult.

In my thesis I examine both the Windows and UNIX-based operating system solutions for this problem. I will go around what kind of tools these platforms have to manage single processes and complex process trees. To measure the reliability of these tools, I created several test applications on different languages such as Java, C# and Python.

After the measurement, I compare the tools and solutions capabilities and reliability, and highlights their weaknesses.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Terminology . . . . .	6
1.2	UNIX concept . . . . .	7
1.3	Windows concept . . . . .	8
1.4	Concept of the problem . . . . .	8
1.5	Iterating through process tree nodes . . . . .	9
1.5.1	Iteration on UNIX-based systems . . . . .	9
1.5.2	Iteration on Windows systems . . . . .	10
1.5.3	Conclusion . . . . .	11
1.6	UNIX solutions . . . . .	11
1.6.1	Process groups . . . . .	12
1.7	Windows solutions . . . . .	12
1.7.1	Job Objects and Nested Jobs . . . . .	13
1.7.2	Windows API and .NET . . . . .	13
<b>2</b>	<b>Available tools</b>	<b>15</b>
2.1	Process inspector, monitoring tools . . . . .	15
2.1.1	Windows: Sysinternals Process Explorer . . . . .	15
2.1.2	Windows: YAPM (Yet Another Process Monitor) . . . . .	16
2.1.3	UNIX: Glances . . . . .	17
2.1.4	UNIX: Bluepill . . . . .	17
2.2	Summary of monitoring tools . . . . .	18
<b>3</b>	<b>Measurement</b>	<b>19</b>
3.1	The environment . . . . .	19
3.2	My test applications . . . . .	19
3.3	Testable process tree creation . . . . .	20
3.4	Simple process tree creator . . . . .	20
3.5	Two Process application, two-direction monitoring . . . . .	21

3.5.1	Why Python and not Java . . . . .	21
3.6	Job Objects, the Windows platform only process tester . . . . .	22
3.7	Testing methods . . . . .	23
3.8	Test the Process tree creator . . . . .	23
3.8.1	Test on Windows with Sysinternals Process Explorer . . . . .	23
3.8.2	Test on Windows with YAMP . . . . .	28
3.8.3	Test on UNIX with Glances and bash commands . . . . .	30
3.8.4	Test on UNIX with Bluepill . . . . .	34
3.9	Two Process application, two-direction monitoring . . . . .	35
3.9.1	Windows: kill the child . . . . .	35
3.9.2	Windows: kill the root . . . . .	35
3.9.3	UNIX: kill the child . . . . .	36
3.9.4	UNIX: kill the root . . . . .	36
3.9.5	Testing Windows Job Objects . . . . .	37
<b>4</b>	<b>Conclusion</b>	<b>38</b>
4.1	Only the API of the Operating System . . . . .	38
4.2	Extended mechanisms . . . . .	38
4.2.1	Process group commands . . . . .	39
4.2.2	Job Object commands . . . . .	39
4.3	Future work . . . . .	39
<b>5</b>	<b>Summary</b>	<b>40</b>
5.1	Summary . . . . .	40

# Chapter 1

## Introduction

### 1.1 Terminology

Before I present the main topic of this thesis in details, I would like to introduce a few terms, which I will refer to back later.

The main focus will be on the desktop and server UNIX and Microsoft Windows operating system platforms, so only those expressions will be presented, which are connected to these. Embedded and mobile operating system versions will not be examined in this dissertation.

When an application is started automatically or manually, the operating system will *create* a process and start it [1]. The process is an entity, basically an executed program [2]. The processes are separated, isolated from each other, all of them have an isolated part of the memory (RAM), which cannot be reached by any other process except the owner. They have full access to the memory, usually through a memory manager abstraction layer, which is provided by the operating system. This abstraction layer hides the physical addresses of the memory, the processes only see a virtual memory space. Because of this abstraction, the processes are totally isolated from each other, mainly for security reasons, they cannot reach each other's physical memory spaces.

Inside a process, there can be found one or more *threads* [3]. There is always one thread at least, which is called the main thread. This can create and start additional threads. A process can be imagined as a container, which contains elements which executes the program code. While the processes are separated, the threads share the same memory space, so they can reach and modify each other's data, this is the topic of mutual exclusions. As a process can start multiple threads, also could launch other processes. In this case, the main process is called the *root* or the *root process*, and the new ones are called *children* or *child processes*.

For example, an application called A, downloads some data from a network source, than this should be processed, but only a dedicated independent application called B, is capable of do it. So, the A application has to start the B, to process the downloaded data. But, the B application relies on other smaller applications, such as D, E and F, which are responsible for some subtasks. Meanwhile the root (A) application starts another application, which has a different task. With these steps, a *process tree* will be constructed by the operating system.

All of the operating systems operation are based on this concept. During the boot process, the first root process is started, which launches all of the other processes that make possible even for the user, to use his or her applications [4]. Because of this, every process has an *ancestor*, except the main root process. In this concept, a very complex and high tree can be constructed.

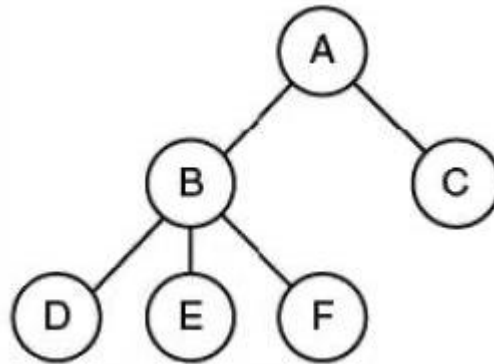


Figure 1.1: Process tree with a root(A) and child processes

To handle these complex constructs, the processes and also the threads have a few unique identifiers. With these, not only one specific process can be controlled, but many of them, which have at least one similar parameter, such as each process has a unique *process identifier* (PID). Also, the processes inherit the process identifier of their ancestor, which is called *parent process id* (PPID). The PID and PPID are important in the iteration of these trees. Furthermore, the processes have a state, which usually represents that a process is being executed or suspended. From this point, there are differences in the process handling between the operating systems.

## 1.2 UNIX concept

Processes in the UNIX based operating systems have a *process group identifier* (PGID) [5] which processes on Windows do not have [6]. The processes can be put in specific process



groups for the easier handling, for example if a process group has ten processes in it, if I want to terminate all of them, I only have to execute the termination command once, not ten times in this case. This is called *process grouping*. Furthermore, if an application start several child processes which starts grandchild processes, all of them will have the same process group id, which their root (parent) process has, so they inherit the process group id by default. Because of this concept, all processes which are connected to the same application can be controlled, such as terminated. Many process groups can be managed if they are put into something bigger container, with the *signals* [7]. With a signal, one or more connected process groups can execute the same command at the same time (if the resources can handle) [8].

### 1.3 Windows concept

The Microsoft Windows platform does not provide higher abstraction managing options for the process handling out of the box, but can be implemented if it is necessary. If we need this kind of managing options later, Microsoft made available an API (Application Programming Interface), which does almost the same as the UNIX process groups and signals, which is called the Job Objects. In addition, they developed an even more powerful version of it recently, the Nested Jobs. Furthermore, if we do not want to use these API-s, we can reach almost the same functionality with some tricks, but it can be more complicated.

### 1.4 Concept of the problem

Nowadays, when everything is about the cloud and remote technologies, it is a very common task, to control few thousand devices from a distant location. During the execution of these tasks, it is significant to know with high reliability, if a command is perfectly performed, or there is some chance that it has failed. If there is a chance, we have to know the probability rate of the occurrence of this event according to the used technologies.

The most critical part of these remote commands are the termination of processes. According to the used platforms and technologies, there are several different methods to control the processes and perform actions on them. The importance of these actions are momentous, because these are usually done automatically by scripts on schedule on hundreds of servers remotely, in some cases from a different part of the world. If the command was not executed successfully, it can result in unexpected high costs, performance load and incorrect behavior. Such as, when the system believes that the application and all of its processes are terminated and even so one or more are still running, it can ruin other applications runtime.

The main problem is, that complex process trees can be created and when the terminating scripts are executed, they may not terminate all nodes of these trees. Furthermore, special processes can be created, which are monitoring the others in the tree, and when one of them is stopped, they can restart it. These complex self-monitoring applications usually cannot be stopped easily, sometimes nodes remain orphans and keep running after the execution of the termination command. The administrators have to be sure, if a script or a process monitoring application can stop these entire process trees or not. In this paper I will examine this problem.

Furthermore crash-data studies concluded [9] that in most software crashes, not the operating system was responsible for it. Because of this it can be said, that the applications can fail because of its own poor, untested implementation or because of other software fails which pulls it with it. For all of these circumstances that would be good if an application or the framework can restart the application or at least, the system can send a reliable signal about the crash of these applications [10].

## 1.5 Iterating through process tree nodes

Both the Windows and UNIX-based operating systems have a different solutions for iterating through the nodes of process trees and execute operations. The algorithm is almost the same, every process on both platforms store its parent process id, so we can go through all of the processes step by step until we reach the wanted node of the tree. Which means we can perform operations on that specific process, such as set its parameters or terminate it.

### 1.5.1 Iteration on UNIX-based systems

On the UNIX-based platforms to get this functionality, we mainly use two commands which gets the process id of a process which is based on the `getpid()` [11] method of the API, and a different one which gets the parent process id of that specific process which is the `getppid()` [12] method. These are C language based APIs. To use these APIs, we should list the running processes, choose one, and then get its parent process id.

For example, the shell command to get the process ID of the running bash, one should run the following:

```
ps aux | grep bash
```

which will return a few parameters of the running bash process, where the first will be the process ID, in my example, this looks like this:

```
valyonb+ 9962 0.0 0.1 26816 3760 pts/0 Ss 12:30 0:00 bash
```

where the process ID is 9962. So, to get the parent process ID of this, one should execute the following command:

```
ps -f 9962
```

which returns:

```
UID          PID  PPID  C  STIME TTY  STAT  TIME  CMD
valyonb+ 9962 9953  0  12:30 pts/0  Ss   0:00  bash
```

or, execute this:

```
ps -o ppid= 9962
```

which will return only the 9953 parent process ID only, as the other command did before.

The algorithm which iterates through the tree is mainly based on only just these two commands.

### 1.5.2 Iteration on Windows systems

The Windows' approach is almost the same, just the API's name and structure are a little different, but the logic is the same.

To get a process ID we should call the `GetProcessId` [13] Windows API functions, which will return an integer, which is the ID. To get the parent process ID, we should use the process handles, which contain a few parameters about a specific process, such as the needed parent id [14] or the startup time info of the process. If we would like to work with these identifiers in our own applications's source code, according to which language we use, such as C++, we can use the `kernel32.dll` for most cases, but in C# we can get this this functionality more easily with the `ManagementObject` [15] class of the .NET framework.

As the UNIX-based systems have the `ps` command, the Windows platforms have `Tasklist` [16] and `Wmic` [17] (Windows Management Instrumentation Command-line interface).

`Tasklist` can provide the basic information about every process running in the system, and the `Wmic` can give back some specific ones. For example, if one is interested in to get the parent process ID of the running notepad application, first one should check the process ID of the notepad then according to that, get the parent process ID with `Wmic`, almost the same steps as were in UNIX, just the commands are different.

If just using Tasklist, it will list every process, so in this way one has to find it manually, or I can be more specific and filter to the notepad.

The basic and the more specific commands are the following:

```
tasklist
```

or:

```
tasklist /fi "imagename eq notepad.exe"
```

the more specific one will get this result back:

Image Name	PID	Session Name	Session	Mem Usage
notepad.exe	7312	Console	1	5 720 K

In this case, the process ID of my notepad is 7312. After that, one can run the following Wmic command, to get the parent ID of this process:

```
wmic process get processid,parentprocessid,executablepath|find "7312"
```

where the result is in reverse order:

```
C:\Windows\system32\notepad.exe 15852 7312
```

which is correct, because notepad was started from the command prompt, and the command prompt's process ID was 15852.

### 1.5.3 Conclusion

Both platforms provide tools by default to check process trees and iterate through them, and have third party applications which are using mainly these APIs listed before, but provide more specific and powerful features with them.

The main differences between the platforms escalate when we try to manage the processes of the tree, or just part of the trees such as a branch.

## 1.6 UNIX solutions

The most common solution for handling complete process trees or just a part of them is the process group. Processes can be assigned to a group, and operations can be executed on them. The tricky part is, when the other processes have some checking mechanism implemented

that interfere with the commands during execution. For example, if the processes of a process group are monitoring the run of each other in a infinite loop, and if one of them detects that another is terminated for somekind of reason, it can restart that process. Because of this mechanism in this example, the termination of the tree cannot be executed with a single API call.

### 1.6.1 Process groups

The simplest solution is, when we apply a command to a process group, such as termination. If we use process groups, every process in the group will be terminated, the root, every children and grandchildren and so on. This can be done, because every child process inherits the process group id from its ancestor.

For example, to kill [18] a process group which has the ID 1989, one can write the following commands, they lead to the same result but use a different algorithm:

```
kill -GPID -1989  
kill -TERM -1989
```

These two commands can be written in a different way, because the “GPID” and “TERM” strings has a code, which means the same, like the following way:

```
kill -9 -1989  
kill -- -1989
```

There is one significant limitation [19]. If we would like to assign one of our applications to a specific process group, we cannot do it easily. The first problem is, the UNIX `setpgid()` [20] API cannot be reached from the bash (terminal). If someone wants to use it, he or she has to write a small C program that reaches that API. Furthermore the other limitation is, a process can only set the group ID of only itself or its children processes, and if one of its child already called an execution function, its group id cannot be changed later by its ancestor either.

One useable technique is, if our application creates a process tree, during the creation it sets the group id from the source code [21], so the commands can be executed on them and no group assignment is needed later, which can be very tough to be achieved.

## 1.7 Windows solutions

The Windows platform does not have anything by default like the process group term as in UNIX. There are more than one solutions that are similar, but the developer has to use them manually.

### 1.7.1 Job Objects and Nested Jobs

This API is called Job Objects, and I quote the exact definition from Microsoft, which is: *”Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on a job object affect all processes associated with the job object”* [22]. The processes can be assigned to these *Job Objects*, and through these objects, a sort of commands can be send to them to execute, such as the stop command, which will terminate all of the processes which were assigned to that specific Job Object. There is one limitation, which is if a process is already assigned to a job object, it cannot be assigned to another, and a process tree can contain only one Job Object. Recently with Windows 8 and Windows Server 2012 Microsoft released the newer version of it, the *Nested Jobs* [23], which can manage a process tree that has more than one Job Object. On Figure 1.2 [49] there can be seen a Nested Jobs structure.

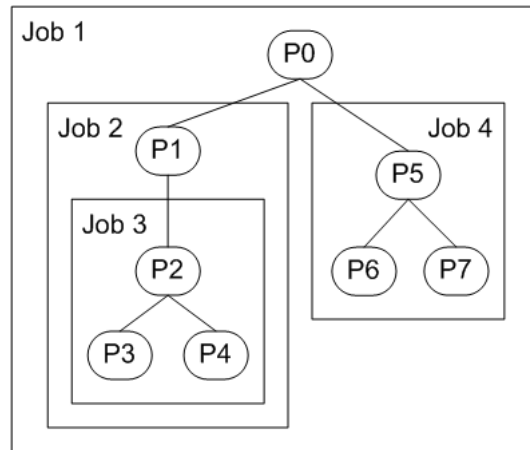


Figure 1.2: Nested Jobs process tree

Nested Jobs makes it possible to control differently specific branches of the process tree. We can create hierarchical Jobs as it is represented on the picture.

### 1.7.2 Windows API and .NET

If we do not want to use any Job Objects, we can get an almost similar process handling with some tricks. Not only the processes on the UNIX based systems inherit their ancestors parent process id but the ones on the Windows systems as well. So, if we use the *System.Diagnostics* and the *System.Management* namespace from the .NET framework [24], we can manage whole process trees. We can get only some levels of the process tree, or with a recursive algorithm all of the nodes of the tree. During the execution of the algorithm, any kind of action can be done [25], such as the termination of some or all processes in the tree [26].

A demonstration program can be found here [50], which is able to iterate through a process tree, it takes the process ID of the root element as argument, and the similar command can be executed on every node. For example, if we want to stop all of the processes in a tree with a termination command or just get the same parameters from every one of them, the same command can be executed on every node during the iteration. This algorithm uses the Process namespace from the Diagnostics, the *ManagementObjectSearcher* and *ManagementObjectCollection* [27] classes from the *Management* namespace. If the command is to abort the process, it will be executed on every process in the tree. Windows offers the TaskKill API out of box, but it will not certainly kill every child and grandchild of the process, so orphan or zombie processes can remain and lead the system into an unexpected state.

The conclusion is, that on the Windows platform, there is no solution for handling a complete process tree out of the box, but there are a few different methods, which can achieve that. With the new Nested Jobs, Microsoft offers a more sophisticated solution for this kind of process tree handling than the process group in the UNIX-based systems, but it also has to be implemented by the developer.

## Chapter 2

# Available tools

This chapter presents tools used for the measurements. I did not find one single tool which is platform-independent and can be used on both platforms, so I chose different ones on Windows and UNIX. They have the same basic functionality but the special features are variant, also they use distinct algorithms and operating system APIs to execute a command. Because of this, the same command with different mechanisms and solutions has been tested, and the differences between them are highlighted.

### 2.1 Process inspector, monitoring tools

Both Windows and UNIX platforms have a few very good process inspector, system monitoring and statistical tools, with lots of similar features but they are implemented in different ways. Because of this design and implementation difference, the reliability and effectiveness of them are tested on both platforms. Furthermore, a lots of features can be reached from the UNIX shell, so in some cases, I will use it instead of a specific tool.

#### 2.1.1 Windows: Sysinternals Process Explorer

Microsoft provides the Process Explorer software by Mark Russinovich [28], which is an excellent tool to monitor active processes. I use this tool for two purpose, first to monitor my testing applications to check how they perform, and second, if I execute a command such as kill a process or kill a process tree using this tool, how the applications react. For example, will it detect that a process has stopped and will it restart that? Will the tool be able to stop my applications, kill the whole process tree or not?



### 2.1.2 Windows: YAPM (Yet Another Process Monitor)

YAPM [29] is an open source application monitoring tool for Windows platform based on the .NET framework. This gets a large amount of data about every application and running process, and can execute the usual operations as well. It uses not just the .NET framework, but the native Windows API to gather statistical information.

This tool is more powerful than the Sysinternals Process Explorer, but it has weaknesses as well. For example, while the Process Explorer by default shows the process tree, YAPM only lists the processes, which can be seen on Figure 2.1. The Process Explorer highlights the levels of the process tree, such as the root process with 16196 process ID is on the highest level and its children are on the second level with 15036 and 18532 process ID. YAPM has a few useful features that Process Explorer does not offer, such as the handling of Jobs (Job Objects), adding a process to a Job, or manually choose which method to kill a process with.

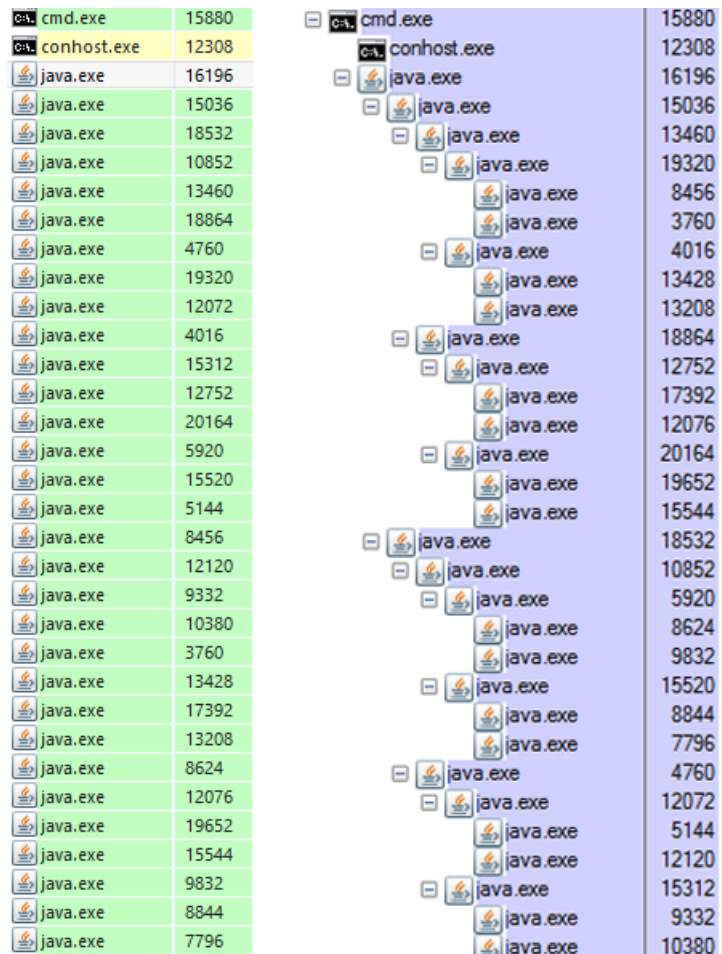


Figure 2.1: YAPM process tree at the left, Sysinternals at the right

### 2.1.3 UNIX: Glances

Glances [30] is a Python based system monitoring tool for UNIX-based systems. To install it on Ubuntu or any Debian-based distribution, one has to run the following commands in terminal:

```
sudo apt-get install python-pip build-essential python-dev
sudo pip install Glances
sudo pip install PySensors
```

The first command installs the necessary Python packages, the second one installs the tool itself, and the third one installs a package which allows the tool to monitor the system. After the installation, it can be started by typing “*glances*” in the terminal window. I used Ubuntu as one of my testing environment, so I will add only those commands that I use on it. On other UNIX-based systems, these commands can be different.

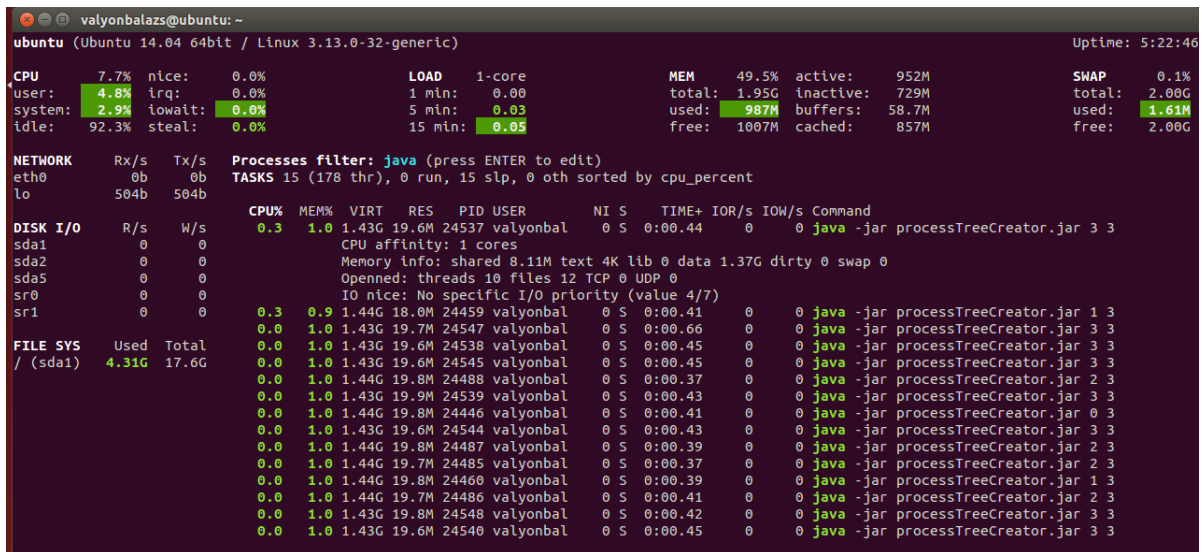


Figure 2.2: Ubuntu 14.04 with Glances 2.1.1

On Figure 2.2 a running Glances can be seen, where it shows processes filtered by the keyword: Java. Furthermore it shows the processor (CPU), memory (RAM) and disk usages.

### 2.1.4 UNIX: Bluepill

Bluepill [32] is not a visualized process monitoring tool, so it does not have a graphical user interface (GUI) like the Process Explorer or YAMP on Windows had, it uses command line display. It dynamically provides data for the user, and it monitors specific processes in the background. It was developed with the Ruby framework. It can be parameterized by a configuration file which processes should be monitored, and what to do if an event occurs. For

example, if a process stops, Bluepill can detect it and restart the process. Another example if we configure that a process cannot have more than five percent of processor usage, and if goes above three times, than the process should be restarted because it has malfunctioned.

Bluepill can be very useful, if an application creates a complex process tree like my process tree tester software, but the developer did not implement a process monitoring algorithm which can restart child processes when they are killed, or it would be really hard to implement that kind of monitoring algorithm because of the complexity of the application. In these cases, we can set Bluepill to watch over the process tree, and when a child process stops, it will restart it automatically.

To install Bluepill on Ubuntu, one should run the following commands:

```
sudo apt-get install ruby ruby-dev
sudo gem install bluepill
```

## 2.2 Summary of monitoring tools

In Table 2.1 I listed the process monitoring tools that I used during the testing. The more detailed description about them is in the previous section. The GUI grouping is referring to a window-based display, which the UNIX-based tools do not have, only a terminal-based display.

Name of the tool	Platform	GUI
Sysinternals Process Explorer	Windows	✓
YAMP	Windows	✓
Glances	UNIX	
Bluepill	UNIX	

Table 2.1: Summary of the used monitoring tools

## Chapter 3

# Measurement

### 3.1 The environment

I run my tests on three platforms, on a Windows 8.1 x64 desktop and two UNIX-based systems, an Ubuntu 14.04 and a Mac OSX 10.9. My test applications are using the Java 7 libraries, .NET 4.5 framework and Python 3 interpreter. The main focus will be on the Windows and Ubuntu, but I will test the application on OSX as well. But in this document I will not list those some different commands that the OSX needs, because all of other UNIX-based systems like almost every Linux works with these commands, and only the OSX is different.

### 3.2 My test applications

I listed my applications [50] in Table 3.1, which I implemented to test the feautres of those monitoring tools that I introduced in the previous chapter, and specific features of these operating systems such as process groups. Detailed descriptions can be found about these applications in the next sections.

<b>Name of the test application</b>	<b>Platform</b>	<b>Language</b>
Process Tree Creator	Windows, UNIX	Java
Two Process Application	Windows, UNIX	Python
Job Objects	Windows	C#

Table 3.1: Summary of my test applications

### 3.3 Testable process tree creation

To test and check the available tools on these platforms, I created different test applications. These are small, usually one or two purpose applications, which I use only to check and measure specific features of platforms and platform-dependent tools.

For the reliable and accurate measurement of these tools, I implemented most of my test applications on a platform-independent language like in Java or Python. In this way the results of the measurements are reliable, because if the tests give the same results on different platforms, they are probably correct and reliable. In this way, I am available to execute the same software on these platforms the same way. But I have to test some Windows specific elements such as Job Objects which can be used best with C# language, so, the only Windows specific test applications were written in this way.

### 3.4 Simple process tree creator

The process tree creator was written in Java, because I can use it on both Windows and UNIX platforms easily, and it does not require any platform-dependent API, I could create it with the official Java Class Library (JCL).

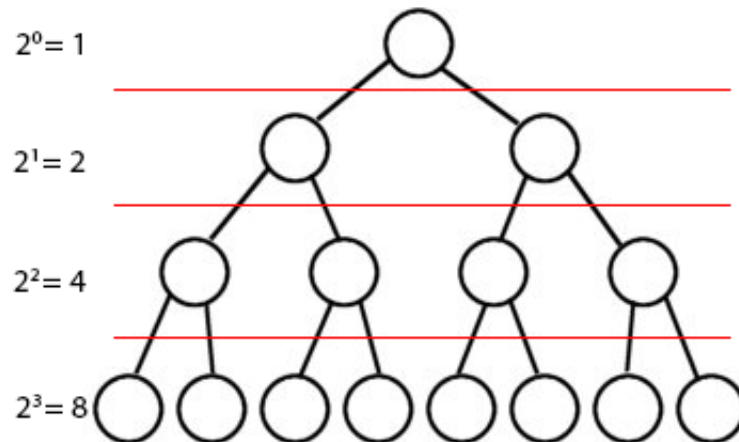


Figure 3.1: Binary process tree

This test application has only one purpose, which is to create a process tree, where the root has 2 children processes, monitors them, and ensures that, every process will always have 2 children. An example of this binary process tree can be seen on Figure 3.1. When a child process is stopped, the ancestor of that process will restart until, the root process is not stopped too. With this software, I can test the system monitoring tools capability of process

tree handling, such as ‘kill a single process’ or ‘kill a process tree’.

Every process in the process tree monitors its children, so, when any of its children stops, it will restart the process.

Every node in the tree starts 2 independent threads and each thread contains an infinite loop, where it checks if the started process stopped and returned or not. When the child process stopped, the thread will restart it. Every ancestor is responsible for the execution of its children. To achieve this functionality I used the Process class and its `waitFor` method, to check if a process is still running.

### 3.5 Two Process application, two-direction monitoring

To create this application, I ran into a few platform-dependent problems. During the specification and design phase, I tried to create a multiplatform software, which detects the type of the operating system and according to that will execute the appropriate part of the code.

I could write the Windows part with two external jar libraries [51], because the official Java Class Library has no direct support for using the native Windows API, but with the libraries I could reach the wanted functionality. This two libraries were the JNA-3.5.1 and the JNA-Platform-4.1.0, and with these I used the `kernel32.dll` from the operating system, to execute its methods for the process management. The most used object was the Native class from the `com.sun.jna` package of the JNA-3.5.1 library.

#### 3.5.1 Why Python and not Java

After that I started to work on the UNIX part of the application, I had to find out, that with these and other libraries, I cannot implement all of the functionalities that I could in the Windows part. Notably, it can be done, but it is more difficult, one has to create dll files that can access the native API methods, and that are imported into the Java code. Because of this, I choose a different language for the UNIX version of this software. C or C++ could be a good choice for the UNIX based operating systems, but to achieve this functionality I wanted to use a tool, which is easier to get started with such as a Python script .

I have ported the Java code into a python script. After I finished the code, I came to a conclusion that the python script is much simpler than the Java code, and it runs on both Windows and UNIX-based operating systems. To achieve this functionality I had to install

the python developer tools on the platforms, and it run smoothly. I used the psutil external library with the python 3 framework.

This solution creates 2 processes, a root and a child, and they monitor each other as it can be seen on Figure 3.2. When the child process is stopped, the parent process restarts it. Unlike the process tree creator, where every node was responsible only for its children, in this case, the child is also responsible for monitoring its ancestor. Because of this, both processes are responsible to check each other's state continuously, and when one of them is stopped, the other one has to restart it.

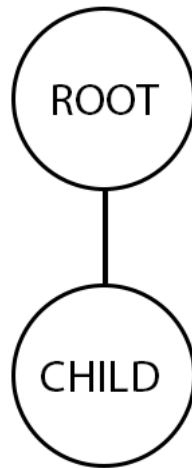


Figure 3.2: Two process application, two-way monitoring

To create this application, I used 2 external libraries, such as the JNA-3.5.1 and the JNA-Platform-4-1-0 in the Java version. I had to do this, because the official Java Class Library does not have support to use the lower level API of the operating system. For the Windows solution, I used kernel32.dll and its methods, to monitor the processes. The python script was mainly based on the psutil and the subprocess libraries.

### 3.6 Job Objects, the Windows platform only process tester

This is for testing the Job Objects. Because this application is only for the Windows platform, I have written this in C# language with .NET framework.

To use the Job Objects in the C# language based application, I had to use the original Windows API. To do this, I had to import kernel32.dll and use some methods from it, such as CreateJobObject, AssignProcessToJobObject and CloseHandle. I used native dll files from C#, so in fact this is a higher abstraction level that covers the original solution, which means

that the .NET framework and the C# language covers the native API methods, and gives a more readable and efficient developer solution.

After creating a few processes, I created a Job Object item with the `CreateJobObject` method, I assigned the already created processes to it. If I destroyed the Job object with the `CloseHandle`, all the processes which were assigned to the Job were stopped, but the others kept running, so the processes were successfully attached to it.

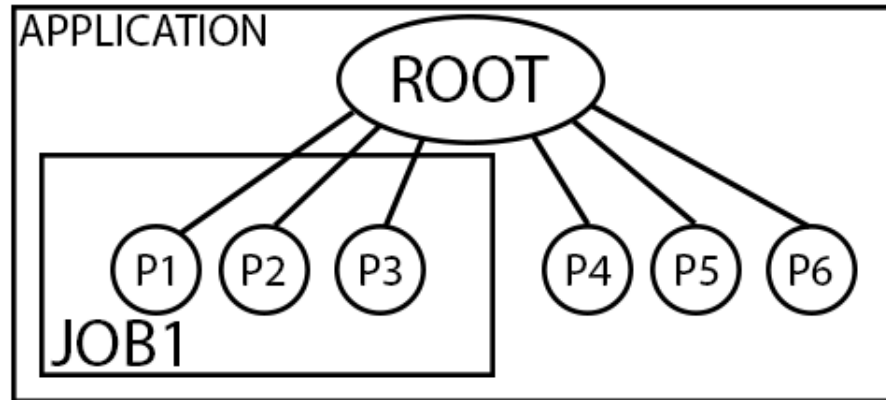


Figure 3.3: Job Objects test application architecture

### 3.7 Testing methods

During the testing phase, I used different methods for different solutions. First of all, I ran my applications that I wrote before, and I executed different operations on them from bash, command prompt, monitoring tools and from my applications.

### 3.8 Test the Process tree creator

The first test checks, how the process tools on different platforms can handle my process tree application, which can restart its own processes.

#### 3.8.1 Test on Windows with Sysinternals Process Explorer

I created a 5 level binary process tree with my `ProcessTreeCreator` application, than I executed different operations with the Process Explorer tool on it.

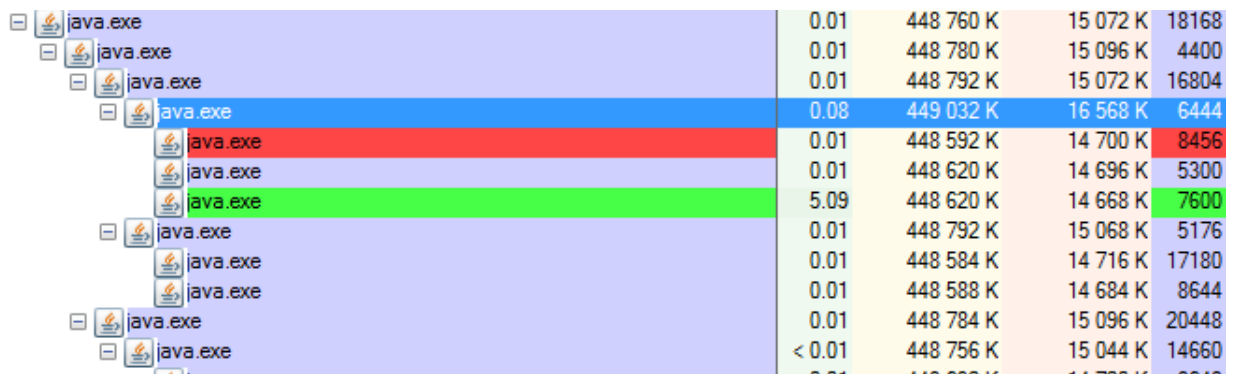


### 3.8.1.1 Kill a single child process

I use the “Kill process” command in the Sysinternals Process Explorer application, on different nodes in the binary process tree. I will check how my processTreeCreator application will react if I close processes on different levels of the tree, such as a lowest node, a node in the middle of the tree or the root.

#### 3.8.1.1.1 Kill a single child process on the lowest level

First, I tried to kill a child process on the lowest level. It has the 8456 process ID. After I killed it with the tool, the process stopped, but immediately a new one has been started by its ancestor, with the 7600 process id. I could make a screenshot about the moment, when the old process was being stopped and the new one was started.



java.exe	0.01	448 760 K	15 072 K	18168
java.exe	0.01	448 780 K	15 096 K	4400
java.exe	0.01	448 792 K	15 072 K	16804
java.exe	0.08	449 032 K	16 568 K	6444
java.exe	0.01	448 592 K	14 700 K	8456
java.exe	0.01	448 620 K	14 696 K	5300
java.exe	5.09	448 620 K	14 668 K	7600
java.exe	0.01	448 792 K	15 068 K	5176
java.exe	0.01	448 584 K	14 716 K	17180
java.exe	0.01	448 588 K	14 684 K	8644
java.exe	0.01	448 784 K	15 096 K	20448
java.exe	< 0.01	448 756 K	15 044 K	14660

Figure 3.4: Killing a single child process on the lowest level

The conclusion is, that in this process tree a process can be killed, but a new one always will be started, so the functionality which the process has will be accessible. So, in this way, the application cannot be killed. If I run this command in a loop, the application cannot be killed. The process will be stopped, but the application will not, its ancestor always will restart it.

The solution of this application can be used where high process reliability is expected. But it is important, that the node has to be on the lowest level of the tree. So, in this way, the process tree cannot be killed, if we execute the command on a child on the lowest level.

#### 3.8.1.1.2 Kill a single child process in the middle of the tree

If I stopped a child process in the middle of the tree, that part of the tree which was below of it, become orphan, but kept running and after the new process had been started by its ancestor, it created new children, and they their owns. So, a 5 level high binary tree has  $(1 + 2 + 4 + 8 + 16)$  31 nodes by default.

But if I kill a child process on any level which is not the first (the root is on the first level) or the lowest level of the tree where the leaves are placed, for example on the second level (below the root process, the red one), there will be 2 separated orphan trees, and each has 7 children in this case. Together, after the restart of the process, there will be  $(31 + 14)$  45 processes, where 14 processes will execute the same code. This can lead to a system fail.

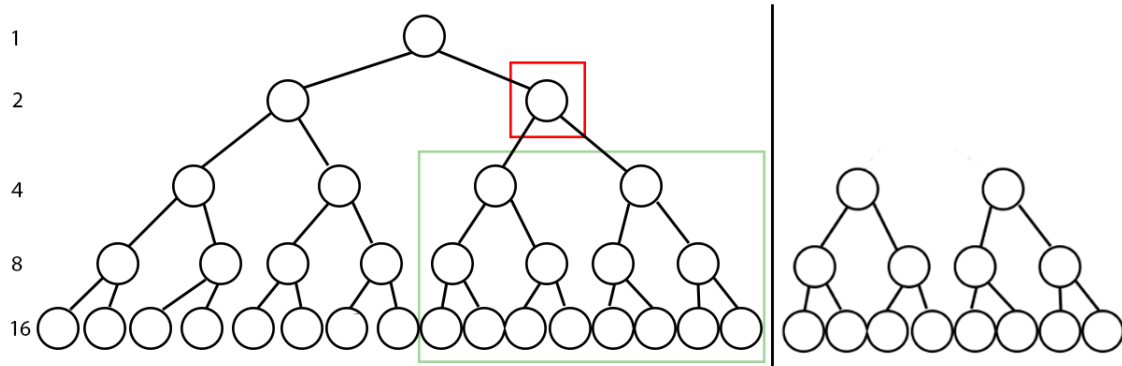


Figure 3.5: Kill a single child in the middle of the tree, and the orphans after

To prevent this kind of malfunction, there should be a check mechanism in every process, which periodically, in an infinite loop checks that how many processes are running on the same level and executing the same task as its ancestor does. Although this multiple orphan subtrees can be prevented, if the children has the ancestor detection function, because in that way, only the stopped ancestor would be restarted once, and not the whole part of tree as it can be seen on the Figure 3.5.

So in this way, the process tree can not be stopped, because the children processes will always be restarted by their ancestors, like in the previous chapter.

The implementation of the prevention mechanism is left as future work.

### 3.8.1.2 Kill the root process

I tried to kill the root process the same way as I tried to do it with a child. Because of the design of the application, the root process was not checked at all by any process, so it was not restarted after I executed the kill order.

After that, all of its child processes became orphans, but the lower levels of the tree have kept the tree structure as it can be seen on Figure 3.7, so the grand- and great-grandchildren processes stayed with their ancestor. If the orphan trees are checked in the Process Explorer, a four level tree is displayed as it can be seen on Figure 3.6. The process IDs are the same, so the direct children of the old root process has become the two new roots of the two new orphan trees.

	java.exe	0.01	448 780 K	15 104 K
	java.exe	0.01	448 792 K	15 072 K
	java.exe	0.01	449 032 K	16 568 K
	java.exe	0.01	448 620 K	14 696 K
	java.exe	0.01	448 620 K	14 680 K
	java.exe	0.01	448 792 K	15 068 K
	java.exe	0.01	448 584 K	14 716 K

Figure 3.6: A 4 level high orphan process tree

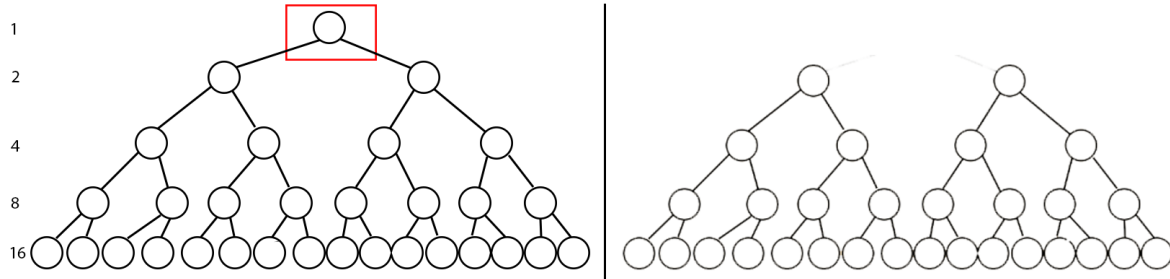


Figure 3.7: Kill the root process with Sysinternals Process Explorer

So, if we use this strategy for an algorithm, that starts on the top of the tree with the root process and after that goes on down-direction on the remnant of the tree, the process tree can be killed and stopped. This can be done, because only the ancestors are monitoring their children, but if the ancestors are stopped first, the children will not be restarted and can be terminated.

### 3.8.1.3 Kill the process tree

I tried to kill the whole process tree with the tool in different ways: I executed the “Kill process tree” command on different nodes of the tree, and I got variant results. Based on this, I assume that, the tool uses the single process kill method down way recursively. In this case, it will only kill processes which are children of that specific process, but does look for its ancestor.

#### 3.8.1.3.1 Kill the process tree by a node on the lowest level

As I executed the “Kill process tree” command on a node on the lowest level of the binary process tree, only one thing happened: that specific process has stopped, and was restarted by its own ancestor. All the other processes in the tree remained untouched.

#### 3.8.1.3.2 Kill the process tree by a node on a middle level

After the execution of the command, the process which I choose to click on was stopped and all of its children and their grandchildren stopped too. Just as before, its ancestor remained

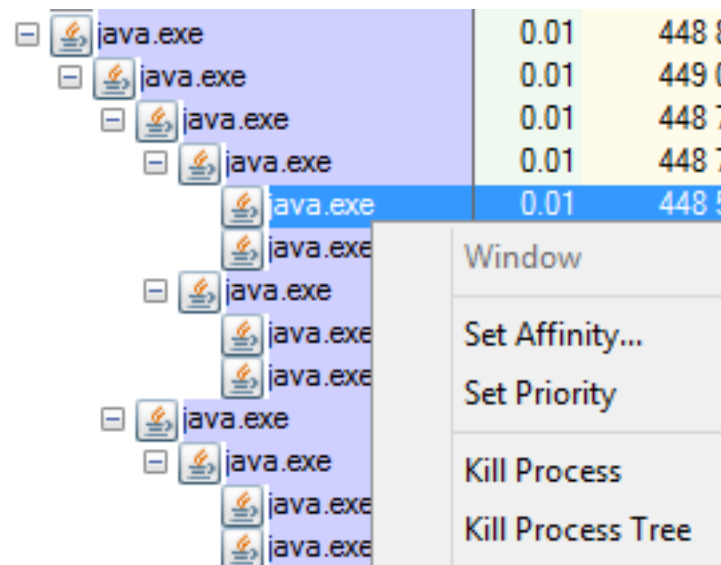


Figure 3.8: Execute 'Kill process tree' command on a lowest node with Sysinternals Process Explorer

untouched by the tool. I can assume that, the tool uses the kill process recursively, only in down direction in the tree.

After the execution of the termination command which can be seen on Figure 3.8 was finished, all of the stopped processes were restarted by their ancestor. This solution is far better than the “kill a single process” in the middle of a tree, because as I show that before, lots of orphan processes will remain in the system, but this uses a recursive technique and eliminate all of them, so there will be no duplicated processes.

#### 3.8.1.3.3 Kill the process tree by the root

If I clicked on the root process of the tree and I executed the command on it, the whole process tree was stopped. It could be done, because only the ancestor processes checked if their children is alive or not, and the tool’s algorithm goes from the top to the bottom on the tree.

If the processes have a mechanism that checks its own ancestor’s state, the process tree cannot be stopped with this method.

#### 3.8.1.3.4 Conclusion about Sysinternal’s kill methods

The single process kill method can stop a single, independent process. But my processTreeCreator application which uses a one-way checking algorithm down direction in the tree, made the tool to leave the operating system in an unstable state, because orphan processes can remain in the system which can lead to malfunction.

The process tree kill method was more effective than the single process kill mechanism. The most important result is, in these type of applications if we want to stop a single process or just restart it, we should use this method, because it will not leave orphan processes, but the children processes will be restarted, and this is not preferable in every situation.

But, if my application used a two-way checking algorithm, the process tree killer method would not be able to stop the tree either. So, if someone wants a reliable application which cannot be stopped easily or with this tool, should implement this kind of solution. This kind of double monitoring algorithm for a complete process tree is left as future work.

### 3.8.2 Test on Windows with YAMP

I will repeat the same tests as I did with Process Explorer on my special process tree. Furthermore, I will test other special functions of this tool, because it has a lots of special features that the Sysinternals Process Explorer does not have, such as I can choose which method I would like to use to kill a process.

#### 3.8.2.1 Kill a single child process

I will try to kill a node on the lowest level, on the middle of the process tree.

##### 3.8.2.1.1 Kill a single child process on the lowest level

I chose a lowest node of the tree with Process Explorer for the better visualization, then I looked by process ID in YAMP. I executed the default kill command, than the same event happened as with the Sysinternals Process Explorer, the process was first stopped, than restarted by its ancestor.

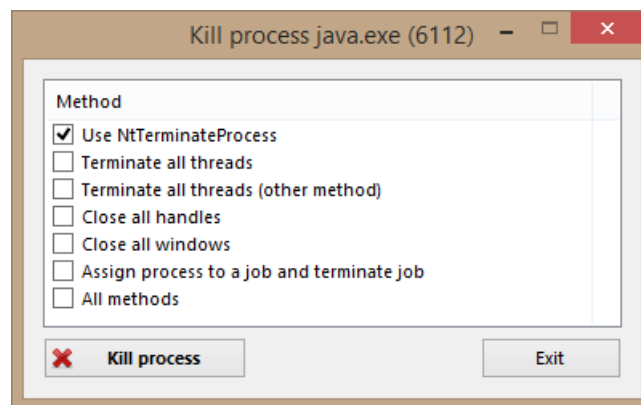


Figure 3.9: YAMP different process kill methods

After that, I executed the different kill methods, and got the same result, the process was restarted as I expected. But it's good to know, that the tool provides several ways to do it.

### 3.8.2.1.2 Kill a single child process on a middle level

After I killed the root process, all of its offsprings became orphans, but kept running as before with Sysinternals Process Explorer as it can be seen on Figure 3.10. The representation of the remaining process trees in the Process Explorer is more obvious than the YAMP's.

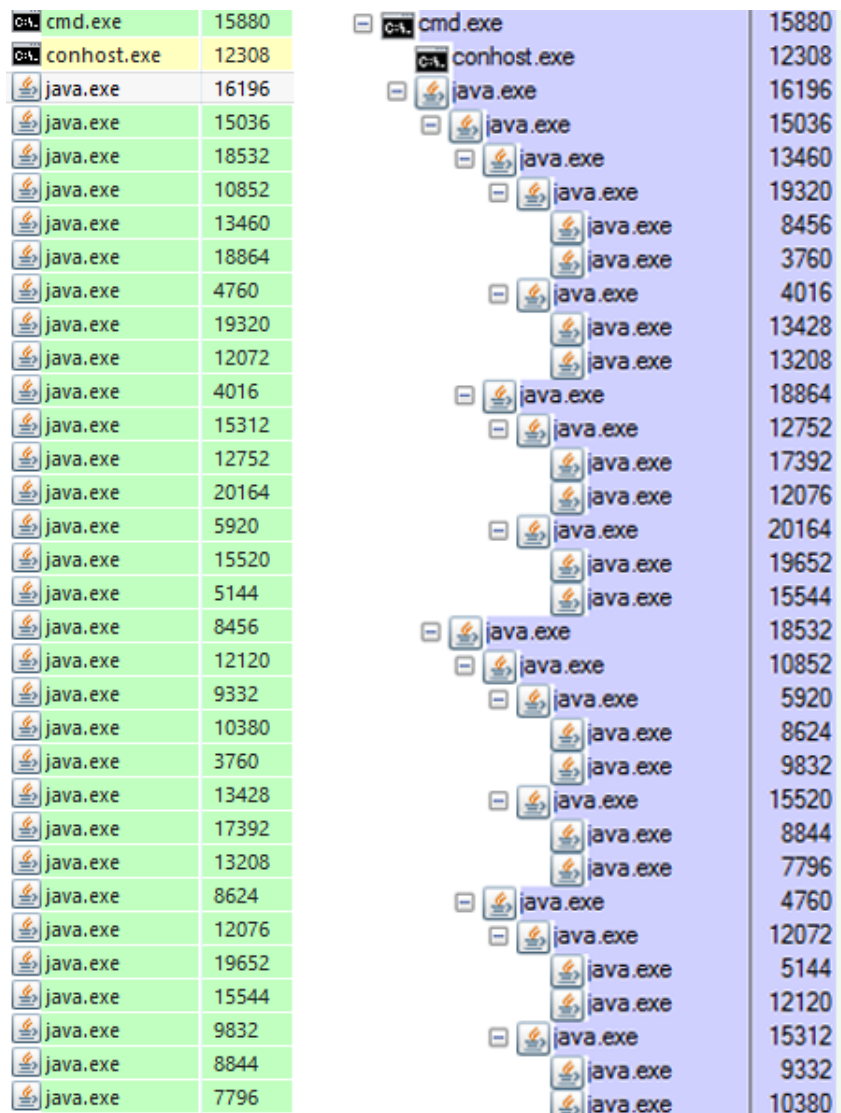


Figure 3.10: YAMP to the left, Process Explorer to the right, after the root was killed

### 3.8.2.2 Kill the process tree

As I did before, I try to kill the whole process tree by executing the command on nodes which are placed on different levels of the tree.

#### **3.8.2.2.1 Kill the process tree by a node on the lowest level**

YAMP uses the same algorithm as Process Explorer, it tries to kill only in down direction of the process tree, and does not check for its ancestor. Only the node was stopped and it was immediately restarted by its own ancestor.

#### **3.8.2.2.2 Kill the process tree by a node on a middle level**

As with Sysinternals Process Explorer, I managed to kill the process and its children and their grandchildren, but not the whole process tree. The command was only executed in down direction recursively.

#### **3.8.2.2.3 Kill the process tree by the root**

When I executed the command on the root element of the tree, the whole tree was stopped successfully. This could happen because of the only one-way checking algorithm which was implemented in the processTreeCreator application. If I made it two-way, this tool could not stop the tree either.

#### **3.8.2.3 Conclusion about YAMP's kill methods**

I tried different kill methods, not just the default two, a few was as effective as the default with my test application, other are not, but they could be in different cases, such as the "Close handlings" option.

The tool visualization functions about the process trees are weaker than the Sysinternal's, but every other feature is much more detailed and complex. As I said before, the tool only list the running processes and does not display the process tree, so it is really hard to determine, which process is connected to. For the system administrators, this can be a more powerful tool for monitoring and executing commands. It can monitor machines on remote locations, which the Sysinternals Process Explorer cannot.

### **3.8.3 Test on UNIX with Glances and bash commands**

To monitor the running processes in the operating system, I installed Glances, because it's a visualized monitoring tool for processes, and with it, I can check the results of my terminal commands. I will make the same tests as I did on Windows before.

I could use a simple terminal bash command to get the process ID of the nodes of the process tree, but using Glances has the advantage of auto-updating the list and I can see immediately the changes as it can be seen on Figure 3.11. But here are the other command:



```

valyonbalazs@ubuntu:~$ ps aux | grep java
valyonb+ 24446 0.0 0.9 1504840 20224 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 0 3
valyonb+ 24459 0.0 0.9 1504840 18420 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 24460 0.0 0.9 1504840 20224 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 24485 0.0 0.9 1504840 20220 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 24486 0.0 0.9 1504840 20212 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 24487 0.0 0.9 1504840 20296 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 24488 0.0 0.9 1504840 20228 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 24537 0.0 0.9 1502328 20116 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 24538 0.0 0.9 1502328 20112 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 24539 0.0 0.9 1502328 20328 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 24540 0.0 0.9 1502328 20112 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 24544 0.0 0.9 1502328 20112 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 24545 0.0 0.9 1502328 20104 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 24547 0.0 0.9 1502328 20220 pts/9 Sl+ 17:08 0:01 java -jar processTreeCreator.jar 3 3
valyonb+ 24548 0.0 0.9 1502328 20320 pts/9 Sl+ 17:08 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 25171 0.0 0.0 17428 924 pts/22 S+ 17:34 0:00 grep --color=auto java

Processes filter: java (press ENTER to edit)
TASKS 15 (178 thr), 0 run, 15 slp, 0 oth sorted by cpu_percent

  CPU% MEM% VIRT RES PID USER NI S TIME+ IOR/s IOW/s Command
  0.3 1.0 1.43G 19.6M 24537 valyonbal 0 S 0:00.44 0 0 java -jar processTreeCreator.jar 3 3
    CPU affinity: 1 cores
    Memory info: shared 8.11M text 4K lib 0 data 1.37G dirty 0 swap 0
    Openned: threads 10 files 12 TCP 0 UDP 0
    IO nice: No specific I/O priority (value 4/7)
  0.3 0.9 1.44G 18.0M 24459 valyonbal 0 S 0:00.41 0 0 java -jar processTreeCreator.jar 1 3
  0.0 1.0 1.43G 19.7M 24547 valyonbal 0 S 0:00.66 0 0 java -jar processTreeCreator.jar 3 3
  0.0 1.0 1.43G 19.6M 24538 valyonbal 0 S 0:00.45 0 0 java -jar processTreeCreator.jar 3 3
  0.0 1.0 1.43G 19.6M 24545 valyonbal 0 S 0:00.45 0 0 java -jar processTreeCreator.jar 3 3
  0.0 1.0 1.44G 19.8M 24488 valyonbal 0 S 0:00.37 0 0 java -jar processTreeCreator.jar 2 3
  0.0 1.0 1.43G 19.9M 24539 valyonbal 0 S 0:00.43 0 0 java -jar processTreeCreator.jar 3 3
  0.0 1.0 1.44G 19.8M 24446 valyonbal 0 S 0:00.41 0 0 java -jar processTreeCreator.jar 0 3
  0.0 1.0 1.43G 19.6M 24544 valyonbal 0 S 0:00.43 0 0 java -jar processTreeCreator.jar 3 3
  0.0 1.0 1.44G 19.8M 24487 valyonbal 0 S 0:00.39 0 0 java -jar processTreeCreator.jar 2 3
  0.0 1.0 1.44G 19.7M 24485 valyonbal 0 S 0:00.37 0 0 java -jar processTreeCreator.jar 2 3
  0.0 1.0 1.44G 19.8M 24460 valyonbal 0 S 0:00.39 0 0 java -jar processTreeCreator.jar 1 3
  0.0 1.0 1.44G 19.7M 24486 valyonbal 0 S 0:00.41 0 0 java -jar processTreeCreator.jar 2 3
  0.0 1.0 1.43G 19.8M 24548 valyonbal 0 S 0:00.42 0 0 java -jar processTreeCreator.jar 3 3
  0.0 1.0 1.43G 19.6M 24540 valyonbal 0 S 0:00.45 0 0 java -jar processTreeCreator.jar 3 3

```

Figure 3.11: Upper half: terminal command process list, lower half: Glances process list

```
ps aux | grep java
```

To kill a process, I will use the command as I presented in the first chapter, with signal 15, so if I would like to kill the process with the 24547 process ID:

```
kill -15 24547
```

### 3.8.3.1 Kill a single child process on the lowest level

I executed the kill terminal command on a lowest node of the process tree, and the same happened as before on Windows, the process was stopped and then restarted by its ancestor.

### 3.8.3.2 Kill a single child process on a middle level

As I expected, the result was the same as on Windows, if I killed a process in the middle of the tree, the process was immediately restarted by its ancestor and the lower part of the tree became orphan and kept running. The screenshot below shows the processes before the command, and after that. For the smaller image, I used a 4 level high tree, not a 5.



```

valyonbalazs@ubuntu:~$ ps aux | grep java
valyonb+ 26158 3.5 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 0 3
valyonb+ 26170 3.5 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 26171 3.5 0.9 1504840 20344 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 26196 3.5 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26197 3.5 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26200 3.5 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26201 3.5 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26248 3.0 0.9 1502328 20208 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26249 3.0 0.9 1502328 20212 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26251 3.0 0.9 1502328 20204 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26252 3.5 0.9 1502328 20208 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26254 3.0 0.9 1502328 20208 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26255 3.5 0.9 1502328 18396 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26256 3.0 0.9 1502328 20204 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26257 3.5 0.9 1502328 18404 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26338 0.0 0.0 17428 924 pts/22 S+ 17:57 0:00 grep --color=auto java
valyonbalazs@ubuntu:~$ kill -15 26171
valyonbalazs@ubuntu:~$ ps aux | grep java
valyonb+ 26158 0.3 0.9 1504840 20220 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 0 3
valyonb+ 26170 0.3 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 26196 0.3 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26197 0.3 0.9 1504840 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26200 0.3 0.9 1504840 20396 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26201 0.3 0.9 1504840 20396 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26248 0.3 0.9 1502328 20208 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26249 0.3 0.9 1502328 20212 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26251 0.3 0.9 1502328 20204 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26252 0.3 0.9 1502328 20208 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26254 0.3 0.9 1502328 20208 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26255 0.3 0.9 1502328 18396 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26256 0.3 0.9 1502328 20204 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26257 0.3 0.9 1502328 18404 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26345 2.0 0.9 1504840 20352 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 26357 2.3 0.9 1504840 20220 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26358 2.3 0.9 1504840 20212 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 26383 1.6 0.9 1502328 20216 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26384 1.6 0.9 1502328 20208 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26385 1.6 0.9 1502328 20212 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26386 2.0 0.9 1502328 18404 pts/9 SL+ 17:57 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 26429 0.0 0.0 17428 924 pts/22 S+ 17:57 0:00 grep --color=auto java
valyonbalazs@ubuntu:~$

```

Figure 3.12: Killed a process on the middle of the tree, orphans kept running

### 3.8.3.3 Kill the root process

After I killed the root process, the process tree kept running as it did before on Windows. Because the root process is not checked by any other processes, it was not restarted.

### 3.8.3.4 Kill the process tree

As I did before on Windows, I will execute the process tree killer command on nodes at different levels of the process tree, and examine what happens with the node and the tree.

#### 3.8.3.4.1 Kill the process tree by process group ID

To do this, I used the command I wrote before. Such as I would like to kill a process group which has the group ID of 27801:

```
kill -9 -27801
```

Even so to kill my processTreeCreator application on Windows was really hard, because I had to execute commands recursively just on the root node, or go through the tree in a specific order, from the top to the lowest level. But on UNIX with the process group solution, it was pretty easy and was successful, only just one command has to be executed as it can

be seen on Figure 3.13.

```
valyonbalazs@ubuntu:~$ ps aux | grep java
valyonb+ 27801 1.6 0.9 1504840 20220 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 0 3
valyonb+ 27813 1.4 0.9 1504840 20332 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 27814 1.4 0.9 1504840 20216 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 27839 1.4 0.9 1504840 20356 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 27840 1.4 0.9 1504840 20220 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 27843 1.4 0.9 1504840 20216 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 27844 1.4 0.9 1504840 20216 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 27890 1.4 0.9 1502328 20208 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27891 1.4 0.9 1502328 18404 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27892 1.4 0.9 1502328 20208 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27893 1.4 0.9 1502328 20204 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27900 1.4 0.9 1502328 20200 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27901 1.4 0.9 1502328 20344 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27908 1.4 0.9 1502328 20204 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27909 1.4 0.9 1502328 20212 pts/9 SL+ 18:35 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 27980 0.0 0.0 17428 924 pts/22 S+ 18:35 0:00 grep --color=auto java
valyonbalazs@ubuntu:~$ kill -9 -27801
valyonbalazs@ubuntu:~$ ps aux | grep java
valyonb+ 28003 0.0 0.0 17424 924 pts/22 S+ 18:36 0:00 grep --color=auto java
valyonbalazs@ubuntu:~$
```

Figure 3.13: UNIX: kill a process tree by process group ID

There is a special case, where the process tree has a process group id, and some of its inner processes have a different one, or the process group has inner groups. In this case, not all of the processes can be stopped with a single command.

#### 3.8.3.4.2 Kill the process tree by a recursive algorithm

I found a very well written algorithm, which recursively kills a process tree, I named it as the killtree algorithm shell script [18]. This command is not implemented by default on the UNIX-based systems, but I made a shell script [50], and ran it manually, and the result can be seen on Figure 3.14.

```
valyonbalazs@ubuntu:~$ ps aux | grep java
valyonb+ 29081 1.0 0.9 1504840 20216 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 0 3
valyonb+ 29093 1.1 0.9 1504840 20220 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 29094 1.0 0.9 1504840 20212 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 1 3
valyonb+ 29119 1.0 0.9 1504840 18412 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 29120 1.0 0.9 1504840 20212 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 29123 1.0 0.9 1504840 20220 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 29124 1.1 0.9 1504840 20216 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 2 3
valyonb+ 29169 1.0 0.9 1502328 18396 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29170 1.1 0.9 1502328 18388 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29172 1.0 0.9 1502328 20212 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29173 1.0 0.9 1502328 20212 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29182 1.1 0.9 1502328 20212 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29183 1.1 0.9 1502328 20212 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29187 1.0 0.9 1502328 20204 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29188 1.0 0.9 1502328 20212 pts/9 SL+ 19:33 0:00 java -jar processTreeCreator.jar 3 3
valyonb+ 29260 0.0 0.0 17428 924 pts/22 S+ 19:33 0:00 grep --color=auto java
valyonbalazs@ubuntu:~$ ./killtree.sh 29081 9
valyonbalazs@ubuntu:~$ ps aux | grep java
valyonb+ 29278 0.0 0.0 17424 924 pts/22 S+ 19:34 0:00 grep --color=auto java
valyonbalazs@ubuntu:~$
```

Figure 3.14: UNIX: kill tree recursive algorithm

It does the same job, as on Windows the Sysinternals or YAMP does when I execute the “kill process tree” command. It goes down direction from the root, and systematically kills the processes. It can be done, because my application only uses a one-direction checking.

### 3.8.4 Test on UNIX with Bluepill

To use Bluepill, I had to create a configuration file, where I specified that I would like Bluepill to monitor my processTreeCreator application. If the application was stopped, it has to restart it. Furthermore, I used the killtree bash script that I used before, to systematically kill every process in the process tree and just after that start the application again. By this method, the duplication can be eliminated.

```
valyonbalazs@ubuntu:~/Desktop$ ps aux | grep java
valyonb+ 6500 0.0 0.0 17444 924 pts/0 S+ 04:59 0:00 grep --color=auto java
valyonbalazs@ubuntu:~/Desktop$ sudo bluepill load config
[sudo] password for valyonbalazs:
valyonbalazs@ubuntu:~/Desktop$ ps aux | grep java
root 6513 2.5 0.8 213676 17940 ? SL 04:59 0:00 bluepilld: java

root 6524 10.5 0.9 1504840 20408 ? SL 04:59 0:00 java -jar processTreeCreator.jar 0 2
root 6538 1.7 0.9 1504840 20220 ? SL 04:59 0:00 java -jar processTreeCreator.jar 1 2
root 6539 1.7 0.9 1504840 20216 ? SL 04:59 0:00 java -jar processTreeCreator.jar 1 2
root 6563 2.6 0.9 1502328 20212 ? SL 04:59 0:00 java -jar processTreeCreator.jar 2 2
root 6565 2.6 0.9 1502328 20212 ? SL 04:59 0:00 java -jar processTreeCreator.jar 2 2
root 6568 2.3 0.9 1502328 20276 ? SL 04:59 0:00 java -jar processTreeCreator.jar 2 2
root 6569 2.3 0.9 1502328 20208 ? SL 04:59 0:00 java -jar processTreeCreator.jar 2 2
valyonb+ 6609 0.0 0.0 17444 924 pts/0 S+ 05:00 0:00 grep --color=auto java
valyonbalazs@ubuntu:~/Desktop$ sudo ./killtree.sh 6524 9
valyonbalazs@ubuntu:~/Desktop$ ps aux | grep java
root 6513 0.5 0.8 213676 17940 ? SL 04:59 0:00 bluepilld: java

root 6628 20.0 0.9 1504840 20404 ? SL 05:00 0:00 java -jar processTreeCreator.jar 0 2
root 6642 3.0 0.9 1504840 20356 ? SL 05:00 0:00 java -jar processTreeCreator.jar 1 2
root 6643 3.0 0.9 1504840 20248 ? SL 05:00 0:00 java -jar processTreeCreator.jar 1 2
root 6668 2.5 0.9 1502328 20208 ? SL 05:00 0:00 java -jar processTreeCreator.jar 2 2
root 6669 2.5 0.9 1502328 20372 ? SL 05:00 0:00 java -jar processTreeCreator.jar 2 2
root 6670 2.5 0.9 1502328 20212 ? SL 05:00 0:00 java -jar processTreeCreator.jar 2 2
root 6671 2.5 0.9 1502328 20204 ? SL 05:00 0:00 java -jar processTreeCreator.jar 2 2
valyonb+ 6713 0.0 0.0 17444 924 pts/0 S+ 05:00 0:00 grep --color=auto java
valyonbalazs@ubuntu:~/Desktop$ sudo bluepill quit
Killing bluepilld[6513]
valyonbalazs@ubuntu:~/Desktop$ sudo ./killtree.sh 6628 9
valyonbalazs@ubuntu:~/Desktop$ ps aux | grep java
valyonb+ 6729 0.0 0.0 17444 924 pts/0 S+ 05:00 0:00 grep --color=auto java
valyonbalazs@ubuntu:~/Desktop$
```

Figure 3.15: UNIX: Bluepill restarting the process tree after stop

As it can be seen on Figure 3.15, before the start of the Bluepill, the processTreeCreator was not running. After the start, I listed the running processes, where the process IDs are presented. I executed the recursive killtree script on the root element of the process tree, and it was successful, because with the next listing, a new process tree was listed with new process IDs, and their start time was one minute later. Then I stopped the Bluepill and ran again the killtree. After that, the process tree was completely stopped and was not restarted.

My application can take care of itself except if the root node is terminated, but this can be eliminated with the two-direction monitoring algorithm, as I mentioned before. So, the

Bluepill tool can be useful for applications which do not have these kind of mechanisms. In this case, this tool was only helpful if the root was terminated.

So the conclusion is, that if a process tree was successfully stopped, Bluepill can automatically restart it as a failsafe solution. If our application has a monitoring algorithm (as it has), but its not two-directional and can be stopped, this mechanism can ensure, that the process tree will be regenerated. This reliability can be important if an application has to be running in a specific time period, or after it failed or stopped will be restarted.

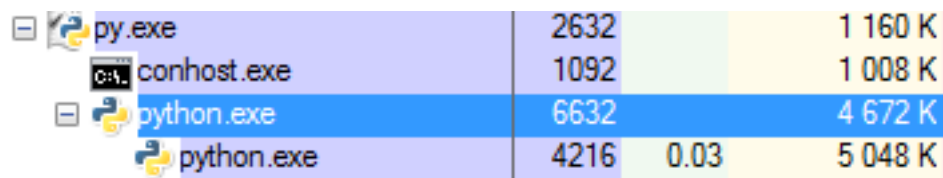
### 3.9 Two Process application, two-direction monitoring

I have written this application in two forms, a Windows-only Java version and a platform-independent python script. The python script can run on both platforms, so my main focus will be on this application. The algorithm creates a root and a child process, which monitor each other.

To run on Windows, I had to install the python interpreter, which I downloaded from the official python site. On UNIX, I had to install a few packages to get the python interpreter running. Furthermore, I had to install manually the psutil external library on both platforms, which on Windows was only a few clicks, but on UNIX it had other dependencies too.

#### 3.9.1 Windows: kill the child

I monitored the processes with Sysinternals Process Explorer on Windows. After I started the parentProcessCreator root process from a python script, it started the child process from another script file, which can be seen from the process monitoring tool.



py.exe	2632		1 160 K
conhost.exe	1092		1 008 K
python.exe	6632		4 672 K
python.exe	4216	0.03	5 048 K

Figure 3.16: Python two directional monitoring processes

I killed the child process with the process ID of 4216, it was restarted by the root. It can be seen, that the process ID of the root is the same as before.

#### 3.9.2 Windows: kill the root

I killed the root process with the process ID of 6632. As I expected, the root process was stopped, than restarted by child, so this time, the child process survived with the same

py.exe	2632	1 160 K
conhost.exe	1092	1 008 K
python.exe	6632	4 672 K
python.exe	3492	5 052 K

Figure 3.17: Python: the restarted child process

process ID, and the root has a new one. The interesting part is that, now the root became the child process, and the child became the root. If I kill the new root, it will become the child, and the old root will be root again.

python.exe	3492	0.04
python.exe	152	0.04

Figure 3.18: Windows-Python: after the root process was killed

### 3.9.3 UNIX: kill the child

I started the application, and I waited until, both the root and the child process were created successfully. Then I killed the child from the terminal, which was immediately restarted by the root process, as I expected.

### 3.9.4 UNIX: kill the root

I killed the root process and it also was restarted as before on Windows, so the functionality is the same, and it works the same.

```
valyonbalazs@ubuntu:~$ ps aux | grep python
root      1304  0.0  0.5 60540 11292 ?        Ss   01:51   0:06 /usr/bin/python /usr/bin/supervisord -c /etc/supervisor/supervisord.conf
valyonb+ 22093  0.0  5.2 661548 107944 ?        SNL  09:34   0:04 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map
valyonb+ 23330  0.1  0.4 39796 8492 pts/2    S+   11:04   0:00 python3 parentProcessCreator.py
valyonb+ 23331  0.2  0.4 48336 9984 pts/2    S+   11:04   0:00 python childProcessCreator.py 23330
valyonb+ 23338  0.0  0.0 17448  924 pts/8    S+   11:04   0:00 grep --color=auto python
valyonbalazs@ubuntu:~$ kill -15 23331
valyonbalazs@ubuntu:~$ ps aux | grep python
root      1304  0.0  0.5 60540 11292 ?        Ss   01:51   0:06 /usr/bin/python /usr/bin/supervisord -c /etc/supervisor/supervisord.conf
valyonb+ 22093  0.0  5.2 661548 107944 ?        SNL  09:34   0:04 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map
valyonb+ 23330  0.0  0.4 39796 8504 pts/2    S+   11:04   0:00 python3 parentProcessCreator.py
valyonb+ 23341  1.3  0.4 48336 9984 pts/2    S+   11:05   0:00 python childProcessCreator.py 23330
valyonb+ 23343  0.0  0.0 17448  924 pts/8    S+   11:05   0:00 grep --color=auto python
valyonbalazs@ubuntu:~$ kill -15 23330
valyonbalazs@ubuntu:~$ ps aux | grep python
root      1304  0.0  0.5 60540 11292 ?        Ss   01:51   0:06 /usr/bin/python /usr/bin/supervisord -c /etc/supervisor/supervisord.conf
valyonb+ 22093  0.0  5.2 661548 107944 ?        SNL  09:34   0:04 /usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map
valyonb+ 23341  0.2  0.4 48336 10012 pts/2    S   11:05   0:00 python childProcessCreator.py 23330
valyonb+ 23346  2.3  0.4 48336 9972 pts/2    S   11:05   0:00 python parentProcessCreator.py 23341
valyonb+ 23348  0.0  0.0 17448  924 pts/8    S+   11:05   0:00 grep --color=auto python
valyonbalazs@ubuntu:~$
```

Figure 3.19: UNIX-Python: after the root process was killed

As it can be seen on Figure 3.19, after the start, the root created a child, then I killed it. I listed the processes again, where the root has the same process ID but the child has a new one, because it's a new process. After that, I killed the root and listed them again, the child



became the root and has the same process ID as before, and now the old-root (new child) has a new one.

### 3.9.5 Testing Windows Job Objects

In this section I will examine the Windows Job Objects API. I wrote a test application, which creates 6 processes, and it adds the first 3 of them to a Job. Then it waits for 10 seconds, and stops the Job. This mechanism tries to simulate the UNIX process groups, where I assign some processes to a group, then I send a kill signal to that specific group.

For the testing, I used notepad.exe because it can be seen well when the job stops. Furthermore I followed the execution in Sysinternals for the better visualization.

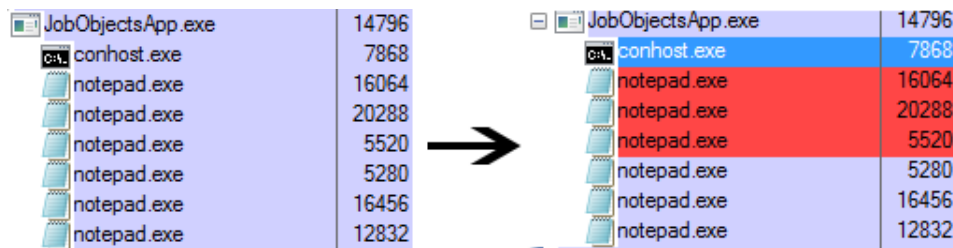


Figure 3.20: Windows Job Objects

As it was expected, the first three of the six notepads were killed after 10 seconds as it can be seen on Figure 3.20. So it can be said, that the Job Objects API and the newer, more sophisticated Nested Job API are similar to the process groups in the UNIX systems.

There are special cases, for example that this application use other applications which has Job Objects insted of a simple notepad. In this case, a few other applications will be started and added to a Job, which has inner Jobs. If I stop this Job, the application which are assigned to it will be stopped, but those applications that are assigend to its inner Job Objects will be not, so they will be running like orphans. In this special case, the Job Objects also not a perfect solution.

## Chapter 4

# Conclusion

### 4.1 Only the API of the Operating System

In the Table 4.1 below, I summarized, what kind of nodes can be killed with a single API call. The kill child or kill root column is about a permanent destroying, if the node is restarted by another node, it's not a success. At the one-directional process tree, the monitoring was always in down direction, so in this case, the one-directional process tree can only be killed, if I executed the recursive killing algorithm on the root, because only the ancestors did monitoring on their children, and the ancestors can be killed if their ancestors were killed before.

	Kill child	Kill root	Kill process tree
Process tree (no monitoring)	✓	✓	✓
One-directional process tree		✓	✓
Two-directional process tree			✓ (recursively)

Table 4.1: Summary of the successrate of different API calls

These results are the same on Windows and on UNIX even though the used API and mechanisms are different.

### 4.2 Extended mechanisms

In these table below, I focus on the process group of the UNIX and the Job Objects of the Windows platform.

The main conclusion of these tables is, that if a process has to be stopped on any UNIX-based platforms, and we are not sure if it has children processes, the process group commands should be used. Even so on Windows platforms the ultimate solution is not clear, because

if a third-party application should be monitored, we cannot use the Job Objects out of box, but we can create a small application where we manually add that application and start from it. Only this way can we be sure, that the whole process tree will be stopped.

As I mentioned before, there are special cases, when a Job Object has inner Jobs, or a process group has inner groups, in these cases not all processes can be stopped with a single command or stopped at all. These cases are left as future work.

#### 4.2.1 Process group commands

As it can be seen in Table 4.2, when I executed a process group command on any process tree, it was done successfully, regardless was there any kind of node monitoring or not.

	Kill child	Kill root	Kill process tree
Process tree (no monitoring)	✓	✓	✓
One-directional process tree	✓	✓	✓
Two-directional process tree	✓	✓	✓

Table 4.2: Summary of the successrate of the process group commands

#### 4.2.2 Job Object commands

The Job Objects behaved the same on Windows as the process group did on UNIX.

	Kill child	Kill root	Kill process tree
Process tree (no monitoring)	✓	✓	✓
One-directional process tree	✓	✓	✓
Two-directional process tree	✓	✓	✓

Table 4.3: Summary of the successrate of Job Objects

### 4.3 Future work

It worth checking, can be a creation of a complex and general process tree, which is platform-independent and has a two-directional monitoring algorithm over every node in the tree. Generality is a main goal, such as path of applications can be given as parameters, and the algorithm will build a whole tree from these. Furthermore, the platform-independency makes it easy to use, because the user has to learn only one syntax and can run it anywhere, and the maintenance and the continuous development is much easier if it has to be done in only one language.



# Chapter 5

## Summary

### 5.1 Summary

I have created 3 different applications, which significantly consisted of 4 individual softwares, because the parent-child process creator application was a pair, where the parent started the child, and the child could restart the parent, and these had to be individual files. The applications are not too big, only a few hundred lines of code, but it was not very trivial to design them.

I tested my applications on 3 different platforms, a Windows 8.1 and two UNIX-based, an Ubuntu 14.04 and a Mac OS 10.9 (Mavericks).

I have written my algorithms on 5 different languages, such as Java, C# and Python, used some libraries based on C++ and bash scripts.

So it can be said, that I used a lot of very different kind of technology to test these ideas and solutions, so because of this, my test results can be considered comprehensive. Even so I tested different platforms with different architectural design and philosophy, the results were almost totally similar. It can be stated that the same functionality can be achieved on both platforms with different techniques, such as the process groups on UNIX and the Job Objects on Windows.

As I demonstrated, with platform independent languages such as Java or Python, applications can be written with the same code and they react to commands the same way on both platforms, such as my Python script did it. With this kind of solution, if there is no need for some low-level code because of performance issues such as a C++ code, with one application, several platforms can be reached with one step. Reliable and platform-independent

applications can be written.

I could create and destroy whole process trees on both platforms with more than one way, but with different algorithms, while they gave the same result.

With the one-directional monitoring test application, I could not successfully kill children processes on any of the platforms, because the ancestors restarted them. I demonstrated this with the platform independent Java code. Furthermore I was only able to kill the root processes successfully, and from the root node kill the process tree recursively.

The two-directional application is really hard to kill, almost all of the techniques failed, but the process-group command worked on UNIX, and the Job Objects on Windows as well. I could not kill these processes with any casual methods, but process groups and Job Objects worked. Some special cases are not covered in my tests, such as when a sub-tree of a process-tree belongs to another process group, which has nodes that belongs to a different one. These cases are left as future work.

# Bibliography

- [1] Downloaded: 25th July 2014., Admin, OSInfoBlog, April, 2013, <http://www.osinfoblog.com/post/29/operating-system-concepts---processes/>
- [2] J. Y. Abe, Software process monitor, US7823021 B2, 26 10 2010.
- [3] Downloaded: 1st August 2014., Mutual Exclusion, Wikipedia article, [http://en.wikipedia.org/wiki/Mutual\\_exclusion](http://en.wikipedia.org/wiki/Mutual_exclusion)
- [4] J. M. Tim, Gnu/Linux Application Programming, ISBN:1-58450-568-0.
- [5] Downloaded: 3rd August 2014., M. Slattery, Stackoverflow, 01 07 2011., <http://stackoverflow.com/questions/6548823/use-and-meaning-of-session-and-process-group-in-unix>
- [6] S. A. R. W. Richard Stevens, Advanced Programming in the UNIX Environment ISBN: 0321637739
- [7] Downloaded: 27th July 2014., J. Frost, Computer Science, UC Santa Barbara, 17 08 1994. <http://www.cs.ucsb.edu/~almeroth/classes/W99.276/assignment1/signals.html>
- [8] J. P. T. O. M. L. Shelley Powers, Unix Power Tools.
- [9] Downloaded: 5th August 2014., V. G. a. D. P. Archana Ganapathi, Usenix, University of California, Berkeley, 03 12 2006, [http://static.usenix.org/events/lisa06/tech/full\\_papers/ganapathi/ganapathi\\_html](http://static.usenix.org/events/lisa06/tech/full_papers/ganapathi/ganapathi_html)
- [10] T. S. Bartley, Method for mutual computer process monitoring and restart. USA, US20020184295, 9 08 2005
- [11] Downloaded: 8th August 2014., linux.die.net, <http://linux.die.net/man/2/getpid>
- [12] Downloaded: 8th August 2014., linux.die.net, <http://linux.die.net/man/2/getppid>

- [13] Downloaded: 9th August 2014., GetProcessId function, msdn.microsoft.com, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms683215\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms683215(v=vs.85).aspx)
- [14] Downloaded: 9th August 2014., username: Jay, Stackoverflow, 17 02 2009. <http://stackoverflow.com/questions/185254/how-can-a-win32-process-get-the-pid-of-its-parent>
- [15] Downloaded: 11th August 2014., ManagementObject Class, msdn.microsoft.com, [http://msdn.microsoft.com/en-us/library/system.management.managementobject\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system.management.managementobject(v=vs.110).aspx)
- [16] Downloaded: 16th August 2014., TaskList, ss64.com, <http://ss64.com/nt/tasklist.html>
- [17] Downloaded: 16th August 2014., Wmic, msdn.microsoft.com, [http://msdn.microsoft.com/en-us/library/aa394531\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394531(v=vs.85).aspx)
- [18] Downloaded: 4th August 2014., username: olibre, Stackoverflow, 28 02 2013. <http://stackoverflow.com/questions/392022/best-way-to-kill-all-child-processe>
- [19] Downloaded: 5th August 2014., username: wy, Stackoverflow 20 05 2012. <http://stackoverflow.com/questions/16639275/grouping-child-processes-with-setpgid>
- [20] Downloaded: 7th August 2014., setpgid function , manpages.ubuntu.com, <http://manpages.ubuntu.com/manpages/raring/man2/setpgid.2.html>
- [21] Downloaded: 18th August 2014., Kevin S., Stackoverflow, 02 03 2011. <http://stackoverflow.com/questions/5161321/change-process-groups-on-runtime-getruntime-exec-processes>
- [22] Downloaded: 23rd August 2014., Job Objects, msdn.microsoft.com, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms684161\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx)
- [23] Downloaded: 23rd August 2014., Nested Jobs, msdn.microsoft.com, [http://msdn.microsoft.com/en-us/library/hh448388\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/hh448388(v=vs.85).aspx)
- [24] Downloaded: 25th August 2014., username: Contango, Stackoverflow, 01 05 2012. <http://stackoverflow.com/questions/5901679/kill-process-tree-programatically-in-c-sharp>

- [25] Downloaded: 28th August 2014., MOBZystems, MOBZystems, <http://www.mobzystems.com/code/killprocesstree>
- [26] Downloaded: 1st September 2014., username: L. M. Silva, [social.msdn.microsoft.com](http://social.msdn.microsoft.com), Microsoft, 04 03 2008. <https://social.msdn.microsoft.com/Forums/vstudio/en-US/d60f0793-cc92-48fb-b867-dd113dabcd5c/how-to-find-the-child-processes-associated-with-a-pid?forum=csharpgeneral>
- [27] Downloaded: 1st September 2014., `ManagementObjectCollection` class, [msdn.microsoft.com](http://msdn.microsoft.com), [http://msdn.microsoft.com/en-us/library/system.management.managementobjectcollection\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system.management.managementobjectcollection(v=vs.110).aspx)
- [28] Downloaded: 7th September 2014., Sysinternals Process Explorer, M. Russinovich, [technet.microsoft.com](http://technet.microsoft.com), 11 09 2014. <http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx>
- [29] Downloaded: 9th September 2014., YAPM, [yaprocmon.sourceforge.net](http://yaprocmon.sourceforge.net), <http://yaprocmon.sourceforge.net/>
- [30] Downloaded: 11th September 2014., Glances ,uploader: nicolargo, [pypi.python.org](http://pypi.python.org), <https://pypi.python.org/pypi/Glances>
- [31] Downloaded: 11th September 2014., username: Qasim, [askubuntu.com](http://askubuntu.com), Ubuntu, 10 05 2013. <http://askubuntu.com/questions/293426/system-monitoring-tools-for-ubuntu>
- [32] Downloaded: 13th September 2014., username: akzhan, Github, <https://github.com/bluepill-rb/bluepill>
- [33] Downloaded: 14th September 2014., G. Smith, [garrensmith.com](http://garrensmith.com), 24 09 2012. <http://www.garrensmith.com/2012/09/24/Staying-up-with-Unicorn-Upstart-Bluepill.html>
- [34] Downloaded: 20th September 2014., Java Class Library (JCL), Oracle, [http://en.wikipedia.org/wiki/Java\\_Class\\_Library](http://en.wikipedia.org/wiki/Java_Class_Library)
- [35] Downloaded: 21st September 2014., Process Class, [docs.oracle.com](http://docs.oracle.com), Oracle, <http://docs.oracle.com/javase/8/docs/api/java/lang/Process.html>
- [36] Downloaded: 21st September 2014., username: R. K., Stackoverflow, 02 10 2011. <http://stackoverflow.com/questions/7550392/kill-a-process-tree-on-windows-using-java>

- [37] Downloaded: 22nd September 2014., username: mikeslattery, Stackoverflow, 20 11 2012. <http://stackoverflow.com/questions/9877993/jna-query-windows-processes>
- [38] Downloaded: 22nd September 2014., J. Friesen, Java Native Access (JNA), Java World, 05 02 2008. <http://www.javaworld.com/article/2077828/java-se/open-source-java-projects-java-native-access.html?page=2>
- [39] Downloaded: 27th September 2014., python, python.org, Python, <https://www.python.org/>
- [40] Downloaded: 28th September 2014., Psutil, uploader: giampaolo, Github, 26 09 2014. <https://github.com/giampaolo/psutil>
- [41] Downloaded: 29th September 2014., Subprocess management, docs.python.org, <https://docs.python.org/3/library/subprocess.html>
- [42] Downloaded: 30th September 2014., username: EFraim, Stackoverflow, 02 03 2009. <http://stackoverflow.com/questions/604522/performing-equivalent-of-kill-process-tree-in-c-on-windows>
- [43] Downloaded: 2nd October 2014., Process Handles, msdn.microsoft.com, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms684868\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684868(v=vs.85).aspx)
- [44] Downloaded: 2nd October 2014., username: A. Yezutov, CreateJobObjects, Stackoverflow, 06 02 2012. <http://stackoverflow.com/questions/6266820/working-example-of-createjobobject-setinformationjobobject-pinvoke-in->
- [45] Downloaded: 2nd October 2014., CloseHandle, PInvoke.net, <http://www.pinvoke.net/default.aspx/kernel32.closehandle>
- [46] Downloaded: 3rd October 2014., Job Object security, msdn.microsoft.com, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms684164\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684164(v=vs.85).aspx)
- [47] Downloaded: 5th September 2014., Killtree alg., username: zhigang, Stackoverflow, 09 07 2010. <http://stackoverflow.com/questions/392022/best-way-to-kill-all-child-processes>
- [48] Downloaded: 15th September 2014., username: demerus, Github, 07 07 2013. <https://github.com/arya/bluepill/issues/160>
- [49] Downloaded: 24th August 2014., Nested Jobs image, msdn.microsoft.com, <http://i.msdn.microsoft.com/dynimg/IC534328.png>

- [50] Downloaded: 30th November 2014., Balazs Valyon, BSc Thesis source codes, <https://github.com/valyonbalazs/bscThesis>
- [51] Downloaded: 23rd September 2014., Java Native Access (JNA), uploader: junak-michal, <https://github.com/twall/jna>

# List of Figures

1.1	Process tree with a root(A) and child processes . . . . .	7
1.2	Nested Jobs process tree . . . . .	13
2.1	YAMP process tree at the left, Sysinternals at the right . . . . .	16
2.2	Ubuntu 14.04 with Glances 2.1.1 . . . . .	17
3.1	Binary process tree . . . . .	20
3.2	Two process application, two-way monitoring . . . . .	22
3.3	Job Objects test application architecture . . . . .	23
3.4	Killing a single child process on the lowest level . . . . .	24
3.5	Kill a single child in the middle of the tree, and the orphans after . . . . .	25
3.6	A 4 level high orphan process tree . . . . .	26
3.7	Kill the root process with Sysinternals Process Explorer . . . . .	26
3.8	Execute 'Kill process tree' command on a lowest node with Sysinternals Process Explorer . . . . .	27
3.9	YAMP different process kill methods . . . . .	28
3.10	YAMP to the left, Process Explorer to the right, after the root was killed . .	29
3.11	Upper half: terminal command process list, lower half: Glances process list .	31
3.12	Killed a process on the middle of the tree, orphans kept running . . . . .	32
3.13	UNIX: kill a process tree by process group ID . . . . .	33
3.14	UNIX: kill tree recursive algorithm . . . . .	33
3.15	UNIX: Bluepill restarting the process tree after stop . . . . .	34
3.16	Python two directional monitoring processes . . . . .	35
3.17	Python: the restarted child process . . . . .	36
3.18	Windows-Python: after the root process was killed . . . . .	36
3.19	UNIX-Python: after the root process was killed . . . . .	36
3.20	Windows Job Objects . . . . .	37



# List of Tables

2.1	Summary of the used monitoring tools . . . . .	18
3.1	Summary of my test applications . . . . .	19
4.1	Summary of the successrate of different API calls . . . . .	38
4.2	Summary of the successrate of the process group commands . . . . .	39
4.3	Summary of the successrate of Job Objects . . . . .	39