리눅스 운영체제에서 프로세스 자원 사용량을 효율적으로 오케스트레이션하기 위한 프레임워크

최상훈*, 김성진*, 조여름*, 박기웅**
세종대학교 시스템보안 연구실
csh0052@gmail.com*, sys.ryan0902@gmail.com*, sshh802@gmail.com*,
woongbak@sejong.ac.kr**

Framework for efficient ly orchestrating resource usage of processes in Linux Operating System

SangHoon-Choi*, Seong-Jin Kim*, YeoReum* Jo, Ki-Woong Park**
Dept. Information Security, Sejong University, Seoul, South Korea

요 약

컴퓨팅 시스템에서는 특정 프로세스가 반드시 필요한 상황이 아님에도 불구하고 메모리에 상주하며 구동되고 있는 경우가 존재한다. 이러한 상주형 프로세스는 해당 프로세스를 사용하지 않고 있을 때에도 리소스를 점유하며 구동되어서 리소스 활용 효율성을 저하시킨다는 문제가 있다. 또한, 어떤 상주형 프로세스들은 자신이 종료되는 것을 방지하는 메커니즘이 적용되어 있어서 종료되지 않은 상태로 컴퓨터의 리소스를 지속적으로 소모한다는 문제가 있다. 따라서 이러한 문제점들을 개선하기 위해 상주형 프로세스들의 리소스를 효율적으로 제어 및 관리할 수 있는 방안이 필요하다.

본 논문에서는 Linux Signal과 Control group namespace를 활용하여 리눅스 운영체제에서 구동 중인 프로세스의 자원을 효율적으로 오케스트레이션 할 수 있는 p-Fusebox 프레임워크를 제안한다. 우리의 실험결과에 따르면 p-Fusebox를 사용하였을 때 1시간 동안의 메모리 누적사용량이 약 38% 감소함을 확인할 수 있었다.

1. 서론

컴퓨터 시스템의 주요 동향 중 하나는 한정된 컴퓨팅 리소스를 효율적으로 활용할 수 있는 컴퓨팅 환경을 제공하는 것이다. 컴퓨팅 리소스의 효율성을 극대화한 대표적인 분야로는 클라우드 컴퓨팅 서비스가 있다.

클라우드 컴퓨팅 서비스는 고사양의 물리 머신을 가상화를 통해 논리 자원으로 나누어 다수의 사용자에게 서비스를 제공한다. 그리고 각 논리 자원의 크기를 사용자의 필요에 따라 또는 처리하고 있는 워크로드에 따라 유연하게 Scale up 하거나 Scale down (오케스트레이션) 하는 기능을 지원하기 때문에 자원이 많이 필요한 상황에서는 충분한 리소스를 보장해 주고, 자원이 많이 필요 없는 상황에서는 필요한 만큼 적은 리소스를 할당하여 리소스를 효율적으로 활용할 수 있도록 한다. 더 나아가 클라우드 컴퓨팅 분야에서 논리 자원을 더욱 효율적으로 오케스트레이션하기 위한 획기적인 전략들이 활발하게 연구되어지고 있다 [1][2][3][4]. 이와 같이 컴퓨팅 리소스의 효율적인 관리는 주로 고사양의 물리 머신에 초점이 맞추어 연구되어지고 있다. 하지만 여전히 개인적인 용도로 사용하는 컴퓨팅 시스템에서 컴퓨팅 리소스의 효율성 극대화에 관한 연구는 고사양의 물리 머신에 대한 연구에 비해 상대적으로 부족하다. 저사양 물리 머신에 해당하는 컴퓨팅 시스템에서는 특정 프로세스가 반드시 필요한 상황이 아님에도 불구하고 메모리에 상주하며 구동되고 있는 경우가 다수 존재한다. 이러한 상주형 프로세스는 해당 프로세스를 사용하지 않고 있을 때도 리소스를 점유하며 구동되어서 리소스 활용 효율성을 저하시킨다는 문제가 있다. 또한, 어떤 상주형 프로세스들은 자신이 종료되는 것을 방지하는 kill-protection 메커니즘이 적용되어 있어서 사용자가 임의로 종료하기 어렵고, 이 프로세스들은 종료되지 않은 상태로 컴퓨터의 리소스를 지속적으로 소모한다는 문제가 있다. 따라서 위에서 언급한 문제점들을 개선하기 위해 상주형 프로세스들의 리소스를 효율적으로 제어 및 관리할 수 있는 방안이 필요하다. 본 논문

에서는 리눅스 운영체제 환경에서 상주형 프로세스의 리소스를 효율적으로 제어 및 관리할 수 있는 p-Fusebox 프레임워크를 제안한다.

2. 배경지식

이번 장에서는 p-Fusebox 프레임워크의 설계(디자인)에 대한 이해를 돕기 위해 p-Fusebox 프레임워크의 기반 기술인 Linux namesapce, control group 그리고 Linux Signal에 대해 소개한다.

- namespace [5]

Namespace는 시스템 리소스를 추상화하여서, 해당 namespace안에 있는 프로세스들에게 격리된 리소스 공간을 가진 것처럼 보이도록 만든다. 해당 namespace에서 리소스의 변화는 해당 namespace의 멤버 프로세스들만 알 수 있고, 멤버 프로세스가 아닌 namesapce 밖의 프로세스들은 알 수 없다. 네임스페이스의 한 가지 사용 예시는 컨테이너를 구현하는 것이다. 현재 Linux에서 제공하는 네임스페이스의 유형은 7가지로 다음과 같다. Control Group(Cgroup), IPC, Netrowk, Mount, PID, User, UTS. 본 논문에서 제안하는 p-Fusebox 프레임워크는 namespace중 Cgroup을 기반 기술로 사용하여 구현되었다.

- Control Group(Cgroup) [5]

Cgroup은 다양한 유형의 리소스(메모리, CPU 등)를 제한하고 모니터링 할 수 있는 계층적 그룹으로 프로세스를 구성할 수 있도록 하는 커널 기능이다. Cgroup namespace는 /proc/[pid]/cgroup 와 /proc/[pid]/mountinfo를 통해 표시되는 프로세스의 cgroup의 범위를 가상화한다.

- Linux Signal [5]

Signal은 프로세스나 동일 프로세스 내의 특정 스레드로 전달되는 비동기식 신호이다. Signal의 일반적인 용도는 프로세스를 인터럽트 하거나 일시중단 또는 종료하는 것이다. Signal이 전송되면 운영체제는 Signal을 전달하기 위해 타겟 프로세스의 정상적인 흐름을 인터럽트 한다. 프로세스가 signal handler를 등록해 놓은 경우 해당 루틴이 실행되며, 그렇지 않으면 default signal handler가 수행된다.

3. 리소스 오케스트레이션을 위한 p-Fusebox

본 논문에서 리눅스 운영체제에서 구동 중인 프로세스의 자원을 효율적으로 오케스트레이션 할 수 있는 p-Fusebox 프레임워크를 제안한다.

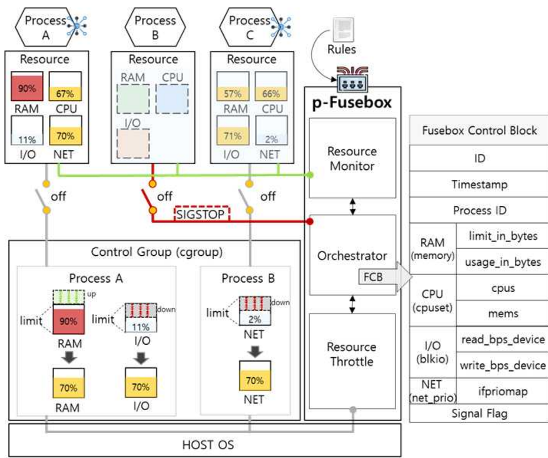
3.1 p-Fusebox 설계

p-Fusebox는 프로세스의 자원을 제어하기 위해 Linux Signal과 Control group namespace를 활용한다. 리눅스 운영체제에는 SIGSTOP(19)을 활용하여 유저 프로세스 뿐만 아니라 데몬 프로세스까지 모두 일시정지 상태로 만들고 프로세스의 자원을 강제로 회수할 수 있다. SIGSTOP을 통해 일시 정지된 프로세스는 SIGCONT(18)를 통해 다시 재개 할 수 있다. 하지만 SIGSTOP을 네트워크 프로세스나 다른 프로세스와 통신하는 IPC 프로세스에 활용하는 데에는 어려움이 따른다. 네트워크 프로세스나 다른 프로세스와 통신하고 있는 프로세스를 SIGSTOP 한 상태로 너무 오랫동안 정지시켜 두면 연결이 끊어질 위험이 있다. 프로세스가 갑자기 SIGSTOP으로 중지되면 정지된 프로세스의 통신 대상은 해당 프로세스가 중단되었다는 사실을 알지 못하고, 통신 대상이 응답을 받기까지 허용할 수 있는 최대 대기 시간보다 오랫동안 중단된 프로세스로부터 응답을 받지 못한다면 통신 대상은 중단된 프로세스와의 연결을 종료할 것이다. 또한, SIGSTOP의 또 다른 문제는, 어떤 프로세스는 자신의 child process가 SIGSTOP 된 것을 감지하고 해당 자식 프로세스를 SIGCONT 시키거나, 종료해 버릴 수도 있다는 것이다. 이는 특정 프로세스에 SIGSTOP을 한 사용자가 의도하지 않은 결과를 초래할 수 있다. 따라서 p-Fusebox는 프로세스의 특징에 따라 linux Signal과 cgroup을 적절하게 사용하여 상주형 프로세스의 리소스를 관리할 수 있도록 한다.

3.2 p-Fusebox 구현

p-Fusebox 프레임워크의 구조는 (그림 1)과 같다. p-Fusebox는 세 가지 모듈로 구성되어있다.

첫째, Resource Monitor는 사용자가 지정한 Process의 Resource 사용량과 Limit을 실시간으로 모니터링하는 역할을 수행한다. Resource Monitor는 구동 중인 프로세스의 리소스 자원을 모니터링하여 p-Fusebox Control Block(FCB)를 생성한다. FCB는 다음과 같은 5가지의 리소스 정보(RAM, CPU, I/O, NET, SignalFlag)를 활용한다. 먼저, RAM의 경우 현재 프로세스의 메모리 제한에 해당하는 `memory.limit_in_bytes` 와 메모리 사용량에 해당하는 `memory.usage_in_bytes`을 활용하며, CPU의 경우 프로세스가 사용할 수 있는 코어 수에 해당하는 `cpuset.cpu`와 프로세스가 활용할 수 있는 메모리 노드에 해당하는 `cpuset.mems`를 활용한다.



(그림 1) p-Fusebox구조

I/O의 경우 프로세스의 읽기/쓰기 속도를 제어할 수 있는 `blkio.throttle.read_bps_device`, `blkio.throttle.write_bps_device` 를 활용하며, Net의 경우 프로세스의 네트워크 우선순위를 지정할 수 있는 `net_prio.ifpriomap`를 활용한다. 그리고 Signal Flag는 타겟 프로세스가 다른 프로세스와의 통신 여부를 판단하는 데 사용된다. 둘째, Orchestrator는 FCB 의 정보를 활용하여 프로세스의 점수를 측정한다. 프로세스의 점수 측정이 필요한 이유는 프로세스가 강제로 종료되는 현상을 해결하기 위함과 불필요한 메모리 할당을 해제시켜야 하기 때문이다. 프로세스의 점수를 측정하는 알고리즘은 (그림 2)와 같다.

Algorithm 1 Resource Throttle
Input: Usage(x),Limit(y)
1: **function** SCORE_CALCULATION(x, y)
2: $score \leftarrow (x_n/y) + ((1-(x_{n-1}/x_n))*2)$
3: $size \leftarrow abs((x_n - x_{n-1}) + ((y - x_n)*0.3))$
4: **if** $Score > 1.0$ **then**
5: Limit Scale Up [$size$]
6: **end if**
7: **if** $Score < 0.5$ **then**
8: Limit Scale Down [$size$]
9: **end if**
10: **end function**
11: **for** $iteration = 1, 2, \ldots$ **do**
12: $Score_Calculation(Process\ Resource\ limit, Process\ Resource\ usage)$
13: **end for**

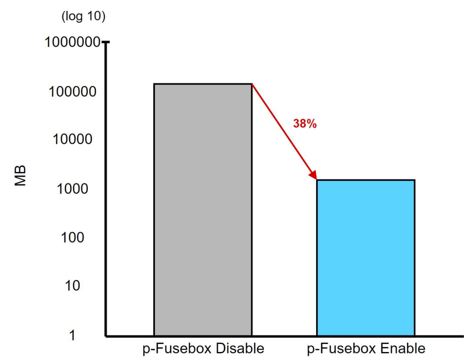
(그림 2) p-Fusebox의 프로세스 점수 계산 알고리즘

프로세스의 점수를 평가하는데 활용되는 데이터는 현재 리소스 사용량과 리소스의 최대 사용량이다. Orchestration는 점수를 계산하여 점수가 1.0 을 넘기게 되면 프로세스의 최대 리소스 사용량 증가가 필요하다고 판단한다. 만약에 점수가 0.5 이하로 계산된다면 리소스 최대 사용량을 감소시켜야 한다고 판단하여 불필요한 메모리 할당을 해제할 수 있도록 도와준다. 이때 최대 리소스 사용량은 size를 통해 계산된 크기만큼 감소한다. 만약에 FCB의 Signal Flag가 Ture인 경우 점수를 계산하지 않고 Signal를 통해 직접적으로 제어한다. 마지막으로, Resource Throttle 는 Orchestrator를 통해 계산된 점수를 활용하여 동적으로 리소스를 제어하는 역할을 수행한다. Resource Throttle는 사용자가 등록한 프로세스들을 p-Fusebox cgroup에 별도의 공간을 할당하고 각 프로세스의 리소스를 병렬적으로 제어한다. Resource Throttle는 프로세스 cgroup에 접근하여 리소스의 최대 사용량을 동적으로 제어하는 역할을 수행한다.

4. p-Fusebox 평가

본 장에서는 2가지의 실험을 통해 p-Fusebox의 리소스 효율성을 평가한다. 첫째, p-Fusebox의 누적 메모리 사용량을 비교 측정한다. 우리는 sysbench[6] 를 통해 메모리 벤치마크를 수행하는 과정에서 sysbench 프로세스에 할당된 누적 메모리의 사용량의 크기를 측정한다. sysbench는 크로스 플랫폼, 멀티스레드 벤치마크 도구이다. 둘째, p-Fusebox의 Resource Throttle 기능을 평가한다. 우리는 잘못된 리소스 제어로 인해 프로세스가 종료되는 문제를 해결하기 위해 동적으로 리소스 제한을 제어하는 Resource Throttle를 제안하였다. 우리는 Resource Throttle를 평가하기 위해 sysbench를 통해 메모리 벤치마크를 수행하는 과정에서 프로세스가 소모하는 메모리에 양에 따라 동적으로 메모리 할당량을 제어할 수 있는지 측정한다. 본 논문의 성능 평가는 Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz CPU를 사용하고, 8GB RAM을 사용하는 Ubuntu 18.04 (64-bit) 운영체제에서 수행되었다.

4.1 p-Fusebox 메모리 누적사용량 평가

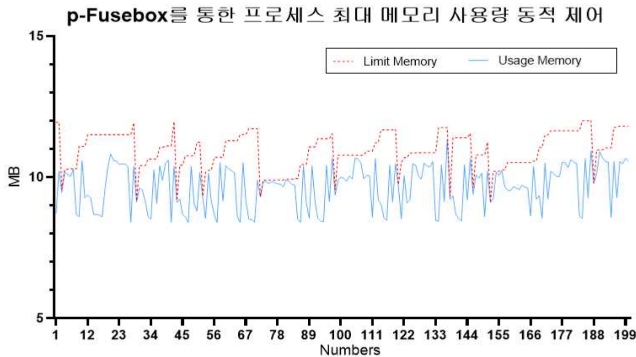


(그림 3) 메모리 누적사용량 비교측정

p-Fusebox의 성능을 평가하기 위해 우리는 sysbench를 활용하였다. 우리는 sysbench를 일반적으로 사용하였을 때와, p-Fusebox를 통해 리소스를

제어하였을 때의 1시간 동안 메모리 누적사용량을 비교하였다. 본 실험에서는 1초 간격으로 메모리 사용량을 수집하였다. 실험결과는 (그림 3)과 같다. 실험결과 p-Fusebox를 사용하였을 때 누적 메모리 사용량이 38% 감소하는 것을 확인할 수 있다.

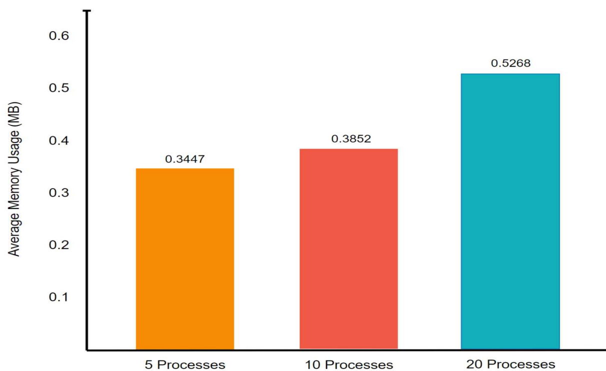
4.2 메모리 동적 할당 성능 평가



(그림 4) 프로세스 점수를 활용한 최대 메모리 제어

p-Fusebox 메모리 동적 제어에 대한 실험결과는 (그림 4)와 같다. p-Fusebox는 타겟 프로세스의 점수를 측정하여, 실시간으로 limit을 동적으로 제어함으로써 프로세스의 메모리 사용량이 증가하더라도 프로세스가 메모리 부족으로 강제 종료되지 않도록 함을 알 수 있다.

4.3 p-Fusebox 오버헤드 평가



(그림 5) 프로세스 모니터링에 따른 리소스 사용량 측정

p-Fusebox 프레임워크의 오버헤드 측정 결과는 (그림 5)와 같다. p-Fusebox가 동시에 다수의 프로세스를 제어할 때 제어 대상 프로세스 수의 증가에 따른 오버헤드를 측정하였다. 오버헤드는 1초 간격으로 1시간동안 p-Fusebox가 소모한 메모리 사용량을 측정하여 평균을 계산하였다. 실험 결과 5 개의 프로세스를 제어할 때, 10개의 프로세스를 제어할 때, 그리고 20개의 프로세스를 제어할 때 p-Fusebox의 메모리 오버헤드는 각각 0.3447 MB, 0.3852MB, 0.5268 MB로 측정되었다. 이와 같은 결과는 p-Fusebox의 메모리 오버헤드는 제어를 통해 감소되는 누적 메모리 사용량과 비교하여 적다는 것을 의미한다.

5. 결론

본 논문에서는 리눅스 운영체제에서 상주형 프로세스의 리소스 사용량을 효율적으로 오케스트레이션하기 위한 p-Fusebox를 제안하였다. 우리가 제안한 p-Fusebox는 Linux Signal(SIGSTOP, SIGCONT)과 Control group namespace를 활용하여 상주형 프로세스들의 리소스 사용을 모니터링하여 p-Fusebox Control Block(FCB)를 생성하고 FCB의 정보를 기반으로 프로세스의 점수를 측정한 후, 이 점수를 통해 프로세스에 허용하는 메모리 할당량을 동적으로 조절하여 프로세스가 강제로 종료되는 현상을 방지하면서도, 프로세스의 자원 사용량을 극소화한다. 결과적으로 누적 메모리 사용량이 약 38% 감소함을 확인할 수 있었다.

Acknowledgement

본 연구는 2021년도 과학기술정보통신부의 재원으로 정보통신방송기술개발사업 (No. 2019-0-00273, 1%)과 한국연구재단 (No. NRF-2020R1A2C4002737, 99%)지원 지원을 받아 수행된 연구임

참고문헌

[1] Rodriguez, Maria, and Rajkumar Buyya. "Container orchestration with cost-efficient autoscaling in cloud computing environments." Handbook of research on multimedia cyber security. IGI Global, 2020. 190-213.

[2] Zhong, Zhiheng, and Rajkumar Buyya. "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources." ACM Transactions on Internet Technology (TOIT) 20.2 (2020): 1-24.

[3] Zhou, Naweiluo, et al. "Container orchestration on HPC systems." 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). IEEE, 2020.

[4] Rovnyagin, Mikhail M., et al. "Ml-based heterogeneous container orchestration architecture." 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus). IEEE, 2020.

[5] Linux Kernel, https://www.kernel.org

[6] Kopytov, Alexey. "Sysbench manual." MySQL AB (2012): 2-3.