Self-prevention: 바람직하지 못한 시스템 상태를 피하기 위한 예방 및 적응형 커널 서브시스템 관리

김영필○ 유혁
고려대학교 컴퓨터학과
ypkim○@os.korea.ac.kr, hxy@os.korea.ac.kr

Self-prevention: In-advance and adaptive kernel subsystem management to avoid undesirable state

Young Pil Kim○ Hyuck Yoo
Korea University, Dept. of Computer Science and Engineering

요 약

본 연구에서는 관리의 복잡도를 줄이기 위한 대상으로써 운영체제 커널을 목표로 하였다. 특히, 운영체제의 핵심 기능들이 구현된 소프트웨어인 운영체제 커널이 본 논문에서 다루고자 하는 영역이다. 본 연구에서는 커널의 실제적인 서비스를 제공하는 커널서브시스템의 독립성을 살리고 시스템 내부의 변화에 따른 관리 복잡도를 줄이기 위해, 커널 내 구성요소들 간의 상호 관계 조율에 필요한 개념과 그 구조를 제안하고 있다. 본 논문에서 정의한 Self-prevention은 커널 내의 자율적인 상호 관계 조율을 위한 모든 방안들의 총칭이다. 이러한 self-prevention의 구현을 위해 커널 내의 핵심 관리부의 관여를 줄이고 예상치 못한 동작에 대한 처리를 서브시스템의 자율에 맡길 수 있도록 하기 위해서 3가지의 부가적인 커널 컴포넌트들(sampler, analyzer, preventer)을 정의하였고, 그 상호관계들을 서술하고 있다. 본 연구의 기여도는 크게 두 가지로써 먼저, self-managing 혹은 자율형 컴퓨팅 분야의 새로운 시도라는 점과 두 번째로 서브시스템의 자율성과 독립성을 유지하여 운영체제의 기능 확장성에 도움이 될 수 있다는 것이다.

1. 서 론

하드웨어 성능은 여전히 폭발적으로 증가하고 있으며, 이는 컴퓨팅 인프라의 확산을 낳았다. 또한, 컴퓨팅 환경의 수혜자들인 사용자들이 증대되고, 이들의 컴퓨팅 서비스에 기능적인 요구나 질적인 요구는 날로 증대 되어 이를 지원하기 위한 컴퓨팅 시스템의 복잡도를 증가시키고 있다. 또한 인터넷의 확산이나 휴대 전화의 보급으로 인해 네트워킹 인프라가 구축되어 컴퓨팅 자원의 접근마저 빈번해져, 서버와 같은 제공자(provider)에 해당하는 소프트웨어의 복잡도는 가속화 되고 이를 관리하는 비용은 그 복잡도에 비례하게 되었다.[6]

본 논문에서 다루고자 하는 것은 이러한 관리의 복잡도를 줄이기 위한 방안이며, 효과적인 관리를 위해 인간이 관여하는 정도를 최소화 하려 한다. 이러한 문제를 다루는 기존 연구 영역은 Self-managing[2], 자율형 컴퓨팅(Autonomic 혹은 Autonomous computing)[1]이다. 자율형 컴퓨팅은 이러한 소프트웨어 시스템의 복잡도를 해결하기 위해 시스템 스스로가 시스템을 제어하고 조직하는 능력을 갖도록 하고 있다[6]. 이는 소프트웨어 개발의 모든 단계(시스템소프트웨어에서 미들웨어 및 응용프로그램에 이르기까지)에 걸쳐 영향을 주는 새로운 시도에 해당한다.

본 연구에서는 관리의 복잡도를 줄이기 위한 대상으로써 운영체제 커널을 목표로 하였다. 다양한 시스템 소프트웨어 가운데, 운영체제는 자율형 컴퓨팅 개념을 적용하기에 가장 적절한 영역에 해당한다. 운영체제는 프로세서와 물리 메모리, 주변장치들을 비롯한 각종 하드웨어들과, 다수의 사용자들의 행위(behavior)에 따라 생성되거나 제거되는 다양한 작업들을 그 목적에 맞는 정책에 따라서 관리 및 제어해야 한다. 특히, 운영체제의 핵심 기능들이 구현된 소프트웨어인 운영체제 커널(kernel)이 본 논문에서 다루고자 하는 영역이다.

좀 더 정확히 이야기 한다면, 본 연구에서 초점을 두고 있는 것은 커널의 핵심 관리부(core part)와 그 대상이 되는 커널 서브시스템(kernel subsystem)들 간의 상호작용과 예상하지 못한 행위(unexpected behavior)에 대한 자율적인 처리이다. 커널은 그 자체로 독립적인 관리 소프트웨어이다. 과거와는 달리 현재 근대적인 시스템들은 시스템 기능의 확장을 염두하고 있기 때문에[3][15], 시스템은 보통 핵심 관리 부분과 상호 유기적인 하위의

이 논문은 2004년도 정부 (과학기술부) 의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No. R01-2004-000-10588-0)

여러 서브시스템들로 구성된다. 기존에 단일한 구조(monolithic systems)에서 하부 기능 간에 밀접한 관계를 유지하던 것과는 달리 다양한 확장 가능한 구조

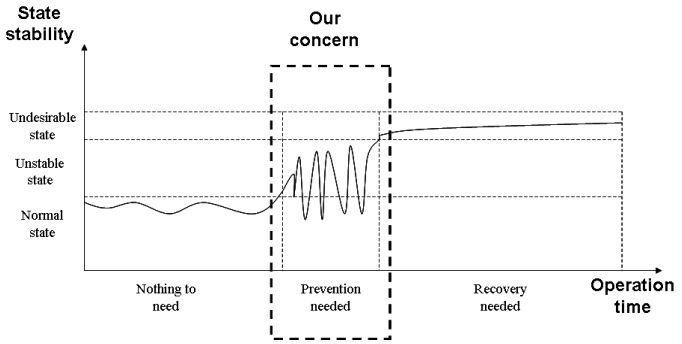

그림 1 커널의 상태 변화와 그에 따른 안정성 분류

(extensible system, micro kernel)들에서는 각 서브시스템들이 독립적인 형태로 써 모듈(module)이나 컴포넌트(component)의 형태로 존재한다. 서브시스템이 독립적인 형태를 띄어 갈수록 핵심 관리부와 주고받아야 하는 정보들은 많아지고, 이는 관리의 복잡도가 증가하는 현상을 가속화 시킨다. 게다가, 다양한 커널 서비스(kernel service)들의 구체적인 수행을 담당하고 있는 각 서브시스템들은 필연적으로 변화한다. 다른 말로 하자면, 서브시스템 내부의 상태(state)가 끊임없이 바뀐다는 것이다. 이러한 상태들의 변화 요인은 정상적인 경우에는 문제가 없으나, 비정상적인 경우에 그 처리를 핵심 관리부가 해야 한다는 점이 문제다. 또한 핵심 관리부가 문제 해결을 위해서는 서브시스템의 컨텍스트(context)를 이해하고 필요한 정보들을 전달 받아야 하나, 이를 처리하는 시간은 문제 해결을 위한 시간에 포함되지 않으므로 처리 시간만 가중 시킬 뿐이다.

본 연구에서는 커널의 실제적인 서비스를 제공하는 서브시스템의 독립성을 살리고 시스템 내부의 변화에 따른 관리 복잡도를 줄이기 위해, 커널 내 구성요소들(핵심 관리부와 그 서브시스템)간의 상호 관계 조율에 필요한 개념과 그 구조를 제안하고 있다. 본 논문에서 정의한 Self-prevention은 커널 내의 자율적인 상호 관계 조율을 위한 모든 방안들의 총칭이다. 이러한 self-prevention의 구현을 위해 커널 내의 핵심 관리부의 관여를 줄이고 예상치 못한 행위에 대한 처리를 서브시스템의 자율에 맡길 수 있도록 하기 위해서 3가지의 부가적인 커널 컴포넌트들(sampler, analyzer, preventer)을 정의하였고, 그 상호관계들을 서술하고 있다.

논문의 구성은 다음과 같다. 2장에서는 Self-prevention의 기본적인 개념과 이를 구현하기 위해 필요한 하위 컴포넌트들의 내용을 서술한다. 3장에서는 코어 시스템(core system)의 전체적인 설계를 보인다. 4장에서는 관련연구를 소개하고, 마지막으로, 5장에서 결론을 제시한다.

2. Self-prevention 개관

본 논문에서는 운영체제 커널의 관리 복잡도를 줄이고 자율성을 부여하기 위해서 Self-prevention 개념을 도입하였다. 따라서 본 섹션에서는 Self-prevention의 개념을 설명하고, 그 필요성과 적용 가능성에 대해서 논의한다. 그리고 구체적으로 기존의 커널 시스템에 Self-prevention을 도입하기 위해서 필요한 하위 구성요소들에 대해서 서술 할 것이다.

2.1. Self-prevention의 정의와 필요성

본 논문에서 정의하는 Self-Prevention이란, 커널이 스레싱(thrashing)과 같은 바람직하지 못한 상태(undesirable state)로 진입하기 전에 미리 예방적인 행동(preventive action)을 수행하는 것을 말한다. Self-prevention의 기본 가정은 커널 내부의 상태(state)는 끊임없이 변화한다는 것이다. 이러한 변화는 시스템이 의도한 바로 진행될 수 있고, 그렇지 못한 상황으로 진행될 수 있다. Self-prevention이 다루고자 하는 상황은 의도하지 않은 상황이며, 이러한 상황을 구분 짓기 위해서 변화하는 상태를 안정성(stability)에 따라서 몇 가지로 분류하였다. 각각은 정상 상태(Normal state), 불안정 상태(Unstable state), 그리고 바람직하지 못한 상태(Undesirable state)이다. (그림1) 정상 상태는 커널의 동작이 의도한 바대로 수행되는 경우를 뜻하며, 이런 경우 부가적인 조치가 필요하지 않다. 그러나 어떠한 원인(에러 혹은 비정상적인 동작)에 의해 의도하지 않은 상황이 되었다면 조치가 가능한 경우 부가적인 동작이 요구되며, 이 경우를 바람직하지 못한 상태로 정의 하였다. 불안정 상태는 정상 상태가 바람직하지 못한 상태로 변화하는 과도기적인 상황을 의미하는 것으로, 본 논문에서 다루고자 하는 상황이다. 즉 예방이 가능한 상황을 나타낸다.

불안정 상태를 대상으로 한 이유는, 커널이 어떠한 상황에 대해서 충분히 제어가 가능하고 조치를 취할 수 있고, 그러한 필요성이 있는 유일한 상황이기 때문이다. 바람직하지 못한 상태는 불안정 상태에 비해 확실히 문제가 발생했음을 확인 할 수 있지만, 발생한 문제를 처리 할 수 있는가와 특정한 조치를 취할 수 있는 기회를 얻을 수 있는가를 보장하기 어렵다. 일례로, 대부분의 시스템에서는 처리할 수 없는 문제가 발생했을 때, 기껏 스택(stack)등의 메모리 내용을 출력하고 커널 패닉(kernel panic) 메시지만을 보여주고 정지되어 버릴 뿐이다.[8] 이에 반해, 불안정 상태는 제어가 가능한 시점이므로 최소한 특정한 조치를 취할 수 있는 기회는 보장되는 것이므로, 본 연구에서 더욱 비중을 두게 되었다.

이러한 바람직하지 못한 상태의 발생 원인은 커널의 상태가 끊임없이 변화하기 때문이다. 사용자로부터의 상호작용과 하드웨어 주변 장치로부터의 I/O 그리고 네트워크를 이용하여 비동기적으로 발생하는 원거리 이벤트들 등으로 인해 커널의 내부 상태는 변화한다. 본 논문에서는 커널이 이러한 변화하는 상태를 시스템 스스로 관리하기 위해서 필요한 정형화된 체계 확립을 목표로 하고 있다. 이후 절에서는 Self-prevention 개념을 적용할 수 있는 커널 서비스 영역에 대해서 논의하고, 개념을 구체화하기 위해 필요한 하위 구성요소들에 대해서 서술하겠다.

2.2. 적용 분야

본 섹션에서는 Self-prevention의 적용 및 활용 가능성을

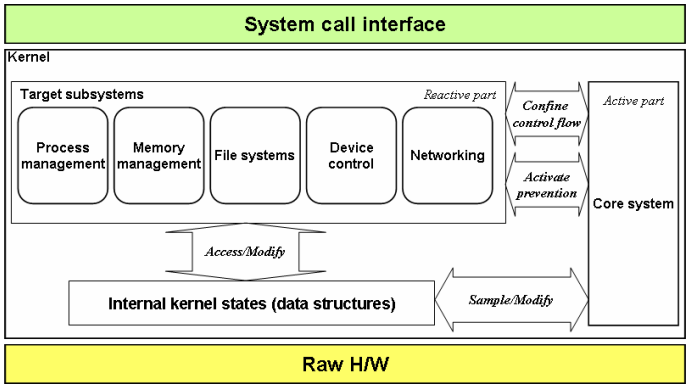

그림 2 Self-prevention의 개념이 적용된 커널 구조

살펴본다. 앞서 정의한 대로 Self-prevention은 커널 내에서 바람직하지 못한 상태로 진입하기 전에 예방책을 도입하자는 개념이다. 이는 커널이 기존에 수동적(reactive) 으로 동작하는 것이 아니라, 좀 더 능동적으로 관리하는 것이 요구된다. 이렇게 Self-prevention을 개념을 적용하여 능동적인 관리를 할 수 있는 커널 서비스의 몇 가지를 서술하면 다음과 같다.

* 버퍼 관리(buffer management) - 커널 내에 존재하는 자료구조 가운데 버퍼 풀과 같이 한정된 크기를 가지고 있으며, 그 버퍼에 대한 데이터 유입 속도와 소비 속도 간에 차가 큰 경우, 즉 스레싱이 발생하는 경우에는 오버플로(overflow)가 발생할 가능성이 있다. 대표적인 예가 소켓 버퍼 오버플로(socket buffer overflow)[4]로써, NIC(Network Interface Card)으로부터 빠르게 패킷이 유입되는 상황에서는 버퍼의 소비자인 응용프로그램이 소비할 기회를 갖지 못하게 되어 패킷이 유실되는 상황이 발생한다. 이러한 경우에 Self-prevention을 적용하여 유입속도와 버퍼의 상황을 고려하여 미리 대처한다면, 버퍼 오버플로가 발생하여 스레싱 상황에 놓이는 바람직하지 못한 상태를 피할 수 있다.

* 데드락 회피(deadlock avoidance) - 실시간 시스템과 같이 우선순위 기반의 자원 점유방식을 사용하는 시스템에서는 데드락(deadlock)이나 우선순위 역전현상(priority inversion)[9]과 같은 문제가 발생할 수 있다. 특히 크리티컬(critical)한 시스템 일수록 데드락에 대한 처리가 중요하다. 데드락에 대한 처리는 사전에 미리 그 발생을 방지하려는 회피(avoidance) 전략[10]과 발생한 후에 이를 복구하는 복구(recovery) 전략[11]이 존재하며, 데드락 자체를 바람직하지 못한 상태로 본다면, Self-prevention 개념은 다양한 회피 전략을 적용할 수 있는 커널 내 틀이 될 수 있을 것이다.

* 전력 관리(power management) - 전력관리는 예방책이 필요한 대표적인 사례이다. 시스템이 전력을 관리하기 위해서 프로세서나 IO 장치들을 APM(Advanced Power Management)이나 ACPI(Advanced Configuration and Power Interface)와 같은 전력 관리 방법을 이용하여 성능 대비 전력 소모량(performance/watt)을 극대화하기 위해서는 전력 잔류량과 예상 소모량을 감안하여 미리 동작 모드를 전환하는 전략이 필요하다.[12] 따라서 Self-prevention을 적용하여 전력 소모량을 기준으로 삼아, 잔류 전력이 부족한 상황인 바람직하지 못한 상태에 이르기 전에 미리 동작 모드를 전환하는 방법이 효과적일 수 있다.

* 보안(security) - 고속 그리고 고 대역의 네트워크와 인터넷의 확산은 서비스의 접근 가능성을 높였으나, 반대로 보안에 대한 위협수준도 크게 증대되고 있다. 시스템 수준에서 제공하는 대부분의 보안 정책들은 예방을 목적으로 하고 있으며 대표적인 예가 침입 탐지(intrusion detection)[13]이다. Self-prevention은 침입 탐지 시에 감시(monitoring)와 인터포지션(interposition)을 효과적으로 사용할 수 있는 기반이 되며, 바람직하지 못한 상태는 시스템에서 보호하고자 하는 자원의 오염(contamination)이나 우선순위(privilege)를 무시한 부정 접근(abnormal access)상황이 될 수 있다.

이 외에도 커널과 같은 시스템 내에서 예방이 요구되는 영역은 다양하다. 따라서 self-prevention의 적용 영역은 광범위하다. 이후 절에서는 self-prevention 개념을 구체화시키기 위해 필요한 구성요소들을 살펴보겠다.

2.3. Self-prevention의 서브 컴포넌트: 코어 시스템(core system)과 타겟 서브시스템(target subsystem)

Self-prevention은 커널이 바람직하지 못한 상태로 진입하기 전에 능동적으로 미리 예방적인 조치를 취하는 것이다. 이를 위해서 커널을 크게 능동적(active)인 요소와 수동적(reactive)인 요소로 분화하여 크게 두 가지의 주요 커널 컴포넌트를 정의하였다. 능동적인 요소는 자율적으로 예방 조치를 취하는 커널의 컴포넌트이며 수동적인 요소는 커널의 기능을 수행하며 커널 state를 변화시키는 커널 서브시스템이다. 본 연구에서는 전자를 코어 시스템으로 칭하였고, 후자를 타겟 서브시스템으로 칭하였다. 즉, Self-prevention은 코어 시스템이 타겟 서브시스템을 주시하며 타겟 서브시스템이 변화시키는 커널 상태가 바람직하지 못한 상태에 진입하기 전에 능동적인 예방 조치를 취하는 것이다.(그림 2)

그림을 통해 알 수 있듯이, self-prevention의 주체는 코어 시스템이며 그 대상은 커널의 서브시스템과 그 서브시스템이 관여하는 커널의 상태 (커널 내 자료구조들)이다. 코어 시스템은 각 서브시스템이 접근하고 조작하는 커널 상태들을 샘플링(sampling)하고 스냅샷(snapshot)을 수집하여 이를 바탕으로 예방이 필요한가를 판단한다. 필요하다면 코어 시스템은 해당하는 서브시스템의 동작을 제어하고, 예방 조치를 적용한다. 커널 상태의 샘플링 된 스냅샷은 과거의 데이터이며, 이를 통해서 바람직하지 못한 상태를 판단하므로, 예방이 되며, 커널 내에서 코어 시스템과 서브시스템간의 상호작용만이 일어나므로 자율적인 커널의 관리가 된다.

3. Core System

Self-prevention개념이 구체화되기 위해서는 커널이 바람직하지 못한 상태로 진입하기 전에 미리 예방적인 동작을 수행하여야 한다. 이러한 예방적인 동작 수행의 역할을 담당하는 것은 코어 시스템이며, 이를 구체화할 때 요구되는 것은 크게 3가지로 나눠볼 수 있다. 첫 번째로, 커널의 상

태를 인지하는 방법이 필요하며, 두 번째로 인지된 커널의 상태를 보고 예방적인 동작이 수행될 필요성을 판단하는 방법이 필요하다. 마지막으로, 적절한 예방적인 동작들을

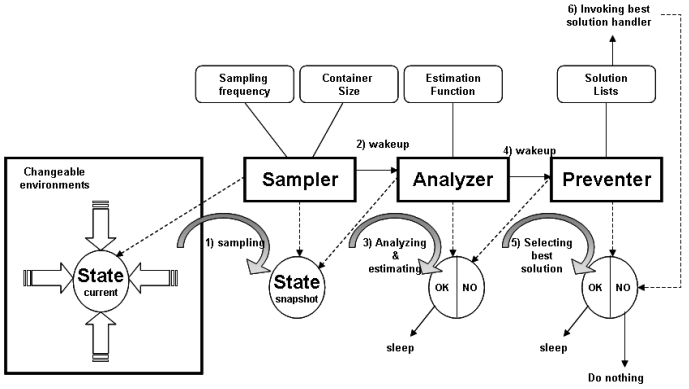

그림 3 코어 시스템의 3가지 구성요소들과 그 상호 작용

적용하는 방법이 필요하다. 이 세 가지의 방법들은 단계별로 쪼갤 수 있고, 각 단계 내에서는 독립적이기 때문에 개별적인 구성요소로써 분화시킬 수 있다. 그림 3은 Self-prevention을 구체화하기 위해서 앞서 언급한 3가지의 동작들을 분화된 구성요소 정의하고 각 구성 요소들 간의 상호 작용들을 표현한 것이다.

정의된 구성요소는 각각 sampler, analyzer, preventer이며 각각의 역할을 구체적으로 살펴보면 다음과 같다.

3.1. Sampler

Sampler는 커널의 상태를 인지하고, 관심 있는 표본을 수집하는 역할을 수행한다. Self-prevention에서 가정하고 있는 것은 커널의 상태가 끊임없이 변화한다는 것이다. 끊임없는 변화 속에서 바람직하지 못한 상태나 불안정 상태를 판단하기 위해서는 기본적으로 커널의 상태를 인지하고 관심 있는 표본들을 수집할 방법이 필요하다. Sampler는 커널의 상태를 모니터링 하여 그 시점의 상태를 스냅샷으로 생성한다. 수집된 커널 상태 스냅샷은 이후 단계에서 분석의 근거 자료가 된다.

1) 커널의 상태 인지

본 논문에서 정의하고 있는 커널의 상태는 다음과 같다.

- 바람직하지 못함(undesirableness)의 판단에 영향을 끼칠 수 있는 동적으로 변화 가능한 값들의 집합

따라서 커널 상태의 인지에 필요한 요소는 1) 현재 값을 나타내는 현재 값(current value), 2) 한계 기준이 되는 바람직하지 못한 값(undesirable value), 3) 고유한 식별을 위한 식별자(identifier), 그리고 4) 커널 상태 값의 컨테이너(container)의 위치 정보(커널 내 가상 주소)로 구성된다.

2) 표본 수집

Sampler는 두 가지의 하위 구성속성을 가지는데, 각각은 샘플링 빈도(sampling frequency)와 컨테이너 크기(container size)이다. 샘플링 빈도는 변화하는 커널 상태의 수집 빈도를 의미한다. 이 빈도가 의미 있는 이유는, 커널 상태의 수집 행위로 인한 로드(load)의 조절 요소이고, 수집된 데이터를 통해 바람직하지 못한 상태를 예측할 때의 정확도와 관련이 있기 때문이다. 수집 빈도수를 높여 조

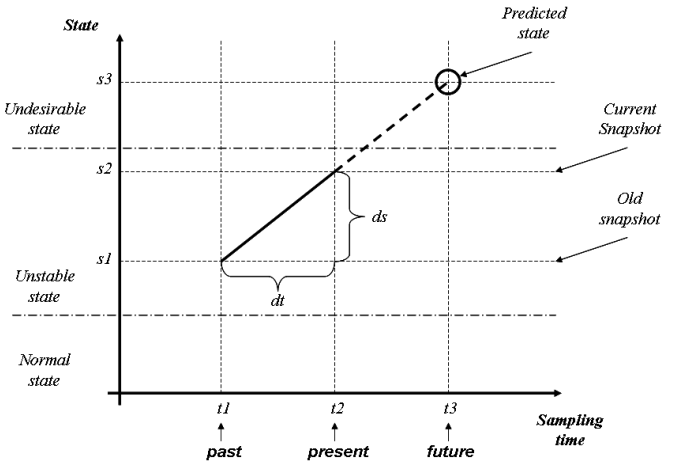

그림 4 선형 상태 변화의 예측 예제

밀한 수준의 수집(fine-grain collection)을 수행한다면, 현재 상태에 근사한 스냅샷을 얻을 수 있으므로 좀 더 정확한 분석에 이용될 수 있을 것이다. 그러나 이는 빈번한 수집으로 인한 오버헤드(overhead)를 증가 시킬 수 있다. 그 반대인 경우, 즉 수집 빈도수를 낮추어 성긴 수준의 수집(coarse-grain collection)을 수행한다면, 수집 오버헤드는 낮출 수 있으나 비교적 과거의 커널 상태의 스냅샷이므로 예방을 위한 예측이 부정확할 수 있다. 컨테이너 크기는 분석에 이용할 스냅샷 개체(snapshot unit)의 수를 나타낸다. 비교적 많은 수의 개체를 보관한다면 다음 상태를 예측할 때 정확도를 높일 수 있다. 그러나 이는 커널 상태 변화 속도와 샘플링 빈도와 관련이 있다. 만약 커널 상태의 변화 속도가 크고 샘플링 빈도도 빈번하다면, 많은 수의 스냅샷을 저장할 필요가 없다. 이렇게 변화 요소가 많기 때문에, 본 연구에서는 sampler의 샘플링 빈도와 컨테이너 크기를 특정 값으로 고정하지 않고 유연하게 변경 될 수 있도록 하였다.

커널 상태 스냅샷(Kernel state snapshot)은 샘플링(sampling)의 결과로 보관되는 커널 상태이며, 커널 상태 식별에 필요한 정보와 샘플링 시간(sampling time)을 포함한다. 이는 이후 analyzer의 판단과 preventer가 조작을 가할 대상 커널 상태의 식별에 사용되게 된다.

3.2. Analyzer

Analyzer는 Sampler가 수집한 커널 상태의 스냅샷을 이용하여 현 시점에서 예방이 필요한가를 판단하고, 필요하다면 예방 처리를 수행하는 preventer를 활성화시킨다. Analyzer가 가지는 하위 속성인 예측 함수(estimation function)는 수집된 커널 상태 스냅샷을 이용하여 예방의 필요성을 판단하는 함수이다. 함수의 내용은 예측(prediction) 방법과 그 정의에 따라서 가변적일 수 있다. 가장 간단한 예측 함수는 상태가 선형인(linear) 상황의 예측[14]이며 그 예는 그림 4와 같다. 그림 4를 통해 알 수 있듯, 샘플링을 하는 시점인 t1, t2, t3가 존재할 때, 현재 시점인 t2에서 미래인 t3의 kernel state를 예측하려고 한

다면, 다음의 간단한 예측 식을 적용할 수 있다.

Predicted state is s3,
*s3 = (ds / dt) * (t3 - t1) + s1*
if (s3 >= undesirable_state) return YES
else return NO

예측 함수는 산출된 s3를 바람직하지 못한 상태의 하위 경계값(lower boundary value)과 비교하여 그 경계 안에 존재한다면 예방(prevention)이 필요함을 뜻하는 YES를 반환한다. 이 예측 함수를 사용하기 위한 sampler의 하위 속성들을 정의한다면, 먼저, 샘플링 빈도는 1/dt가 되며, 컨테이너 크기(container size)는 이전에 저장된 스냅샷(old snapshot)과 현재 저장할 스냅샷(current snapshot)을 보관해야 하므로 2 가 된다. Analyzer는 예측 함수의 결과를 바탕으로, preventer의 활성화 유무를 결정하게 된다.

3.3. Preventer

Preventer는 Analyzer에 의해서 예측된 바람직하지 못한 상태의 진입을 피할 수 있는 예방책들을 적용하는 역할을 담당한다. 따라서 preventer가 가지는 하위 속성은 가용한 예방책들 나타내는 솔루션 리스트(solution list)이다. 이 솔루션 리스트들은 바람직하지 못한 상태를 벗어날 수 있는 처리 함수의 함수 포인터(function pointer)들을 유지하고 있으며, 이 처리 함수의 위치는 해당하는 타겟 서브시스템 내의 루틴(routine)이 된다. 리스트로써 유지하는 이유는 가용한 대응 솔루션(alternative solution)들이 존재할 경우 이를 지원하기 위함이다. 여기에서 가용한 여러 대응 방법 가운데 가장 적합한 솔루션을 찾는 것은 별개의 문제이므로 본 논문에서는 다루지 않을 것이다.

여기서, 예방 솔루션(prevention solution)의 적용 전략은 best-effort를 기본 원칙으로 한다. 즉, 예방 기법의 적용 결과에 대한 검증을 preventer가 수행하지 않는다는 것이다. 그 이유는 현재 시점에서는 커널 상태가 실제로 바람직하지 못한 상태로 진입한 것이 아니고 예측된 것이기 때문에, 부가적인 검증으로 인한 오버헤드를 감내하는 것은 적절치 못하며, sampler에 의해서 새로이 커널 상태가 갱신되므로, 그것을 이용하는 것이 예측 정확도를 높일 수 있을 것이라는 기대 때문이다. 따라서 preventer는 선택된 예방 솔루션을 적용하고 실패와 성공의 유무에 상관없이 더 이상의 부가적인 연산을 수행하지 않는다.

3.4. 코어 시스템 내 서브 컴포넌트들 간의 상호 작용

Self-prevention의 오버헤드의 분석을 위해서는 코어 시스템을 구성하는 세 가지 구성요소들이 서로 어떠한 영향을 끼치고 타겟 서브시스템이 관리하는 커널 상태에 언제 영향을 미치는지 살펴보아야 한다. 따라서 각 구성요소들의 상호관계와 커널 상태의 조작 시점을 살펴보면 다음의 그림 5와 같다. 이는 코어 시스템의 주요 구성요소인 sampler, analyzer, preventer와 관리 대상인 타겟 서브시스템들의 활성화 지점(activation point)과 상호작용을 도식화한 것이다.

코어 시스템의 세 가지의 구성요소들이 활성화 되는 시점은 각기 Sampler가 활성화 되는 capturing point, analyzer가 활성화되는 analyzing point, 그리고

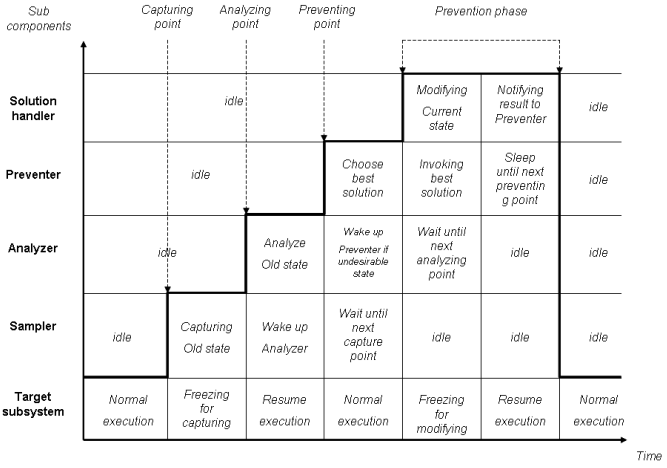

그림 5 코어 시스템 내 서브 컴포넌트들 간 상호 작용과 그 연산

preventer가 활성화되는 preventing point가 존재한다. 이 중 capturing point는 샘플링 빈도에 의해서 결정되며, 그 활성화 인터벌(activation interval)은 균등하다. 그러나 analyzing point와 prevention point는 활성화 조건에 따라서 활성화되는데 그 조건은 다음과 같다.

- *Analyzing point: 포획된 상태(captured state)가 불안정 상태에 속할 때.*
- *Prevention point: analyzer에 의한 예측 결과에서 예방이 요구 될 때*

코어 시스템에 의해 한 번의 예방이 적용될 때 타겟 서브시스템이 관리하는 커널 상태는 두 번의 프리징(freezing, 상태 변화가 금지된 상황) 과정을 거쳐야 한다. 이 시점에서 상태의 일관성(consistency) 보장을 위한 지연 시간이 발생할 수 있으며 이는 self-prevention으로 인한 오버헤드로 간주할 수 있다.

4. 관련연구

본 연구는 두 개의 광범위한 연구 분야에 걸쳐 있다. 이는 각각 자율형 컴퓨팅과 확장형 운영체제 연구이다. 자율형 컴퓨팅은 현재 많은 연구들[1][2][5]이 활발히 진행중이다. 본 연구는 그러한 연구들에 포함 될 수 있으나 다음과 같은 점에서 차이가 있다. 본 연구의 관심사는 순수 운영체제 커널로 전문화 및 한정화 한다는 것이다. 또한 본 연구에서는 커널 수준에서 지원 가능한 자율적인 실패처리(failure handling)를 다루고 있으며, 이는 커널 구조적인 접근을 하고 있다. 이와 유사한 연구로는 두 가지가 있는데, aspect-oriented 연구와 model-based 연구가 있다. [16]에서는 aspect oriented 언어를 사용하여 언어 기반의 접근방법을 통해 커널에 자율성을 부여하려 했고 [17]에서는 decision-theoric dispatch를 이용하여 reactive model 기반의 접근방법을 이용하여 시스템 자율성 부여를 시도

하였다.
많은 확장형 운영체제 연구들에서는 전통적으로 기능 확장과 확장 단위의 안정성에 초점을 두었다.[3] 본 연구는 명시적으로 확장형 운영체제 연구에 속하지는 않지만 self-prevention 개념이 확장형 시스템의 기능 확장 단위인 커널 내 서브시스템의 독립성을 향상 시키는데 도움을 줄 수 있을 것이다.

5. 결론

본 연구의 기여도를 정리하면 다음과 같다. 먼저, 본 연구는 self-managing 혹은 자율형 컴퓨팅 분야의 새로운 시도라는 점이다. 기존의 자율형 컴퓨팅의 적용 분야들에서는 주로 인간 전문가에 의해 관리 가능한 데이터베이스 시스템(database system)[5]이나 서버 구성(configuration)의 관리[7]에 초점을 두었지만, 본 연구에서는 일반 커널(generic kernel)내의 computing 자원의 관리에 관련된 핵심 관리부와 그 관리 대상으로써의 커널 서브시스템에 초점을 두었다. 두 번째로, 기존 확장형 운영체제(extensible operating system) 연구와 autonomous computing 연구와의 연결 고리가 될 수 있다는 점이다. 본 연구에서 다루는 대상 중 하나인 커널 서브시스템들은 확장형 운영체제 연구의 맥락에서는 기능 확장의 단위가 될 수 있다. 본 연구에서는 커널 서브시스템의 자율성을 부여하고 최대한 독립성을 유지하려 했기 때문에 이는 운영체제의 기능 확장성에 도움이 될 수 있을 것으로 본다.

참고문헌

[1] Richard Murch. Autonomic Computing. IBM Press, 2004.
[2] Michael G. Hinchey, Roy Sterritt. Self-Managing Software. IEEE Computer Vol. 39, No. 2, February 2006. pp. 107-109.
[3] G. Denys, F. Piessens, F.Matthijs. A Survey of Customizability in Operating Systems Research. ACM Computing Surveys, Vol. 34, No. 4, December 2002. pp. 450-468.
[4] P. Druschel, G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In Proceedings of USENIX Operating Systems Design and Implementation, 1996.
[5] Guy M. Lohman, Sam S. Lightstone. SMART: Making DB2 (More) Autonomic. In Proceedings of International Conference on Very Large Data Bases, 2002.
[6] Krishna Kant, Prasant Mohapatra. Scalable internet servers: issues and challenges. ACM SIGMETRICS Performance Evaluation Review, Vol. 28, Issue 2, 2000. pp. 5-8.
[7] Alfred Z. Spector. Challenges and Opportunities in Autonomic Computing. Keynote address in 16th Annual ACM International Conference on Supercomputing 2002.
[8] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, Henry M. Levy. Recovering Device Drivers. In Proceedings of USENIX Operating Systems Design and Implementation, 2004.
[9] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronization. IEEE Transaction on Computers, Vol. 39, No. 9, September 1990. pp. 1175-1185.
[10] Toshimi Minoura. Deadlock Avoidance Revisited. Journal of the ACM, Vol 29, Issue 4, October 1982. pp. 1023-1048.
[11] Walter H. Kohler. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. ACM Computing Surveys, Vol. 13, Issue 2, June 1981. pp. 139-183.
[12] Jacob Rubin Lorch. Operating Systems Techniques for Reducing Processor Energy Consumption. In PhD Thesis of University of California, Berkeley, Fall 2001.
[13] David Wagner, Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In Proceedings of the ACM Conference on Computer and Communications Security, 2002. pp. 255-264.
[14] William H. Press, Brian P. Flannery, Saul A. Teukosky, William T. Vetterling. Numerical Recipes in C : The Art of Scientific Computing. Cambridge University Press, October 1992. pp. 32-36.
[15] Galen C. Hunt, James R. Larus, David Tarditi, Ted Wobber. Broad New OS Research: Challenges and Opportunities. In Proceedings of Workshop on Hot Topics in Operating Systems, 2004.
[16] Michael Engel, Bernd Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating Systems Kernel Aspects. In Proceedings of Aspect Oriented Software Development 2004.
[17] Paul Robertson, Brian Williams. Automatic Recovery from Software Failure. Communications of the ACM, Vol. 49, No. 3, March 2006.