(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2013-0031860
(43) 공개일자 2013년03월29일

(51) 국제특허분류(Int. Cl.)
G06F 11/36 (2006.01) *G06F 11/30* (2006.01)
(21) 출원번호 10-2012-7034454
(22) 출원일자(국제) 2010년09월07일
심사청구일자 2012년12월31일
(85) 번역문제출일자 2012년12월31일
(86) 국제출원번호 PCT/KR2010/006068
(87) 국제공개번호 WO 2012/033237
국제공개일자 2012년03월15일

(71) 출원인
현대자동차주식회사
서울특별시 서초구 헌릉로 12 (양재동)
이화여자대학교 산학협력단
서울특별시 서대문구 이화여대길 52 (대현동, 이화여자대학교)
기아자동차주식회사
서울특별시 서초구 헌릉로 12 (양재동)
(72) 발명자
최병주
서울특별시 강남구 압구정동 현대아파트 74-1305
서주영
서울특별시 송파구 문정동 올림픽훼밀리타운 105-304
(뒷면에 계속)
(74) 대리인
특허법인태평양

전체 청구항 수 : 총 12 항

(54) 발명의 명칭 **시스템 테스트 방법**

(57) 요 약

본 발명은 프로세스 제어 블록(Process Control Block)의 위치를 식별하는 단계, 프로세스 제어 블록의 위치에 접근하는 단계, 및 프로세스 제어 블록의 성능 요소를 모니터링하는 단계를 포함하는 시스템 테스트 방법을 제공한다.

(72) 발명자
양승완
경기도 군포시 광정동 삼성장미아파트 1137-2202

오정석
경기도 광명시 광명1동 70-13, 2층

특허청구의 범위

청구항 1

프로세스 제어 블록(Process Control Block)의 위치를 식별하는 단계;

상기 프로세스 제어 블록의 위치에 접근하는 단계; 및

상기 프로세스 제어 블록의 성능 요소를 모니터링하는 단계를 포함하는 시스템 테스트 방법.

청구항 2

청구항 1에 있어서,

상기 프로세스 제어 블록은

커널 메모리 내부에 위치하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 3

청구항 2에 있어서,

상기 프로세스 제어 블록의 위치에 접근하는 단계에 있어서,

가상 드라이버를 사용하여 상기 커널 메모리에 접근하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 4

청구항 1에 있어서,

상기 프로세스 제어 블록의 성능 요소를 모니터링하는 단계에 있어서,

미리 설정된 시간 간격마다 상기 프로세스 제어 블록의 성능 요소를 측정하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 5

청구항 2에 있어서,

상기 프로세스 제어 블록의 성능 요소를 모니터링하는 단계에 있어서,

상기 프로세스 제어 블록의 프로세스 및 쓰레드(Thread)에 대한 성능 요소를 측정하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 6

청구항 5에 있어서,

상기 프로세스 제어 블록의 성능 요소는

프로세서 사용량, 메모리 사용량, 페이지 폴트(Page Fault) 중 하나 이상을 포함하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 7

청구항 6에 있어서,

상기 프로세스 제어 블록의 프로세스에 대한 성능 요소는

아이디(ID), 프로세스의 상태, 프로세스의 우선 순위, 힙(Heap) 사용량, 프로세스의 동작 시간, 사용 시간, 커널 사용 시간 중 하나 이상을 포함하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 8

청구항 6에 있어서,

상기 프로세스 제어 블록의 쓰레드에 대한 성능 요소는

아이디, 운영 상태(Run State), 기본 우선 순위, 현재 우선 순위, 사용 시간, 커널 사용 시간, 콜 스택(Call Stack) 중 하나 이상을 포함하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 9

청구항 1에 있어서,

상기 프로세스 제어 블록은

프로세스의 식별자, 레지스터 컨텍스트(Context), 프로세스의 어드레스, 메모리 사용량, 공유 함수 목록, 프로세스의 자원(Resource), 프로세스의 우선 순위, 프로세스의 상태 중 하나 이상에 대한 정보를 저장하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 10

청구항 1에 있어서,

상기 프로세스 제어 블록의 성능 요소를 모니터링하는 단계는

커널에 분석 코드를 삽입하여 상기 프로세스 제어 블록의 성능 요소를 모니터링하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 11

청구항 1에 있어서,

상기 프로세스 제어 블록의 성능 요소를 모니터링하는 단계는

시뮬레이션 프로그램을 사용하여 상기 프로세스 제어 블록의 성능 요소를 모니터링하는 것을 특징으로 하는 시스템 테스트 방법.

청구항 12

청구항 1에 있어서,

상기 프로세스 제어 블록의 성능 요소를 모니터링하는 단계는

레지스터를 포함하는 하드웨어를 사용하여 상기 프로세스 제어 블록의 성능 요소를 모니터링하는 것을 특징으로 하는 시스템 테스트 방법.

명 세 서

기 술 분 야

[0001] 본 발명은 시스템 테스트 방법에 관한 것이다.

배 경 기 술

[0002] 최근 자동차와 같은 기계 산업 분야에서도 전자제어 기술에 대한 관심이 크다. 전자제어 기술의 핵심은 임베디드 소프트웨어이며, 이들 소프트웨어의 품질이 전체 제품의 품질에 큰 영향을 미친다.

[0003] 종래에도 품질에 도움을 주는 소프트웨어 테스트 기술과 테스트 도구들이 개발되었다. 하지만 임베디드 소프트웨어의 고유한 특징들 때문에, 이들을 그대로 임베디드 소프트웨어 테스트에 적용하기에는 여러가지 문제가 있다.

[0004] 첫째, 임베디드 소프트웨어는 특정 하드웨어와 기능에 맞춤되어 제작되는 소프트웨어이다. 대부분의 임베디드 소프트웨어는 메모리와 같은 시스템 자원의 여유가 없는 제약적 조건에 최적화되어 동작하도록 설계한다. 따라서 타겟 환경에서 동작하는 임베디드 소프트웨어는 호스트 환경에서 동작하는 일반 소프트웨어와 비교했을 때 시스템 운용 시 자원 제약이 심하다는 문제점이 있다. 둘째, 임베디드 소프트웨어는 메뉴 선택과 같은 사용자 명령에 의한 동작보다 전자적 시그널이나 통신 프로토콜과 같은 다양한 유형의 외부 입력에 의한 동작이 더 많

은 비중을 차지한다. 따라서 사용자 명령에 의해 주로 동작하는 종래의 소프트웨어 테스트 기술과 테스트 도구들을 임베디드 소프트웨어 테스트에 그대로 적용하기 어렵다는 문제점이 있다.

[0005] 셋째, 종래의 소프트웨어 테스트 기술은 성능 테스트 시에 높은 프로세서 사용율이 지속되는 경우에만 성능 병목을 나타내는 증후로 간주한다. 이 경우 중요한 컴포넌트의 프로세서 사용율이 기본 성능 이하로 떨어지는 경우에는 이상 징후로 판단하지 않는다. 따라서 시스템의 성능을 정확하게 테스트할 수 없다는 문제점이 있다.

발명의 내용

해결하려는 과제

[0006] 본 발명은 시스템 운용 시에 필요한 자원을 최소한으로 사용하여 시스템의 운용 환경에 영향을 주지 않으면서 성능 병목과 그 원인 및 위치를 식별하기 위한 데이터를 수집하는 시스템 테스트 방법에 관한 기술과 관련된다.

과제의 해결 수단

[0007] 본 발명은 프로세스 제어 블록(Process Control Block)의 위치를 식별하는 단계, 프로세스 제어 블록의 위치에 접근하는 단계, 및 프로세스 제어 블록의 성능 요소를 모니터링하는 단계를 포함하는 시스템 테스트 방법을 제공한다.

발명의 효과

[0008] 본 발명은 시스템 운용에 필요한 자원을 최소한으로 사용하여 시스템의 운용 환경에 영향을 주지 않으면서 시스템을 테스트할 수 있다는 장점이 있다.

[0009] 추가적으로, 본 발명은 시스템의 외부로부터 입력되는 데이터를 기반으로 시스템을 테스트할 수 있다는 장점이 있다.

[0010] 추가적으로, 본 발명은 프로세서의 사용율이 기준 이하로 떨어지는 경우에 대해서도 병목 현상인지 여부를 테스트할 수 있다는 장점이 있다.

도면의 간단한 설명

[0011] 도 1은 본 발명에 따른 시스템 테스트 방법을 나타내는 순서도.

도 2는 본 발명에 따른 시스템 테스트 방법을 구현하는 프로그램 코드.

도 3은 본 발명에 따른 시스템 테스트 방법을 실행하는 시스템의 구조를 나타내는 도면.

발명을 실시하기 위한 구체적인 내용

[0012] **발명의 실시를 위한 최선의 형태**

[0013] 본 발명의 시스템 테스트 방법은 프로세스 제어 블록(Process Control Block)의 위치를 식별하는 단계, 프로세스 제어 블록의 위치에 접근하는 단계, 및 프로세스 제어 블록의 성능 요소를 모니터링하는 단계를 포함한다.

[0014] **발명의 실시를 위한 형태**

[0015] 아래에서는 도면을 참고하여, 본 발명의 다양한 실시예를 구체적으로 살펴본다.

[0016] 본 발명에 따른 시스템 테스트 방법에서, 프로세스 제어 블록(Process Control Block)은 운용 체제가 시스템에 운용되고 있는 프로세스들의 실행 정보를 관리하는 자료 구조를 의미한다.

[0017] 예를 들어, 프로세스 제어 블록은 시스템에 운용 중인 프로세스들의 최신 실행 정보를 실시간으로 관리하는 커널의 자료 구조를 의미할 수 있다.

[0018] 본 발명에 따른 시스템 테스트 방법은 시스템의 메모리에 발생한 결함을 발견하기 위해, 프로세스 제어 블록을 통해 메모리와 관련된 함수 테이블을 후킹한다.

[0019] 특히 본 발명에 따른 시스템 테스트 방법은 프로세스 제어 블록의 데이터 중 성능 병목과 그 원인을 분석하기 위해 페이지 폴트율과 프로세서 사용율과 같은 시스템 실행 정보들을 해킹하여 시스템 성능을 분석할 수 있다.

[0020] 프로세스 제어 블록 해킹은 성능 분석에 필요한 데이터 수집을 프로세스 제어 블록 한 곳으로 집중시킴으로써 테스트로 인한 시스템 성능 저하를 최소화할 수 있다.

[0021] 또한 본 발명은 위와 같은 시스템 성능 저하를 최소화하면서, 동시에 시스템 운용 환경에서의 성능 테스트 요구사항을 만족시킬 수 있다.

[0022] 예를 들어, 본 발명에 따른 시스템 테스트 방법은 다음과 같은 조건을 가정하고 수행될 수 있다.

[0023] [시스템 테스트 조건]

[0024] 1. 테스트 범위: 전체 시스템내의 모든 하드웨어와 소프트웨어 컴포넌트들이 동작하는 상황의 시스템에 대한 성능 테스트.

[0025] 2. 테스트 방식: 시스템 실행 방식을 보장하기 위해 re-compile, re-link, re-execute 되지 않는 방식으로 런-타임 테스트.

[0026] 3. 테스트 대상: 디버깅 정보와 같은 부가적인 코드를 포함하지 않는 테스트 대상 소프트웨어의 원본 소스/이진 코드 변경 없이 시스템에 로딩된 바이너리를 대상으로 한 테스트.

[0027] 4. 성능 데이터: 성능 병목의 원인으로 프로세서뿐만 아니라 메모리, I/O, 네트워크 자원까지 고려할 수 있는 성능 데이터 수집.

[0028] 5. 병목 위치 추적: 성능 병목의 위치로 함수와 같이 소스 레벨 분석이 가능한 위치 데이터 수집.

[0029] 6. 하드웨어 독립성: 테스트를 위한 별도의 하드웨어 추가나 의존성 없이 시스템 운용 환경과 동일한 하드웨어 조건에서의 테스트.

[0030] 7. 소프트웨어 독립성: 테스트를 위해 특별히 제작된 커널 (Instrumented Kernel)이나 가상머신을 사용하지 않는 시스템 운용 환경과 동일한 소프트웨어 조건에서의 테스트.

[0031] 8. 성능 지연율: 실시간 동작을 보장하기 위해 테스트로 인한 시스템 성능 지연 율 최소화.

[0032] 9. 코드 추가율: 제한된 자원 내에서 동작하기 위해 테스트로 인한 코드 추가 율 최소화.

[0033] 하지만 이러한 조건은 시스템 운용 환경에 따라 얼마든지 변화할 수 있으며, 본 발명은 이러한 시스템 운용 환경에 제한되지 않고 적용될 수 있다.

[0034] 프로세스 제어 블록은 시스템에 운용되는 프로세스의 실행 정보를 담고 있는 커널 자료구조이다. 프로세스 제어 블록에는 성능 병목을 판단할 프로세서 사용량과 여유 메모리 사이즈와 같은 성능 요소들도 포함되어 있으며, 커널에 의해 이들 값은 항상 가장 최근 값으로 유지된다.

[0035] 본 발명에 따른 시스템 테스트 방법은 이 값들을 해킹하는 에이전트 개발을 위한 성능 분석에 사용될 수 있다. 아래에서는 성능 병목의 원인을 분석하기 위해 필요한 성능 요소 및 연관된 프로세스 제어 블록 구조를 살펴본다.

[0036] 성능은 시스템이나 컴포넌트들이 주어진 시스템의 제약 조건 내에서 기능을 수행하는 정도를 의미한다. 성능 테스트는 특정 성능 요구사항을 시스템이 만족하는 지에 관한 평가이다.

[0037] 이러한 성능 테스트를 통해 성능 병목과 그 원인을 분석하여 이를 해결함으로 시스템의 성능을 개선할 수 있다.

[0038] 시스템의 성능 병목은 메모리, I/O 디바이스, 네트워크와 같은 제한된 자원들에 대한 경쟁으로 시스템의 성능이 저하되는 현상이다. 시스템 성능 병목의 원인은 자원 부족, 공유자원 경합, 자원 독점, 자원의 잘못된 구성 혹은 자원의 잘못된 동작과 같이 다양하다.

[0039] 먼저 메모리가 원인이 되는 성능 병목은 사용 가능한 메모리가 부족할 때에 발생할 수 있으며, 메모리 부족 증후를 식별할 수 있는 핵심적인 메모리 성능 요소로 페이지폴트와 메모리 사용량이 있다.

[0040] 본 발명에 따른 시스템 테스트 방법은 다양한 성능 병목들을 식별할 수 있다.

[0041] 먼저 본 발명에 따른 시스템 테스트 방법은 페이지폴트를 기준으로 메모리 병목을 판단할 수 있다.

[0042] 예를 들어, 페이지 폴트가 높으면 메모리 병목일 수 있다. 페이지폴트란 프로그램이 자신의 주소 공간에는 존재하지만 시스템의 메모리에는 현재 없는 데이터나 코드에 접근 시도하였을 경우 발생하는 현상이다. 페이지폴트

가 발생하면 운영체제는 그 데이터를 메모리로 가져와서 마치 페이지폴트가 전혀 발생하지 않은 것처럼 프로그램이 계속적으로 작동하도록 한다. 운영체제의 페이지폴트에 대한 예외처리 때문에 어플리케이션의 처리시간은 지연되며, 전체 시스템 성능에 영향을 준다.

[0043] 또한 본 발명에 따른 시스템 테스트 방법은 메모리 사용량을 통해 성능 병목을 식별할 수 있다. 시스템의 메모리는 물리 메모리 사용량, 가상 메모리 사용량이 있고, 프로세스 별로 힙 메모리 사용량으로 구분할 수 있다. 본 발명에 따른 시스템 테스트 방법은 이러한 메모리 사용량들의 합을 기준으로 성능 병목을 판단한다.

[0044] 또한 본 발명에 따른 시스템 테스트 방법은 프로세서 사용량 (혹은 CPU 사용량)을 통해 성능 병목을 식별할 수 있다.

[0045] 예를 들어, 본 발명에 따른 시스템 테스트 방법은 프로세서 사용량이 높게 유지되면서 여유 메모리가 있는 경우는 CPU에 병목이 있다고 판단할 수 있다. 반면 프로세서 사용량이 높고 메모리도 고갈되었다면 성능 문제는 CPU보다 메모리 병목으로 판단할 수 있다.

[0046] 또한 본 발명에 따른 시스템 테스트 방법은 프로세스 사용량 (Processor Usage)에 따라 성능 병목을 식별할 수 있다. 프로세스 사용량은 시스템의 실행 시간으로 전체 CPU 사용량에서 대기 시간(Idle Time)을 제외한 시간을 의미한다.

[0047] 또한 본 발명에 따른 시스템 테스트 방법은 사용 시간 (User Time)에 따라 성능 병목을 판단할 수 있다. 사용 시간은 실행이 사용자 공간에서 머무른 시간을 의미하며, 곧 어플리케이션의 실행 시간을 의미한다.

[0048] 또한 본 발명에 따른 시스템 테스트 방법은 커널 사용 시간 (Kernel Time)에 따라 성능 병목을 판단할 수 있다. 커널 사용 시간은 실행이 커널 공간에서 머무른 시간으로, 커널의 서비스 처리 시간을 의미한다.

[0049] 프로세스 제어 블록은 런-타임에 프로세스들을 제어하기 위한 목적으로 운영체제 커널이 관리하는 자료구조이다. 일반적으로 프로세스 제어 블록에는 프로세스 식별자, 레지스터 컨텍스트, 프로세스의 주소 공간, 프로세스의 메모리 사용량, 공유 함수 목록, 프로세스가 소유하는 자원들, 프로세스의 우선순위와 상태 같은 실행 정보들이 저장되어있다.

[0050] 도 1은 본 발명에 따른 시스템 테스트 방법을 나타내는 순서도이다.

[0051] 도 2는 본 발명에 따른 시스템 테스트 방법을 구현하는 프로그램 코드이다.

[0052] 도 1을 참고하면, 본 발명에 따른 시스템 테스트 방법은 프로세스 제어 블록의 위치를 식별하는 단계(S100), 프로세스 제어 블록에 접근하는 단계(S200) 및 프로세스 제어 블록의 성능 요소를 모니터하는 단계(S300)를 포함한다.

[0053] 프로세스 제어 블록의 위치를 식별하는 단계(S100)에서, 프로세스가 생성되고 사라질 때 그 프로세스 제어 블록도 함께 생성되고 사라지기 때문에 프로세스 제어 블록이 저장되는 위치는 결정되어 있지 않다.

[0054] 그러나 프로세서(예를 들어, CPU)를 점유하는 현행 프로세스 (Current Process)에 관한 프로세스 제어 블록의 base address는 특정 메모리 공간에 관리되거나, 경우에 따라 미리 고정된 주소로 관리되기 때문에 현행 프로세스의 프로세스 제어 블록 정보를 알 수 있다.

[0055] 예를 들어, 도 2의 프로그램 코드의 line 7 에서 처럼 현행 프로세스의 프로세스 제어 블록 base address를 스택 포인터로 계산하고, line 31처럼 모든 프로세스의 프로세스 제어 블록으로 접근할 수 있다.

[0056] 프로세스 제어 블록의 위치에 접근하는 단계(S200)에서, 프로세스 제어 블록은 커널 메모리 공간에 존재한다.

[0057] 이 경우 커널 공간에는 보안(Security) 차원에서 접근을 허용하지 않을 수 있다. 이처럼 커널 공간에 대한 접근이 차단될 때, 아래 프로그램 코드처럼 pseudo codes를 커널 주소 공간을 공유할 수 있도록 가상 드라이버 형태로 구현하여 커널 메모리 공간에 존재하는 프로세스 제어 블록에 접근할 수 있다.

[0058] 프로그램 제어 블록의 성능 요소를 모니터하는 단계(S300)에서, 도 2의 프로그램 코드의 line 61와 line 65처럼 timer-interrupt를 사용하며 지정된 시간 간격 (e.g. 1sec, 100msec) 마다 성능 데이터를 측정할 수 있다.

[0059] 또한 본 발명에 따른 시스템 방법에서는 시스템 성능을 시스템 단위뿐만 아니라 프로세스와 쓰레드의 단위로까지 분석하기 위해 도 2의 프로그램 코드의 line 31과 line 33처럼 프로세스 제어 블록의 프로세스와 쓰레드 리스트들을 순환하면서 성능 요소를 측정할 수 있다.

[0060] 이 경우 시스템 성능 병목이 발생하는 위치를 자세히 추적하기 위해 쓰레드 별로 콜스택(Callstack) 정보를 저장한다. 각 모니터링 단위 별 측정하는 성능 요소는 다음과 같다.

[0061] 프로세스 제어 블록의 성능 요소는 프로세서 사용량, 메모리 사용량, 페이지 폴트(Page Fault) 중 하나 이상을 포함할 수 있다.

[0062] 프로세스 제어 블록의 프로세스에 대한 성능 요소는 아이디(ID), 프로세스의 상태, 프로세스의 우선 순위, 힙(Heap) 사용량, 프로세스의 동작 시간, 사용 시간, 커널 사용 시간 중 하나 이상을 포함할 수 있다.

[0063] 프로세스 제어 블록의 쓰레드에 대한 성능 요소는 아이디, 운영 상태(Run State), 기본 우선 순위, 현재 우선 순위, 사용 시간, 커널 사용 시간, 콜스택 정보 중 하나 이상을 포함할 수 있다.

[0064] 도 3은 본 발명에 따른 시스템 테스트 방법을 실행하는 시스템의 구조를 나타낸다.

[0065] 도 3을 참고하면, 본 발명에 따른 시스템 테스트 방법을 실행하는 시스템은 에이전트부(120) 및 테스트 관리부(210)를 포함한다.

[0066] 에이전트부(120)는 테스트의 대상이 되는 타켓 시스템에 포함되어, 성능 데이터를 측정하는 역할을 수행한다. 에이전트부(120)는 도 1에 도시된 알고리즘을 실행한다.

[0067] 테스트 관리부(210)는 PC (Personal Computer)와 같은 호스트 시스템(200)에 포함되어, 에이전트부(120) 로부터 수집된 성능 데이터를 분석하여 성능 병목을 검출하고 테스트 커버리지를 분석한다.

[0068] 구체적으로, 에이전트부(120)는 운용 환경에 배치된 타겟 시스템에 함께 탑재되어 사용자의 테스트 시작과 종료 명령에 따라 시스템이 운영되면서 주기적으로 시스템 성능과 관련된 데이터를 측정하는 역할을 수행한다.

[0069] 예를 들어, 에이전트부(120)는 PerfAgent.dll 및 PerfROBO.exe를 포함할 수 있다.

[0070] PerfAgent.dll은 도 1에 도시된 알고리즘을 구현하는 가상의 커널 디바이스 드라이버로 성능 테스트를 위해 커널의 프로세스 제어 블록 정보를 해킹한다.

[0071] PerfROBO.exe는 사용자의 테스트 시작과 종료 명령에 의해 PerfAgent.dll의 활성화 여부를 제어하는 테스트 서버 역할을 수행한다.

[0072] 테스트 시작 명령에 의해 PerfAgent.dll이 활성화되면, 타이머 설정을 통해 시스템 성능 모니터링 모듈을 실행시키고 사용자 요구에 의해 테스트를 종료할 때에도 설정된 타이머를 종료시킴으로써 테스트를 종료한다.

[0073] 한편 테스트 관리부(210)는 수집된 성능 데이터를 지정된 저장 매체에 저장할 수 있다. 테스트 관리부(210)는 호스트 시스템(200) 로그 파일을 분석하여 운영 타임에 발생한 성능 병목을 검출하는 역할을 수행한다.

[0074] 예를 들어, 테스트 관리부(210)가 성능 데이터를 저장 매체에 이진 코드로 저장한다면, 테스트 관리부(210)는 테스트 대상에 대한 이진 실행 코드와 수집된 프로파일링 데이터를 입력으로 하며, 테스트 커버리지와 성능 정보를 출력할 때 콜-스택정보를 기반으로 어느 함수 위치에서 문제가 발생하였는지의 위치 정보를 함께 디스플레이할 수 있다.

[0075] 본 발명에 따른 시스템 테스트 방법에서, 시스템의 성능을 모니터링하는 다양한 방법이 사용 가능하다.

[0076] 먼저 커널에 분석코드를 삽입하는 방법이 사용될 수 있다. 이 기술은 커널에 분석코드를 정적으로 혹은 동적으로 삽입시킨다.

[0077] 동적으로 삽입하는 기술로는 런-타임에 커널 코드에 분석 코드를 삽입할 수 있으며, 이 경우 시스템을 재부팅하지 않으면서 성능 분석에 필요한 데이터를 수집하는 코드를 삽입할 수 있다.

[0078] 정적으로 삽입하는 기술은 미리 제작된 커널을 사용한다.

[0079] 예를 들어, 커널 내부에 중요한 시스템 성능 요소를 모니터링하는 분석코드를 심어둘 수 있다. 성능 모니터를 할 경우, 커널 내부에 미리 삽입되어 있는 분석코드를 통해 성능 데이터를 전송받는다. 그 결과 성능 모니터링을 할 때, 프로세서 사용량, 메모리 사용량을 포함하여 굉장히 다양한 시스템 성능 요소의 측정이 가능하다.

[0080] 추가적으로, 성능 모니터링 기술로 시뮬레이터를 이용하는 방법이 사용될 수 있다. 이 방식은 개발 단계 초기의 시스템의 성능을 살펴보는 경우에 유용하다.

[0081] 추가적으로, 성능 모니터링 기술로 하드웨어를 사용할 수도 있다. 하드웨어 성능 요소들은 특별히 제작된 레지

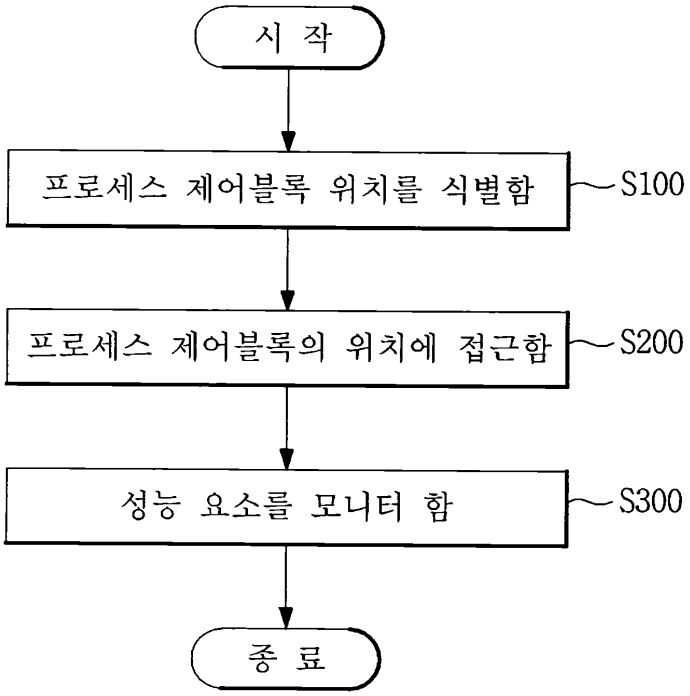
스터 집합을 의미하며, 소프트웨어 기반 성능 요소에 비해 적은 시스템 오버헤드로 CPU, 캐시, 메모리와 연관된 성능을 모니터링하는 것이 가능하다. 이 방식은 소스코드나 이진코드를 수정하지 않아도 된다.

[0082] **산업상 이용가능성**

[0083] 본 발명은 시스템 운용에 필요한 자원을 최소한으로 사용하여 시스템의 운용 환경에 영향을 주지 않으면서 시스템을 테스트할 수 있다.

도면

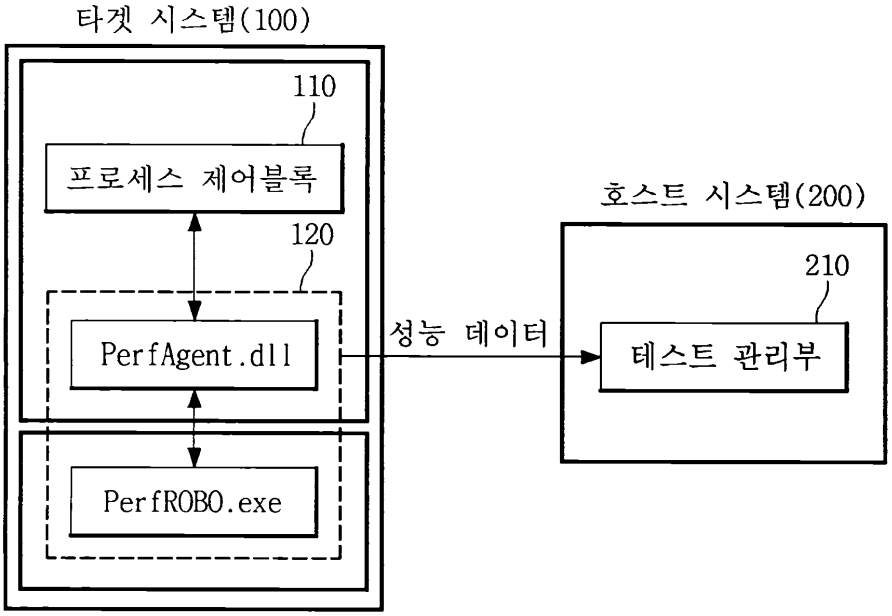
도면1



도면2

```
01  void * pPCB;                                   // PCB information
02  int uTimerId;                                  // Timer ID for performnace monitoring
03  int uTimeOut = 1000;                           // Time-out value in milliseconds
04  DWORD GetKernelStackPoint();  // API for getting value of kernel stack point
05  int SetTimer(uElapse, lpTimerFunc);            // API for creating a timer with the specified time-out value
06  void KillTimer(uTimerId);                      // API for destroying the specified timer
07  void SaveLog(data, ...);              // API for saving data into log-file

08  void PERFAGENT_GetPCB(void)
09  {// Accessing the PCB memory space
10          pPCB = GetKernelStackPoint() & 0xFFFFE000; // e.g., in case of WinCE platform
11  }

12  void PERFAGENT_MemoryUsage()
13  { // Measuring memory usage
14          DWORD total = pPCB->NumPages * pPCB->SysPageSize;
15          DWORD unused = pPCB->FreePhyPage * pPCB->SysPageSize;
16          DWORD usage = total – unused;
17          Save(usage);
18  }

19  void PERFAGENT_HeapUsage(pProc)
20  { // Measuring heap usage
21          DWORD usage ← calculate heap usage using pProc->memPointer;
22          Save(usage);
23  }

24  void PERFAGENT_CallStack(pThread)
25  { // Measuring thread's call stack
26          void * site ← dump callstack using pThread->pcstkTop;
27          Save(site);
28  }

29  void PERFAGENT_PCBHooking()
30  { // PCB hacking
31          PPROCESS pProc = pPCB->procList.pFwd;
32          while (pProc) {
33                  PTHREAD pThread = pProc->thrdList.pFwd;
34                  while (pThread) {
35                          // save thread status
36                          Save(pThread->dwId);
37                          Save(pThread->bBPrio, pThread->bCPrio);
38                          Save(pThread->bRunState, pThread->bWaitState);
39                          Save(pThread->dwUTime);
40                          Save(pThread->dwKTime);
41                          PERFAGENT_CallStack(pThread);
42                          pThread = pThread->pFwd;        // next thread
43                  }
44                  // save process status
45                  Save(pProc->dwId);
46                  Save(pProc->bPrio);
47                  Save(pProc->bState);
48                  Save(pProc->dwProcTime);
49                  PERFAGENT_HeapUsage(pProc);
50                  pProc = pProc->pFwd;            // next process
51          }
52          // save system status
53          Save(pPCB->dwProcTime);
54          Save(pPCB->nPageFault);
55          PERFAGENT_MemoryUsage();
56  }

57  // Performance Agent
58  void PERFAGENT_Start(void)
59  {
60          PERFAGENT_GetPCB();
61          uTimerId = SetTimer(uTimeOut, PERFAGENT_PCBHooking);
62  }

63  void PERFAGENT_Stop(void)
64  {
65          KillTimer(uTimerId);
66  }
```

도면3


타겟 시스템(100)
110
프로세스 제어블록
120
PerfAgent.dll
PerfROBO.exe
성능 데이터
호스트 시스템(200)
210
테스트 관리부