(19)대한민국특허청(KR)
(12) 공개특허공보(A)

(51) 。Int. Cl.
G06F 9/46 (2006.01)
G06F 13/24 (2006.01)

(11) 공개번호 10-2007-0083460
(43) 공개일자 2007년08월24일

(21) 출원번호 10-2007-7001072
(22) 출원일자 2007년01월16일
심사청구일자 없음
번역문 제출일자 2007년01월16일
(86) 국제출원번호 PCT/US2005/023525
국제출원일자 2005년07월01일
(87) 국제공개번호 WO 2006/014354
국제공개일자 2006년02월09일

(30) 우선권주장 11/169,542 2005년06월29일 미국(US)
60/586,486 2004년07월06일 미국(US)

(71) 출원인 엠베디오 인코포레이티드
미국 캘리포니아주 92821 브레아 스위트 307 쓰리 포인트 드라이브

(72) 발명자 데사이 라지브 에스.
미국 캘리포니아주 92821 브레아 스위트 200 새턴 스트리트 3030
자스윈더 싱 라지푸트
미국 캘리포니아주 92821 브레아 스위트 200 새턴 스트리트 3030

(74) 대리인 박종혁
김정욱
정삼영
송봉식

전체 청구항 수 : 총 33 항

(54) 다중 커널을 동시에 실행하는 방법 및 시스템

(57) 요약

공통 인터럽트 핸들러 및 선택적인 공통 스케줄러를 이용하여 동시에 다중 커널을 실행시키는 접근방식이 제공된다. 또한 커널들 사이에 실행을 스위칭하는 기술이 제공된다. 인터럽트 마스크 레벨을 이용한 커널들 사이의 실행 및 인터럽트의 선점이 도시된다. 또한 상이한 커널들에서 실행되는 태스크들 사이의 리소스 공유를 위한 기술들이 제공된다.

대표도

도 8

특허청구의 범위

청구항 1.

멀티-커널 환경에서 다중 커널을 동시에 실행시키는 방법에 있어서,

멀티-커널 환경으로부터 프라이머리 커널을 선택하는 단계;

상기 프라이머리 커널을 시작하는 단계;

상기 프라이머리 커널의 적어도 부분적인 제어하에 있는 적어도 하나의 세컨드리 커널을 추가하는 단계; 및

상기 프라이머리 커널 및 적어도 하나의 상기 세컨드리 커널에서 인터럽트와 인터럽팅 프로세스의 실행을 핸들링하는 인터럽트 핸들러를 공통 인터럽트 핸들러가 되도록 하는 단계를 포함하는 것을 특징으로 하는 방법.

청구항 2.

제 1 항에 있어서,

상기 프라이머리 커널은 범용 운영체제가 될 수 있는 것을 특징으로 하는 방법.

청구항 3.

제 1 항에 있어서,

상기 적어도 하나의 세컨드리 커널 중 적어도 하나는 실시간 운영체제가 될수 있는 것을 특징으로 하는 방법.

청구항 4.

제 1 항에 있어서,

프라이머리 커널로 선택된 커널은 가장 바람직한 능력을 가진 멀티-커널 환경에서의 커널인 것을 특징으로 하는 방법.

청구항 5.

제 1 항에 있어서,

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널에서 보류중인 프로세스의 실행을 스케줄링하는 스케줄러를 공통 스케줄러가 되도록 하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 6.

제 5 항에 있어서,

상기 공통 스케줄러는 가장 바람직한 능력을 가진 멀티-커널 환경에서의 운영체제로부터 선택되는 것을 특징으로 하는 방법.

청구항 7.

제 1 항에 있어서,

상기 공통 인터럽트 핸들러는 가장 바람직한 능력을 가진 멀티-커널 환경에서의 운영체제로부터 선택되는 것을 특징으로 하는 방법.

청구항 8.

제 5 항에 있어서,

상기 공통 인터럽트 핸들러 또는 상기 공통 스케줄러는 상기 프라이머리 커널에 있는 것을 특징으로 하는 방법.

청구항 9.

제 1 항에 있어서,

멀티-커널 환경을 실행하는 컴퓨터 부팅시에, 상기 프라이머리 커널은 상기 적어도 하나의 세컨드리 커널보다 먼저 시작되는 것을 특징으로 하는 방법.

청구항 10.

제 1 항에 있어서,

상기 적어도 하나의 세컨드리 커널 중에 적어도 하나는 상기 프라이머리 커널의 런타임 동적 모듈로서 활성화되는 것을 특징으로 하는 방법.

청구항 11.

제 1 항에 있어서,

고유 커널 식별을 상기 프라이머리 커널에 할당하는 단계; 및

상기 프라이머리 커널에 대해 허용되는 인터럽트를 결정하는 적어도 하나의 인터럽트 마스크 레벨을 상기 프라이머리 커널에 할당하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 12.

제 1 항에 있어서,

고유 커널 식별을 상기 적어도 하나의 세컨드리 커널에 할당하는 단계; 및

특정한 세컨드리 커널에 대해 허용되는 인터럽트를 결정하는 적어도 하나의 인터럽트 마스크 레벨을 적어도 하나의 세컨드리 커널에 할당하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 13.

제 5 항에 있어서,

상기 적어도 하나의 세컨드리 커널용 후크를 상기 프라이머리 커널의 상기 공통 스케줄러 또는 상기 공통 인터럽트 핸들러에 설치하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 14.

제 1 항에 있어서,

현재 활성화된 커널로부터 다음번 활성화될 커널로 프로세스 실행 제어를 스위칭하고, 상기 다음번 활성화될 커널은 상기 적어도 하나의 세컨드리 커널 중에 하나인 태스크를 실행하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 15.

제 14 항에 있어서,

상기 프로세스 실행 제어 태스크는 주기적인 태스크이고, 상기 다음번 활성화될 커널은 세컨드리 커널 폴링 우선순위 방식에 따라 실행할 적어도 하나의 보류중 프로세스를 가진 가장 높은 우선순위의 세컨드리 커널에 대해 상기 세컨드리 커널을 폴링함으로써 결정되고, 프로세스 실행 제어는 상기 적어도 하나의 세컨드리 커널에서 상기 보류중 프로세스중 적어도 일부를 완료한 후에 상기 프라이머리 커널로 전환되는 것을 특징으로 하는 방법.

청구항 16.

제 1 항에 있어서,

상기 공통 인터럽트 핸들러를 인보크하고, 그런다음 상기 공통 인터럽트 핸들러는 적어도 하나의 커널 독립적인 인터럽트 핸들링 기능을 실행하고, 그리고 프로세스 실행 제어를 상기 인터럽트에 연관된 타겟 커널의 인터럽트 서비스 루틴으로 전송하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 17.

제 16 항에 있어서,

상기 타겟 커널은 상기 적어도 하나의 세컨드리 커널의 마스크 레벨에 의해 결정되는 것을 특징으로 하는 방법.

청구항 18.

제 5 항에 있어서,

상기 공통 스케줄러를 인보크하는 단계;

멀티-커널 환경의 어떤 커널이 현재 실행중인 커널인지를 판정하는 단계;

프로세스 실행 제어를 현재 실행중인 커널로 전송하는 단계; 및

상기 현재 커널에 의해 적어도 하나의 커널 특정 스케줄링 기능을 실행하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 19.

제 1 항에 있어서,

프라이머리 커널이 프로세스 실행 제어를 적어도 하나의 세컨드리 커널 중 하나로 보내고 그에 의해 활성화된 커널이 되는 단계;

상기 프라이머리 커널이 자신의 인터럽트 마스크 레벨을 상기 활성화된 커널에 대응하도록 변경하는 단계; 및

상기 프라이머리 커널이 현재 실행중인 커널 식별 코드를 상기 활성화된 커널에 연관된 식별 코드로 변경하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 20.

제 1 항에 있어서,

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널 사이의 리소스 공유를 위해 애플리케이션 프로그램 인터페이스(API)를 설치하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 21.

멀티-커널 환경에서 다중 커널 사이의 시스템 리소스 공유를 위한 방법에 있어서,

멀티-커널 환경으로부터 프라이머리 커널을 선택하는 단계;

상기 프라이머리 커널을 시작하는 단계;

상기 프라이머리 커널의 적어도 부분적인 제어하에 있는 적어도 하나의 세컨드리 커널을 추가하는 단계; 및

상기 제 1 프라이머리 커널 또는 상기 적어도 하나의 세컨드리 커널과 상기 제 2 프라이머리 커널 또는 상기 적어도 하나의 세컨드리 커널 사이에 시스템 리소스 공유를 위한 애플리케이션 프로그램 인터페이스(API)를 설치하고, 상기 제 1 커널에는 적어도 하나의 제 2 커널에 대한 적합한 더미 API 호출이 제공되는 단계를 포함하는 것을 특징으로 하는 방법.

청구항 22.

제 21 항에 있어서,

상기 제 1 커널로부터의 상기 더미 API 호출에 의해 상기 제 2 커널이 활성화될 때, 상기 제 2 커널이 상기 더미 API가 정의된 커널에서의 상기 더미 API 호출을 상기 제 2 커널에 대한 실질적인 API 호출로 대체하는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 23.

제 22항에 있어서,

상기 제 1 커널로부터의 상기의 실질적인 API 호출시에, 상기 제 2 커널의 특정한 시스템 기능을 실행시키고, 그에 의해 상기 제 1 커널에서 상기 제 2 커널의 사용가능한 리소스를 만드는 단계를 더 포함하는 것을 특징으로 하는 방법.

청구항 24.

멀티-커널 환경에서 다중 커널을 동시에 실행시키는 시스템에 있어서,

멀티-커널 환경으로부터 프라이머리 커널을 선택하는 수단;

적어도 하나의 세컨드리 커널을 추가하고, 그를 적어도 부분적으로 제어하는 수단;

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널을 실행하는 수단; 및

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널에서 인터럽트와 인터럽팅 프로세스의 실행을 핸들링하는 수단을 포함하는 것을 특징으로 하는 시스템.

청구항 25.

제 24 항에 있어서,

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널에서 보류중 프로세스의 실행을 스케줄링하는 수단을 더 포함하는 것을 특징으로 하는 시스템.

청구항 26.

멀티-커널 환경에서 다중 커널 사이의 시스템 리소스 공유를 위한 시스템에 있어서,

멀티-커널 환경으로부터 프라이머리 커널을 선택하는 수단;

적어도 하나의 세컨드리 커널을 추가하고, 그를 적어도 부분적으로 제어하는 수단;

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널을 실행하는 수단; 및

제 1 프라이머리 커널 또는 상기 적어도 하나의 세컨드리 커널과 제 2 프라이머리 커널 또는 상기 적어도 하나의 세컨드리 커널 사이에 시스템 리소스 공유를 위한 수단을 포함하는 것을 특징으로 하는 시스템.

청구항 27.

멀티-커널 환경에서 다중 커널을 동시에 실행시키기 위한 컴퓨터 프로그램 제품에 있어서,

멀티-커널 환경으로부터 프라이머리 커널을 선택하는 컴퓨터 코드;

적어도 하나의 세컨드리 커널을 추가하고, 그를 적어도 부분적으로 제어하는 컴퓨터 코드;

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널을 실행하는 컴퓨터 코드;

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널에서 인터럽트와 인터럽팅 프로세스 실행을 핸들링하는 공통 인터럽트 핸들러를 구현하는 컴퓨터 코드; 및

상기 컴퓨터 코드를 저장하는 컴퓨터 판독가능한 매체를 포함하는 것을 특징으로 하는 컴퓨터 프로그램 제품.

청구항 28.

제 27 항에 있어서,

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널에서 보류중 프로세스의 실행을 스케줄링하는 공통 스케줄러를 구현하는 컴퓨터 코드를 더 포함하는 것을 특징으로 하는 컴퓨터 프로그램 제품.

청구항 29.

제 27 항에 있어서,

상기 컴퓨터 판독가능한 매체는 반송파, CD-ROM, 하드디스크, 플로피 디스크, 테이프 드라이브, 및 반도체 메모리에 수록된 데이터 신호로 구성된 그룹으로부터 선택된 것 중에 하나인 것을 특징으로 하는 컴퓨터 프로그램 제품.

청구항 30.

멀티-커널 환경에서 다중 커널 사이에 시스템 리소스를 공유하는 컴퓨터 프로그램 제품에 있어서,

멀티-커널 환경으로부터 프라이머리 커널을 선택하는 컴퓨터 코드;

적어도 하나의 세컨드리 커널을 추가하고, 그를 적어도 부분적으로 제어하는 컴퓨터 코드;

상기 프라이머리 커널과 상기 적어도 하나의 세컨드리 커널을 실행하는 컴퓨터 코드;

제 1 프라이머리 커널 또는 상기 적어도 하나의 세컨드리 커널과 제 2 프라이머리 커널 또는 상기 적어도 하나의 세컨드리 커널 사이에 시스템 리소스를 공유하는 컴퓨터 코드;

상기 제 1 커널에는 상기 적어도 하나의 제 2 커널에 대한 적합한 더미 애플리케이션 프로그램 인터페이스(API) 호출이 제공되는 컴퓨터 코드; 및

상기 컴퓨터 코드를 저장하는 컴퓨터 판독가능한 매체를 포함하는 것을 특징으로 하는 컴퓨터 프로그램 제품.

청구항 31.

제 30 항에 있어서,

상기 제 1 커널로부터의 상기 더미 API 호출에 의해 상기 제 2 커널이 활성화될 때, 상기 더미 API가 정의된 커널에서의 상기 더미 API 호출을 상기 제 2 커널에 대한 실질적인 API 호출로 대체하는 컴퓨터 코드를 더 포함하는 것을 특징으로 하는 시스템 리소스 공유방법.

청구항 32.

제 30항에 있어서,

상기 제 1 커널로부터의 상기의 실질적인 API 호출시에, 상기 제 2 커널의 특정한 시스템 기능을 실행시키고, 그에 의해 상기 제 1 커널에서의 상기 제 2 커널의 사용가능한 리소스를 만드는 컴퓨터 코드를 더 포함하는 것을 특징으로 하는 시스템 리소스 공유방법.

청구항 33.

제 30 항에 있어서,

상기 컴퓨터 판독가능한 매체는 반송파, CD-ROM, 하드디스크, 플로피 디스크, 테이프 드라이브, 및 반도체 메모리에 수록된 데이터 신호로 구성된 그룹으로부터 선택된 것 중에 하나인 것을 특징으로 하는 컴퓨터 프로그램 제품.

명세서

기술분야

본 발명은 일반적으로 멀티태스킹 운영체제에 관한 것이다. 보다 특징지어서는, 공통적인 인터럽트 핸들러 및 스케줄러를 이용하는 다중 커널의 실행을 허용함으로써 단일 운영체제에서 다중 커널의 특징을 지원하는 것에 관한 것이다.

배경기술

사용되는 특정한 애플리케이션에 기초하여 운영체제가 설계되고, 그의 동작이 일반적으로 그에 기초하여 최적화된다. 대개 다른 것에서도 사용가능한 어느 한 유형의 운영 체제의 특징을 가지는 것이 바람직하다.

예를 들면, 리눅스 및 윈도우즈와 같은 범용 컴퓨터 운영체제는 파일 시스템, 디바이스 드라이버, 애플리케이션, 라이브러리 등과 같은 특징의 확장 세트를 가지고 있다. 이러한 운영체제는 다중 프로그램의 동시 실행을 허용하고, 프로그램들을 동시에 서비스하는 것에 연관된 응답시간(또한 대기 시간이라고도 함) 및 CPU 사용, 또는 부하를 최적화하려고 한다. 그러나 불행히도, 이러한 운영체제는 일반적으로, 예를 들면, 로봇 제어, 텔레커뮤니케이션 시스템, 기계 툴, 자동 시스템 등과 같은 임베디드, 실시간 애플리케이션에는 적합하지 못하다. 이들과 다수의 기타의 것들과 같은 애플리케이션 기반의 실세계, 이벤트 및 제어는 하드 실시간 성능으로 알려진 것을 필요로한다. 하드 실시간 성능은 최악의 경우의 응답 시간을 보장한다. 범용 운영체제(GPOS)는 일반적으로 애플리케이션 프로그램의 평균적인 성능에 대한 프로그램 실행 시간의 예측 가능성을 훼손시킨다. iTRON™ 등을 포함하는 다수의 종래 실시간 운영체제(RTOS)는 하드 실시간 특징을 제공한다. 그러나, 유감스럽게도 대부분의 RTOS는 많은 GPOS의 특징을 가지고 있지 못하며, 예를 들면 상이한 파일 시스템, 디바이스 드라이버, 애플리케이션 라이브러리 등에 대한 지원을 제공하지 못한다. 많은 애플리케이션에서, RTOS의 성능과 범용 운영체제의 특징으로 갖는 것이 바람직할 것이다.

예를 들면, 리눅스는 현대적인 운영체제 특징들을 포함하는 현대적인 디바이스, 다수의 개발툴, 네트워킹 등에 대한 다수의 바람직한 특징을 가진 잘알려진 범용 운영체제이다. 그러나 리눅스는 임베디드 운영체제로 설계된 것은 아니다. 제한이 아닌, 셋탑박스, 모바일폰, 및 자동차 네비게이션 시스템 등과 같은 다수의 현대 디바이스들은 리눅스와 같은 범용 운영체제의 특징 뿐만 아니라, 실시간 성능과 같은 임베디드 운영체제의 특징 또한 필요로한다.

예를 들면, iTRON은 다수의 임베디드 디바이스에 일반적으로 사용되는 발달된 실시간 임베디드 운영체제이다. iTRON은 임베디드 디바이스에 바람직한 다수의 특징을 가졌지만, 네트워킹, 상이한 파일 시스템 등에 대한 지원과 같은 리눅스의 특징들은 결여되어 있다.

GPOS 및 RTOS 모두에 대한 예시적으로 필요한 것은, 자동차에 사용되는 네비게이션 시스템을 위한 컨트롤러이다. 상기 컨트롤러는 GPS 센서로부터 데이터를 판독하여 자동차의 위치와 방위를 연산한다. 네비게이션 데이터 DVD로부터 추출된 현재의 위치, 방향 및 위상의 맵에 기초하여, 컨트롤러는 최상의 경로를 연산하여, LCD 화면에 나타낸다. 상기 LCD 화면은 네비게이션 시스템에 파라미터를 입력하기 위한 터치 패널로 오버레이될 수 있다. 상기 판독 센서, 터치 패널 입력의 태스크는 하드 실시간을 필요로하고; 경로 연산, 그래픽 표시, DVD로부터의 판독 태스크들은 표준 프로그래밍 태스크이고, 범용 운영체제의 특징을 이용한다. 상기 하드 실시간 성능은 범용 태스크가 리눅스 커널 상에서 실행될 수 있는 동안 iTRON과 같은 RTOS 커널을 이용함으로써 달성될 수 있다.

또다른 예시로는 비디오 데이터 압축 하드웨어를 이용하는 고체 디지털 비디오 카메라용 컨트롤러가 필요하다. 이러한 애플리케이션에서는, LCD 화면에 표시하고 착탈가능한 저장 매체에 저장하는 동안, 압축 하드웨어로부터 오는 데이터 스트림을 판독하여 이미지 처리 기능을 실행하는 것이 바람직하다. 또한 예를 들면, 광학 줌 및 자동초점 메커니즘을 관리하는

동일한 제어 시스템을 이용할 필요가 있다. 상기 시스템이 일정한 레거시 컴포넌트를 이용한다면, 거기에는 이미 특정한 RTOS(예를 들면 iTRON)에 이용가능한 확장 제어 소프트웨어가 있을 것이다. 디스플레이, 이미지 처리 및 기타 기능들이 일반적으로 GPOS하에서 표준 프로그래밍에 의해 보다 잘 관리되는 반면에, 모터, 데이터 수집 및 저장을 제어하는 태스크는 하드 실시간 운영체제(hRTOS)에 의해 가장 잘 조정될 수 있다. 더구나, 일반적으로 iTRON에서 사용가능한 확장 소프트웨어를 다른 RTOS로 포팅하는 것은 너무나 비용이 많이 들어서, iTRON에서의 디스플레이, 및 파일 시스템 지원 등을 제공하는 것은 또한 간단한 것이 아니다. 따라서, 이러한 예에서. RTOS와 범용 운영체제의 강점을 결합하는 시스템이 이러한 애플리케이션에 최적이 될 것이다.

또다른 예로는 특정 기능의 가속 또는 특정 기능성의 추가를 위한 특별한 목적의 하드웨어를 이용하는 것이 필요한 시스템에서의 필요성이다. 예를 들면, 많은 멀티미디어 디바이스에서, 오디오 또는 비디오용 그래픽 가속칩 또는 DSP 또는 CODEC을 사용하는 것이 필요하다. 일부예에서, 추가적인 하드웨어에 대한 필요성은 운영체제가 일부 태스크에 대해 보장된 성능을 제공할 수 있다면 제거될 수 있다. 예를 들면, 스트리밍 오디오를 지원하는 시스템에서, 패킷 손실을 방지하고 일정한 질의 출력을 유지하기 위해 특정한 비율로 압축 및 인코딩된 오디오를 디코딩하는 것을 보장하는 고 성능 태스크를 가질 필요가 있을 것이다. GPOS 및 RTOS로 구성된 시스템은 일부 경우에 특화된 하드웨어에 대한 필요성을 제거하고, 그에 의해 생산 비용을 감소시킬수 있다.

모든 실세계 시스템은 하드 실시간(HRT), 소프트 실시간(SRT), 또는 비 실시간(NRT) 시스템 중 하나로 분류될 수 있다. 하드 실시간 시스템은 하나 이상의 활동은 반드시 데드라인 또는 타이밍 제한을 놓치지 말아야하며, 그렇지 않으면 그 태스크는 실패한 것으로 되는 시스템이다. 소프트 실시간 시스템은 타이밍 필요조건을 가지지만, 전체로서의 애플리케이션 필요조건에 계속해서 만족되는한, 때때로 그것들을 놓치는 것은 무시할만한 효과를 가지는 시스템이다. 마지막으로, 비 실시간 시스템은, 하드 실시간 시스템도 아니고 소프트 실시간 시스템도 아닌 시스템이다. 비 실시간 태스크는 데드라인이나 타이밍 제한을 가지지 않는다. 많은 현대 애플리케이션에서, 실시간 시스템 성능의 모든 스펙트럼을 지원하는 것이 필요하다. 예를 들면, 보안 애플리케이션에 대해서 네트워크 어플라이언스의 필요조건을 고려해야한다. 네트워크 어플라이언스는 하나의 패킷도 놓치는 일 없이(하드 실시간 태스크) 고속 네트워크 연결에 대한 모든 네트워크 패킷을 샘플링해야만 할 것이다. 하드 실시간 태스크는 이러한 패킷들을 추후에 처리하기 위해 버퍼에 저장할 것이다. 이것은 hRTOS를 이용하여 달성될 수 있다. 버퍼에서 이들 패킷 샘플들은 처리되고 분류되어야만 하지만, 상기 처리 및 분류가 더디어지더라도, 버퍼가 오버플로우가 아닌한은 문제가 되지 않는다(소프트 실시간 태스크). 이것은 hRTOS 및 GPOS에서의 태스크의 결합을 이용하여 달성될 수 있다. 요청시 상기 처리 및 분류된 데이터를 전달하기 위해 웹서버가 사용될 수 있다. 이러한 활동에는 일반적으로 타이밍 제한이 없으며(즉, 비 실시간 태스크); 따라서 이러한 태스크는 GPOS에서 수행될 수 있다.

상기와 같은 점을 고려하여, 효율적으로 그리고 편리하게 다중 커널의 성능 및 특징을 제공하고, 실시간 성능의 전체 스펙트럼을 지원하는 멀티-커널 환경(예를 들면 GPOS와 RTOS)을 구현하는 시스템에 대한 요구가 있다.

발명의 상세한 설명

상기 및 기타의 목적을 달성하기 위해, 그리고 본 발명의 목적에 따라, 다양한 기술이 다중 커널을 동시에 실행시키고, 다중 커널 사이의 리소스를 공유하기 위해 제공된다.

멀티-커널 환경에서 다중 커널의 동시 실행을 위한 방법, 시스템, 컴퓨터 코드 및 수단이 기술된다. 본 발명의 방법의 실시예에서, 프라이머리 커널과 적어도 하나의 세컨드리 커널이 설정되고, 적어도 하나의 세컨드리 커널은 프라이머리 커널의 적어도 부분적인 제어하에 있으며, 선택적인 공통의 스케줄러는 프라이머리 커널 및 세컨드리 커널중 적어도 하나에서의 보류중 프로세스의 실행을 스케줄링하도록 설정되고, 공통 인터럽트 핸들러는 프라이머리 커널 및 세컨드리 커널중 적어도 하나에서의 인터럽트와 인터럽팅 프로세스 실행을 조정하도록 설정된다. 또한 본발명의 다른 실시예에 따라 상기의 방법을 실행하기 위한 수단들이 제공된다. 또한 다른 실시예에 따라 상기 방법을 실행하기 위한 컴퓨터 코드가 제공된다.

멀티-커널 환경에서의 다중 커널 사이의 시스템 리소스를 공유하기 위한 본 발명의 또다른 방법의 실시예가 제공되고, 여기서 프라이머리 커널 및 적어도 하나의 세컨드리 커널이 설정되고, 적어도 하나의 세컨드리 커널은 적어도 부분적으로 프라이머리 커널의 제어하에 있으며, 커널들 사이의 시스템 리소스 공유를 위한 애플리케이션 프로그램 인터페이스(API)가 설정되고, 커널 호출이 다른 커널들 중 적어도 일부를 위한 적절한 더미 API 호출에 제공된다. 본 발명의 또다른 실시예에 따라 이러한 방법을 구현하기 위한 수단들이 제공된다. 또한 본발명의 또다른 실시예에 따라 이러한 방법을 구현하기 위한 컴퓨터 코드가 제공된다.

본 발명의 다른 특징, 이점, 및 목적은 첨부한 도면과 함께 판독되어야 하는 하기의 상세한 설명으로부터 보다 명확하고, 보다 쉽게 이해가 될 것이다.

실시예

본 발명은 본문에 설명된 상세한 도면 및 설명을 참조하여 가장 잘 이해된다.

본 발명의 실시예들이 도면을 참조하여 하기에 기술된다. 그러나, 당업자는 이들 도면에 대해 본문에 주어진 상세한 설명이 예시의 목적이고, 본 발명은 이들 제한된 실시예의 범위를 넘어섬을 쉽게 이해할 것이다.

하기에 보다 상세히 기술될 본 발명의 일 측면은 모든 운영체제 커널들의 특징 및 기능을 유지하면서 2 개 이상의 운영체제 커널을 동작시키는 것이다.

일반적으로, 멀티-커널 시스템을 개발하는 데에 대한 다수의 동기가 있을 수 있다. 그 4가지 이유는:

1. 하나의 커널의 성능 특성이 다른 것에서 바람직한 것이 될 수 있다(예를 들면, 실시간 기능성은 범용 운영체제에서 바람직할 수 있다.).

2. 하나의 운영체제(또는 커널)의 특징은 다른 것에서 바람직할 수 있다(예를 들면, 파일 시스템, 디바이스 드라이버, 실시간 API, 라이브러리).

3. 일부 경우, 멀티-커널 시스템을 사용함으로써 전용 하드웨어에 대한 필요성을 제거하고, 그에 의해 제품 비용을 감소시킬 수 있다.

4. 실시간 성능의 전체 스펙트럼을 지원할 수 있는 hRTOS 및 GPOS로 구성된 시스템에 대한 필요성이 있을 수 있다.

도 1은 본 발명의 일실시예에 따라 하나의 하드웨어 플랫폼 상에서 다중 커널들을 실행시킬 수 있는 예시적인 아키텍처의 도표를 도시한다. 상기 도면에 도시된 바와 같이, 커널0, 커널1, 커널2, 커널n이라고 이름을 붙인 다중 커널들이 "CPU"라고 이름을 붙인 일반적인 중앙처리장치 상에서 실행된다. 도면과 하기의 기재가 단일 CPU 시스템에 따른 실시예들과 예들을 도시하고 논의한다고 하더라도, 본 발명은 단일 CPU 구현에 한정되지 않으며, 본 발명의 교안과, 공지된 기술에 따라 멀티-CPU 시스템을 이용한 본 발명의 교안을 적절하게 실시하기에 적합하도록 구성될 수 있다는 것이 이해될 것이다. 본문에서 커널0을 프라이머리 커널로 하고, 커널들, 커널1, 커널2, ...커널n은 커널0에 의해 실행되는 특정한 번호의 커널들을 나타내며, 그 수는 일반적으로 시스템 리소스에 의해 제한될 수 있다. 상기 커널들은 범용 운영체제(GPOS) 또는 실시간 운영체제(RTOS)에 속하며, 각각 제공된 그의 특징 및 기능에 있어 폭넓게 다양하게 될 수 있다.

본 발명의 커널 선택의 측면이 하기에 보다 상세히 기술될 것이다. 도 2는 본 발명의 일실시예에 따라 다중 커널들을 동시에 실행하는 방법의 플로우 차트를 도시한다. 도시된 본 프로세스에서 각 스텝은 후속하는 도면에서 보다 상세하게 개별적으로 예를 들것이다. 상기 프로세스는 프라이머리 커널(커널0)로서 범용 운영 체제의 커널(예를 들면 도 1의 커널)을 선택함으로써 시작되고, 또는 최대한의 기능과 특징을 가진 운영체제가 스텝(210)에서 선택되고, 스텝(220)에서 시작된다. 본 예시에 따라, 커널을 시작하는 것은 하드웨어를 워밍업하고, 커널0에서 적절하게 로딩 또는 실행하는 부트로더를 로딩하는 것을 포함한다. 시작시 커널0은 인터럽트 핸들러, 스케줄러, 태스크 매니저 등을 시작시키고, 적절한 드라이버를 설치함으로써 시스템 하드웨어를 초기화한다. 다음으로, 스텝(230)에서, 프라이머리 커널에서 가용하지 못한, 아니면 바람직한 주어진 타겟 애플리케이션에 필요한 특정한 특징을 가진 커널들이 프라이머리 커널의 동적 모듈로서 추가된다(예를 들면 커널1, 커널2, ... 커널n). 이러한 프로세스는 원하는 커널들이 모두 추가될 때까지 스텝(230)에서 스텝(230)으로 반복된다.

각 세컨드리 커널은 바람직하게는 활성화시에 고유한 커널 식별 수단(ID)이 할당되고, 식별의 유용한 특징은 하기에 보다 상세히 예시될 것이다. 이러한 커널 ID들은 바람직하게는 미리할당된다. 스텝(240)에서, 추가된 커널은 미리할당된 인터럽트 마스크 및 커널 ID에 따라 프라이머리 커널에 의해 선택되고, 그런다음 스텝(250)에서, 추가된 또는 세컨드리 커널들, 커널1, 커널2...커널n이 동적 모듈로서 활성화된다.

도 3은 본 발명의 일실시예에 따라 도 2에 기술된 프라이머리 커널을 선택하는 예시적인 방법의 플로우차트를 도시한다. 상기 프로세스는 스텝(310)에서 가장 바람직한 기능과 특징을 가진 공통 인터럽트 핸들러를 선택하면서 시작한다. 그런

다음 스텝(320)에서, 상기 공통 인터럽트 핸들러가 속한 커널이 프라이머리 커널로 지정된다. 스텝(330)에서 스케줄러가 공통 스케줄러로서 선택된다. 이러한 스케줄러는 프라이머리 커널 또는 기타의 다른 커널의 스케줄러일 수가 있다. 특정 애플리케이션의 필요에 따라, 본 발명의 또다른 실시예가 구현되거나, 또는 프라이머리 커널 및/또는 본 시스템이 스케줄러의 임의의 적절한 인터럽트 핸들러를 이용할 수 있도록 할 수 있다. 본 발명의 다른 실시예에서, 프라이머리 제어 커널은 있을 수 없고, 대신에 멀티-커널 시스템의 모든 커널들이 공통의 태스크 스케줄러 및 공통의 인터럽트 핸들러에 의해 제어되며, 서로간에 직접 제어하지는 않는다. 본 발명의 다른 실시예에서, 프라이머리 커널의 인터럽트 핸들러는 사용될 수 없고, 대신에 또다른 인터럽트 핸들러가 프라이머리 커널의 외부에서 실시되며, 이러한 상황에서 인터럽트 핸들러 에뮬레이터는 종래 기술에 따라 구현될 수 있고, 그렇지 않으면 본 발명의 다른 새로운 측면들이 구현될 수 있다. 본 실시예에서, 그의 디폴트 인터럽트 핸들러가 공통 인터럽트 핸들러에 사용되는 커널이 자동으로 멀티-커널 시스템의 프라이머리 커널이 된다는 것이 이해될 것이다. 또한, 일부 애플리케이션에서, 시스템 설계자는 프라이머리 커널의 디폴트 인터럽트 핸들러를 제거하고 다른 인터럽트 핸들러로 그것을 대체하도록 선택할 수 있고, 어느 경우에건, 프라이머리 커널에 이용가능한 효과적인 인터럽트 핸들러가 본 실시예의 목적을 위한 상기 커널의 일부로 여겨진다는 것이 이해될 것이다. 본 발명의 교안에 따라 다수의 공통 인터럽트 핸들러용의 기타 적절한 구현의 변형이 당업자에게는 쉽게 명확하게 될 것이다.

도면으로 돌아가서, 고유 ID와 인터럽트 마스크 레벨이 스텝(340) 및 스텝(350)에서 프라이머리 커널에 각각 할당된다. 인터럽트 마스크 레벨은 하기에 보다 상세히 기술될 것이다.

도 4는 본 발명의 일실시예에 따라, 도 2에 기술된 프라이머리 커널을 시작하기 위한 예시적인 방법의 플로우차트를 도시한다. 상기 프로세스는 스텝(410)에서 상기 프라이머리 커널의 공통 인터럽트 핸들러를 설치하면서 시작한다. 그런 다음 스텝(420)에서 공통 스케줄러가 설치된다. 스텝(430)에서, 공통 애플리케이션 프로그램 인터페이스(API)가 리소스 공유를 위해 설치된다. 하나 이상의 세컨드리 커널이 있거나 또는 어떤 선택된 세컨드리 커널의 특정한 리소스가 리소스 공유에 사용가능할지는 사전에 공지되어있지 않다. 리소스 공유를 위한 공통 API는 리소스 API들의 상세를 미리 알지못하더라도 비제한적으로 리소스 공유를 허용한다. 스텝(440)에서, 특정 애플리케이션에 의존하는 원하는 스위칭 방식에 따라 세컨드리 커널로 실행을 스위칭 시키는 주기적인 태스크 또는 프로세스가 설치된다. 바람직한 실시예에서, 커널들 사이의 스위칭은 하드웨어 타이머 인터럽트에 의해 트리거된다. 특정 애플리케이션의 필요에 따라, 당업자는 본 발명의 교안에 따라 다른 적절한 스위칭, 아마도 불규칙적이거나, 이벤트 구동의 방식을 생각해낼 수 있다. 예시의 방법으로, 그러나 한정되는 것은 아닌, 주기적, 불규칙적, 이벤트 기반, 우선순위 기반의 방식을 포함하는 적절한 커널 스위칭 방식이 커널들 사이의 스위칭을 위해 사용될 수 있다.

도 5는 본 발명의 일 실시예에 따라, 도 2에 기술된 세컨드리 커널(들)의 선택 및 추가를 위한 예시적인 방법의 플로우 차트를 도시한다. 상기 프로세스는 스텝(510)에서 멀티-커널 소프트웨어가 설치될 시스템의 요구사항으로부터 도출된 바람직한 기능 및 특징의 세트를 가지는 세컨드리 커널을 선택하면서 시작한다. 고유 ID와 인터럽트 마스크 레벨은 스텝(520, 530)에서 세컨드리 커널에 각각 할당된다.

도 6은 본 발명의 일 실시예에 따라, 도 2에 기술된 세컨드리 커널을 시작하기 위한 예시적인 방법의 플로우 차트를 도시한다. 상기 프로세스는 스텝(610)에서 세컨드리 커널용 '후크'로 알려진 것을 프라이머리 커널의 공통 인터럽트 핸들러에 설치하면서 시작된다. 그런 다음 스텝(620)에서 세컨드리 커널용 후크가 공통 스케줄러에 설치된다. 스텝(630)에서, 세컨드리 커널용 후크는 공통 애플리케이션 프로그래밍 인터페이스(API)에 설치된다. 상기 후크는 프라이머리 커널에서의 제어 애플리케이션(예를 들면 인터럽트 핸들러 또는 스케줄러, 또는 공통 API)으로부터 상기 후크가 연관된 특정한 세컨드리 커널까지의 제어 경로를 이네이블하게 한다.

본 발명의 인터럽트 마스킹 및 커널 우선순위의 측면이 하기에 보다 상세히 기술될 것이다. 모든 현대 컴퓨팅 시스템은 선택적으로 이네이블되거나 디세이블하게될 수 있는 인터럽트들을 가진다. 인터럽트 마스크 레벨은 어떤 인터럽트가 허용되고, 어떤 것이 프로세서를 인터럽트하는 것이 허용되지 않는지를 판단한다. 도 7은 본 발명의 일실시예에 따라 다중 커널용의 공통 인터럽트 핸들러와 공통 스케줄러를 위한 예시적인 아키텍처의 블록도를 도시한다. 도면에서, 커널0을 위한 마스크 레벨은 모든 인터럽트가 허용되도록 된다. 본 발명의 바람직한 실시예에서, 각 세컨드리 커널들은 오직 커널이 실행될 때만 이네이블되는 범위의 인터럽트가 할당된다. 본 실시예에서, 이것은 마스크 레벨을 사용하여 달성된다.

대부분의 현대 프로세서는 인터럽트-마스크-레벨을 지원한다. 상술한 바와 같이, 커널 마스크 레벨은 어떤 인터럽트가 커널에 의해 허용되고, 어떤 것이 허용되지 않는지를 판단한다. 그러나, 커널에 의해 인터럽트가 허용되었다고 하더라도, 인터럽트가 그에 의해 핸들링될 수는 없다는 것에 유의해야한다. 따라서 본 실시예는 커널과 인터럽트에 대해 3 개의 인터럽트 조건을 가진다: (1) 인터럽트는 블로킹될 수 있고, (2) 인터럽트는 허용되지만 처리될 수는 없고, 및 (3) 인터럽트는 커널에 의해 허용되고 처리(핸들링)될 수 있다. 특정한 커널에 의해 허용되고 핸들링되는 인터럽트는 그 커널에 할당되는 것으로 본다. 다시, 모든 인터럽트는 커널0에 의해 허용되고 핸들링된다. 각 인터럽트는 또한 다른 커널에 고유하게 할당될

수 있다. 따라서, 본 실시예의 접근방식 하에서, 인터럽트는 커널0에 의해 허용되고 핸들링되어야 하고, 오직 하나의 다른 커널에 의해서만 허용되고 핸들링될 수 있다. 본 발명의 일부 실시예는 또한 우선순위를 가진 인터럽트를 제공하고, 이 우선순위는 CPU 설계, 또는 당업자에 알려진 다른 수단에 의해 명령을 받을 수 있다.

본 실시예의 일반적인 애플리케이션에서, 멀티-커널 시스템의 설계동안, 인터럽트 우선순위는 바람직하게는 가장 높은 우선순위의 인터럽트가 가장 높은 실행 우선순위를 가진 커널에 할당되도록 지정된다. 도면에 도시된 바와 같이, 커널1은 커널0보다 높은 우선순위를 가지고, 커널2는 커널1보다 높은 우선순위를 가진다. 커널n은 가장 높은 우선순위를 가진다. 따라서, 커널0은 커널1, 커널2....커널n에 의해 선점될 수 있다. 커널1은 커널2 내지 커널n에 의해 선점될 수 있다. 커널n은 어떠한 커널에 의해서도 선점될 수 없다. 다른 대안의 그리고 적절한 인터럽트 우선화 방식은 본 발명의 교안에 따라 당업자에 쉽게 명확하게 될 것이다.

본 발명의 인터럽트 핸들링 측면은 하기에 보다 상세하게 기술될 것이다. 본 발명의 신규한 측면은 공통 인터럽트 핸들러가 먼저 선택된다는 것이다. 공통 인터럽트 핸들러에 연관된 커널을 프라이머리 커널이라고 한다. 바람직한 실시예에서, 모든 인터럽트는 커널0 인터럽트 핸들러에 의해 핸들링된다. 인터럽트를 받을 때(710), 커널0은 비-커널 특정 인터럽트 서비스 루틴을 실행하고 그런다음, 제어를 특정 인터럽트가 할당된 커널의 인터럽트 핸들러로 보낸다. 다시 도면을 참조하면, 커널n에 할당된 인터럽트 N이 발생했을 때, 그것은 먼저 커널0 핸들러에 의해 핸들링되고, 그런 다음 커널 n의 인터럽트 서비스 루틴이 인보크되고, 이 경우 커널n은 본문에서 타겟 커널이 된다고 한다(720). 인터럽트 핸들러가 인보크될 때, 인터럽트 핸들러는 커널에 독립적인 인터럽트 핸들링 기능을 실행하고; 타겟 커널의 인터럽트 서비스 루틴으로 제어를 보낸다는 것에 유의해야한다. 상기 타겟 커널은 바람직하게는 인터럽트 마스크 레벨을 이용하여 식별된다. 이러한 방식으로, 커널0의 인터럽트 핸들러는 멀티-커널 시스템용 공통 인터럽트 핸들러로서 역할을 한다.

도 8은 본 발명의 일실시예와 도 7에 따른 다중 커널용 인터럽트 마스크 레벨의 예시적인 도식 차트를 도시한다. 상기 도면은 본 실시예에서 인터럽트 마스크 레벨이 각 인터럽트에 대한 타겟 커널을 결정하기 위해 어떻게 사용되는지를 도시한다. 인터럽트 번호는 차트의 좌측 또는 축에, 인터럽트의 총 수가 N인 오름차순의 수로 도시된다.

수직 바의 점선(또는 대개는 실선) 영역(810)은 각각의 커널에 의해 핸들링되고 허용되는 인터럽트들을 도시한다. 해치된 영역(820)은 각각의 커널에 의해 허용된 인터럽트들을 도시한다. 벽돌모양으로 텍스처된 영역(830)은 각 커널들이 실행될 때, 즉 CPU 시간을 관리하고 있을때, 블로킹된 인터럽트들을 도시한다.

도 9는 본 발명의 일실시예에 따라, 도 2의 블록도 및 도 8의 바 차트에 도시된 다중 커널용 인터럽트 마스크 레벨의 추가적인 측면을 도시한다. 어떤 인터럽트가 커널에 의해 처리될 수 있는지, 어떤 인터럽트가 커널에 의해 디세이블될 수 있는지 및 주어진 커널에 의해 어떤 인터럽트가 이네이블 될 수 있는지를, 마스크 레벨이 어떻게 판단하는지의 예가 도면에 도시된다. 상기 인터럽트 번호는 차트(910)의 좌측 또는 축에, 인터럽트의 총수가 N인 오름차순의 번호로 도시된다.

특히, 가장 우측 바의 수직 바에서 Ki로 표시된 i번째 커널, 및 'ai'(920)는 커널 Ki에 의해 이네이블될 수 있는 인터럽트를 지시하고, 'bi'(930)는 커널 Ki에 의해 디세이블될 수 있는 인터럽트를 지시하고, 'ci'(940)는 커널 K에 의해 처리된 인터럽트를 지시한다.

본 발명의 스케줄링의 측면은 하기에 보다 상세히 기술될 것이다. 가장 일반적인 운영체제에서, 스케줄러는 하드웨어 타이머를 이용하여 주기적으로 인보크된다. 하드웨어 타이머는 일반적으로 스케줄링 이벤트를 초기화하기 위해 주기적인 인터럽트를 트리거하도록 설정된다. 멀티-커널 시스템에서 각 커널은 운영체제의 목적에 따라 스케줄러를 인보크하기 위해 상이한 주기를 가질 수 있다. 제한이 아닌 예로써, 범용 운영체제의 경우에, 10 밀리초 주기는 원하는 성능에 충분할 수 있다. 그러나, 실시간 커널의 경우, 100 마이크로초마다 스케줄링 이벤트를 갖는 것이 필수적이 될 수 있다.

본 실시예에서, 공통 스케줄러가 멀티-커널 시스템에 대해 선택될 수 있다. 모든 스케줄링 이벤트는 바람직하게는 공통 스케줄러에 의해 먼저 수신된다. 커널 독립적인 스케줄링 기능을 실행한 후에, 상기 스케줄러는 바람직하게는 현재 실행중인 커널(730)의 스케줄러로 제어를 보낸다. 이러한 예의 목적을 위해, 현재 실행중인 커널이 스케줄링 이벤트가 발생했을 때 실행중이었던 커널로 정의된다.

본 발명의 멀티-커널 실행의 측면이 하기에 보다 상세히 기술될 것이다. 본 발명의 다른 신규한 측면은, 보다 높은 순위의 커널이 실행될 때 조차, 시스템은, 기회가 있을때(즉, 높은 우선순위의 커널에서의 태스크가 실행중인 상태가 아닐때- 예를 들면, 대기, 슬리핑, 휴면...등), 보다 낮은 우선 순위의 커널들에서의 태스크의 실행을 허용한다는 것이다.

도 10은 본 발명의 일 실시예에 따라, 주기적인 신호에 의해 커널을 스위칭하는 예시적인 방법의 플로우 차트를 도시한다. 도면에 도시된 실시예에서, 범용 커널(커널0, 도시되지 않음)은 커널 스위칭 프로세스를 트리거하는 스텝(1005)에서의 주기적인 신호 생성에 의해 하나의 커널에서 다른 커널로 스위칭하는 주기적인 프로세스를 실행하고, 그에 의해 상기 프로세스는 먼저 다른 커널에 보류중 태스크가 있는 지를 판정하도록 진행된다. 이것은 다수의 적절한 접근 방식에 의해 달성될 수 있으며; 시리얼 폴링 방법론이 체인에서의 커널이 실행할 보류중 태스크를 가지고 있는 지를 판정하기 위해 사용되는 하나의 적절한 접근 방식이 도면에 도시된다. 도시된 예에서, 스텝(1005)에서의 주기적 신호의 생성시에, 커널1은 실행할 보류중 태스크에 대해 폴링된다. 커널 1이 실행할 하나 이상의 보류중 태스크를 가진다면('예' 경로), 커널1에서의 보류중 태스크(들)의 실행은 예를 들면 현재 실행중인 id를 커널1의 id로 변경하고 그에 의해 CPU 시간을 보류중 태스크(들)의 실행으로 전환함으로써 유효하게된다. 커널1이 실행할 보류중 태스크가 없다면('아니오' 경로), 상기 프로세스는 다음의 커널;예를 들면 스텝(1020)에서 커널2를 폴링하도록 진행되고, 상기 프로세스는 마지막 커널, 스텝(1030)에서 커널n에 도달할 때까지 체인에서의 각각의 연속하는 커널에 대해 동일한 방식으로 진행한다. 커널에서 모든 보류중 태스크가 실행되거나 또는 보류중 태스크가 발견되지 않으면(즉, 스텝(1010-1030) 전체의 "아니오"경로), 프로세스는 종료하고, 커널0 태스크의 실행이 재개된다. 그러나, 본 발명의 일부 대안의 실시예에서, 프라이머리 커널로 제어를 리턴하기 전에 모든 보류중 태스크를 완료해야만 하는 대신에, 다른 폴링 또는 스위칭 방식이 상기 프라이머리 커널로 다시 제어를 리턴하기 전에 보류중 프로세스의 적어도 일부를 서비스하기 위해 종래 공지된 기술(제한이 아닌 예로써, 먼저 가장 높은 수선순위의 커널을 수행하고, 그런다음 연속한 순서로 보다 낮은 우선순위의 커널을 수행하는 등)에 따라 구현될 수 있다. 상기 프로세스는 스텝(1005)에서 주기적인 신호의 다음번 생성시에 재시작된다. 특정한 애플리케이션의 필요에 따라, 당업자는 본 발명의 교안에 따라 다수의 대안 및 적절한 스위칭 방식을 인식할 것이다.

본 발명의 멀티-커널 리소스 공유의 측면이 하기에 보다 상세히 기술될 것이다. 그러나 본 발명의 또다른 신규한 측면은 리소스가 프라이머리 커널과 세컨드리 커널 중 어느 하나 사이 및 세컨드리 커널들 사이에 공유될 수 있다는 것이다. 많은 애플리케이션에서 다른 것으로부터 하나의 운영체제 커널의 특징 및 리소스에 액세스하는 것이 대개 바람직하다. 일부의 예에서, 이것은 멀티-커널 시스템을 구현하기 위한 주된 동기가 될 수 있다.

도 11은 공통 시스템 API가 다중 커널 리소스 공유에 사용되는 본 발명의 일 실시예의 예시적인 블록도를 도시한다. 본 실시예에서, 다중 커널 리소스 공유는 프라이머리 커널과 세컨드리 커널 사이의 리소스 공유(예를 들면, 파일 시스템, 디바이스 드라이버, 라이브러리 등)를 지원하는 각 커널에 대해 더미 API 시스템 호출(예를 들면 Sys_call 1, Sys_call 2.....Sys_call n) 정의에 의해 달성된다. 바람직한 실시예에서, 상기 프라이머리 커널은, 상기 프라이머리 커널이 프라이머리 커널과 세컨드리 커널 사이의 리소스 공유를 지원하는 각 커널에 대한 더미 API 호출을 가진다. 상기 더미 API 호출은 상기 세컨드리 커널이 상기 프라이머리 커널의 동적 모듈로서 활성화될 때 실질적인 API 호출에 의해 대체된다.

실질적인 API 호출이 프라이머리 커널로부터 실행될 때, 상기 세컨드리 커널은 그 API에 대응하는 특정한 함수의 호출을 인보크하고 상기 세컨드리 커널하에서 상기 함수를 실행한다. 이러한 방식으로, 세컨드리 커널의 API는 상기 프라이머리 커널에 이용가능하도록 만들어진다. 따라서, 애플리케이션(사용자 및 시스템)은 상기 세컨드리 커널의 특징에 액세스할 수 있다. 사용자 애플리케이션(1110)이 프라이머리 커널(1120)이 커널n의 API를 실행할 것을 요청할 때, 프라이머리 커널(1120)은 커널n(1140)의 Sys_calln에서 리소스 공유를 위한 공통 API를 호출하기 위해 리소스 공유 Sys_call0(1130)에 대한 공통 API를 사용한다. 커널n, Sys_call n(1150)에서의 리소스 공유를 위한 상기 공통 API는 사용자 애플리케이션에 의해 요청된 특정한 API를 호출한다.

리눅스(GPOS), iTRON(RTOS) 운영체제에 적용되는 것으로 본 프로세스의 예를 든 본 발명의 특정한 실시예가 하기에 논의될 것이다. 당업자는 본 발명의 교안에 따라 운영하기에 적합한 GPOS 및 RTOS를 적절하게 설정하는 방법을 쉽게 인지할 것이다. 본 발명에 의해 인지된 바와 같이, 리눅스 커널과 같은 GPOS와 iTRON 커널과 같은 RTOS를 포함하는 하이브리드 시스템은 많은 현대의 임베디드 디바이스에 가장 바람직한 특징을 가질 것이다.

상기의 교안에 따라, 본 실시예에서, 리눅스 커널이 범용 운영 커널(k0)로 선택되고, iTRON이 세컨드리 커널(k1)로 선택된다. 리눅스의 스케줄러가 공통 스케줄러로 선택되고, 리눅스의 인터럽트 핸들러가 시스템의 공통 인터럽트 핸들러로 각각 선택된다. 컴퓨터를 부팅할 때, 리눅스 커널이 먼저 시작된다. 상기 iTRON 커널은 리눅스 커널의 런타임 동적 모듈로서 삽입된다. 예를 들면 고유한 커널 ID(0 및 1)가 각각 리눅스와 iTRON에 할당된다. 상기 iTRON 커널1은 인터럽트 마스크 레벨(11-15)(예를 들면, Hitachi SH-4 구현에 적당함)이 할당될 수 있다. 따라서, 예를 들면 iTRON 커널은 인터럽트가 허용되지 않은 마스크레벨(1-10)을 가진 인터럽트를 실행한다.

리눅스 스케줄러가 시스템의 공통 스케줄러로 사용되기 때문에, 상기 리눅스 스케줄러는 주기적으로 하드웨어 타이머를 이용하는 시스템에 의해 인보크된다. 스케줄링 이벤트가 트리거되면, 리눅스 스케줄러가 인보크된다. 상기 리눅스 스케줄러는 스케줄러가 인보크될 때 실행하고 있던 커널의 커널 ID를 판단한다. 실행중인 커널이 리눅스였다면, 예를 들면 하기에 유사 코드로 예를 든 linux_schedule() 함수가 호출된다:

```
#define DUET_NUM_KERNELS 2 /* 커널이 2개라고 가정(하나의 프라이머리 커널은 리눅스, 하나의 세컨드리는 itron) */

#define DUET_NUM_POINTERS 3

typedef int (*duetptr)(unsigned long);

typedef int (*duetptr2)(signed long, unsigned long, unsigned long*, unsigned long*);

int linux_do_IRQ(unsigned long);

int duet_running_kernel=0;

int linux_sys_call(signed long function_id, unsigned long argc, unsigned long * arg_type, unsigned long * arg_arr)

{

return 0;

}

duetptr duetptrarr [DUET_NUM_KERNELS][DUET_NUM_POINTERS]=

{

{linux_schedule, linux_do_IRQ, 0},

{0, 0, 0}

};

duetptr2 duetptr2arr [DUET_NUM_KERNELS][DUET_NUM_POINTERS]=

{

{linux_sys_call, 0, 0},

{0, 0, 0}

};

asmlinkage int_sys_duet_sys_call(signed long kernel_id, signed long function_id, unsigned long argc, unsigned long * arg_type, unsigned long *arg_arr)

{

if(duetptr2arr[kernel_id][2])
```

```
return duetptr2arr[kernel_id][2](function_id, argc, arg_type, arg_arr);

else

return-200; /*Invalid Kernel*/

}

asmlinkage void schedule(void)

{

duetptrarr[duet_running_kernel][0](0);

}
```

어떤 커널이 실행중인가에 따라, 특정한 인터럽트가 마스킹될 수 있다. 예를 들면, iTRON 커널이 실행중 일때, 마스크 레벨(1-10)(예를 들면, SH-4 구현)을 가진 모든 인터럽트가 마스킹된다. 마스크 레벨(11-15)을 가진 인터럽트가 발생하면, 리눅스 인터럽트 핸들러가 인보크된다. 리눅스 인터럽트 핸들러는 비-iTRON 특정 코드를 실행시키고, 그런다음 하기 유사 코드에 의해 예를 든 것처럼, do_IRQ를 이용하여 iTRON 인터럽트 핸들러를 실행시킨다:

```
asmlinkage int do_IRQ(unsigned long r4, unsigned long r5,

unsigned long r6, unsigned long r7,

struct pt_regs regs)

{

return duetptrarr[duet_running_kernel][1]((unsigned long)&regs);

}
```

본 실시예에서, 세컨드리 커널(iTRON과 같은)이 설치될 필요가 있다면, 프라이머리 커널은, 그의 목적이 커널들 사이에서 실행을 스위칭하는 것인 주기적인 신호를 먼저 설치한다. 이 주기적인 신호는 하드웨어 타이머에 의해 트리거될 수 있다. 이러한 주기적인 신호가 발생할 때, 인터럽트 핸들러는 세컨드리 커널(iTRON)에서 임의의 태스크 보류중 실행이 있는지 여부를 판단하고, 없다면, 그것은 리눅스 커널로 실행을 보낸다. 이것은 세컨드리 커널이 유휴시간인 동안 프라이머리 커널에서의 태스크의 실행을 허용한다.

본 실시예에서, 프라이머리 커널(예를 들면 리눅스 커널)이 세컨드리 커널(예를 들면 iTRON 커널)로 실행을 보내면, 그것은 바람직하게는 먼저 세컨드리 커널(iTRON)의 인터럽트 마스크 레벨로 인터럽트 마스크레벨을 변경한다. 제한이 아닌 예로써, 실행이 iTRON으로 전환될 때, 인터럽트 마스크 레벨은 하기에 도시된 것과 같은 linux_2_itron()을 인보킹함으로써 설정된다. 이것은 인터럽트 마스크를 0x000000A0에 설정한다. 지금은 오직 11-15 사이의 인터럽트만이 허용될 것이다. 마스크 레벨(0-10)을 가진 인터럽트가 발생하면, 상기 인터럽트는 하기의 유사 코드에 의해 예시된 바와 같이 무시된다:

```
void linux_2_itron(void)

{

/*마스킹*/

duet_imasks=0x000000A0;
```

```
/*iTRON 스케줄, IRQ 설정*/

duet_running_kernel=1;

}
```

실행이 iTRON으로부터 리눅스로 다시 전환될 때, 인터럽트 마스크는 모든 인터럽트를 허용하는 0x00000000에서 설정된다. 실행이 전환되기 전에 커널 id는 또한 상기 실행이 보내질 커널의 id로 변경된다는 것에 유의해야한다. 예를 들면, 실행이 리눅스에서 iTRON으로 전달될 때, 커널id는 0에서 1로 변경된다. 실행이 리눅스로 리턴될 때, 하기의 유사 코드에 의해 예시된 바와 같이, 커널id는 1에서 0으로 변경된다:

```
void itron_2_linux(void)

{

/*리눅스 스케줄, IRQ 설정*/

duet_running_kernel=0;

/* 언마스킹 */

duet_imasks=0x00000000;

}
```

상기 시스템은 Hitachi SHx 군의 프로세서 상에서 구현된다. Hitachi SHx 프로세서 및 다른 많은 프로세서들은 명시적인 인터럽트 우선순위를 지원한다. 인터럽트 우선순위가 하드웨어에서 지원되지 않는 시스템에서, 인터럽트 우선순위는 에뮬레이션 또는 기타 다른 기술을 통해 소프트웨어로 구현될 수 있다.

대부분의 종래 실시간 임베디드 시스템은 프로그래밍; 즉, 특정 이벤트가 발생할 때 태스크가 실행하는 것에 기반을 둔 이벤트를 이용한다. 잘-프로그래밍된 임베디드 컴퓨터 시스템에서, 시스템 CPU는 대부분의 시간을 유휴상태로 휴지할 것이다. 모두는 아닐지라도, 대부분의 임베디드 애플리케이션은 3 유형의 태스크; 하드 실시간(HRT), 소프트 실시간(SRT), 및 비 실시간 또는 보통의(NRT) 태스크로 구성되는 것으로 보여질 수 있으며; 이 태스크 모델 및 대응하는 인터럽트 모델이 하기에서 보다 상세히 기술될 본 발명의 일 실시예에서 레버리지될 것이다. 본 문에서, 본 발명의 또다른 측면은 범용 운영체제의 성능과 충격계수를 증가시키기 위해 임베디드 시스템에서의 이러한 일반적인 유휴 시간을 이용한다. 상기 태스크 모델 및 시스템 유휴 시간을 레버리지하는 본 발명의 일 실시예에서, HRT 태스크는 제한이 아닌 예로써, iTRON API를 이용하는 iTRON 커널 등의 RTOS 커널(들)에서의 태스크로서 구현되고; SRT 태스크는 제한이 아닌 예로써, iTRON API등의 RTOS 커널(들) 및/또는 제한이 아닌 예로써 리눅스 라이브러리(시스템 호출 및 커널 API)등의 GPOS 커널(들)을 이용하여 구현되고; NRT 태스크는 제한이 아닌 예로써, 표준 리눅스 API등의 GPOS 커널(들)을 이용하여 구현된다.

본 실시예는 공지의 또는 개발될 RTOS 및 GPOS 시스템의 조합으로 사용하는 데에 적합하지만; 명확하게 하기 위해, 후속 논의에서는 RTOS는 iTRON으로, GPOS는 리눅스로 가정할 것이다. 본 발명의 접근 방식에 따라, iTRON 커널에서 태스크 보류중 실행이 있는한, 리눅스 프로세스는 실행될 기회를 가지지 못한다. 실행을 위해 준비된 하나 이상의 태스크가 있다면, 가장 높은 우선순위의 태스크가 먼저 실행되고, 준비 또는 보류 상태의 태스크가 더 이상 없을 때까지 그 다음으로 높은 우선순위의 태스크가 그 다음으로 실행된다.

iTRON 시스템에서 태스크 보류중 실행이 없는 경우, 실행 제어가 리눅스로 전달되고, 다시 여기서 가장 높은 실행 우선 순위의 태스크가 먼저 실행된다. 대기시간을 알맞게 작게 유지하기 위해, 모든 SRT 태스크는 표준 리눅스 프로세스(즉, NRT 태스크) 보다 더 높은 실행 우선 순위를 갖는다. 바람직한 실시예에서, SRT와 NRT 사이의 리눅스에서의 우선순위 시스템은 리눅스 'RT 우선순위'를 이용하여 구현된다. 따라서, NRT 프로세스는 SRT 태스크 보류중 실행이 없을때까지 실행되지 않는다. 또다른 실시예에서, 임의의 적절한 공공영역 또는 개인소유의 우선순위 관리 시스템이 상기 HRT, SRT, 및 NRT 프로세스의 우선순위 및 스케줄링을 관리하도록 구현될 수 있다.

상술한 바와 같이, 본 발명의 또다른 신규한 측면은 다중 커널이 파일 시스템, 디바이스 드라이버, 라이브러리 등과 같은 리소스를 공유할 수 있는 프로세스이다. 본 발명의 일실시예에서, 이러한 리소스 공유 프로세스는 리소스 공유가 지원되는 각 커널에 대해 더미 API 호출을 정의함으로써 달성될 수 있다. 제한이 아닌 예로써, RTOS 커널, 예를 들면 iTRON에서 사용가능하고, GPOS 커널, 예를 들면 리눅스로부터의 특징을 사용하는 것이 매우 바람직하다. iTRON 커널에 대한 더미 API 호출이 하기에서 유사 코드로 제한이 아닌 예시의 방법으로 제시된다:

```
#define ITRON_BAS_ERR 300

int itron_syscall(signed long function_id, unsigned long argc, unsigned long * arg_type, unsigned long * arg_arr)

{ switch(function_id)

---------- ----------

/****************************************/

/* 함수 코드 */

/****************************************/

/*섹션 4.1 태스크 관리 서비스 호출*/

case TFN_CRE_TSK:/*(-0x05)*/

case TFN_DEL_TSK:/*(-0x06)*/

case TFN_ACT_TSK:/*(-0x07)*/

/*섹션 4.4.1 세마포어 서비스 호출*/

case TFN_CRE_SEM:/*(-0x21)*/

return(cre_sem((ID)arg_arr[0], (T_CSEM*)arg_arr[1])-ITRON_BAS_ERR);

case TFN_DEL_SEM:/*(-0x22)*/

return(del_sem((ID)arg_arr[0])-ITRON_BAS_ERR);

case TFN_SIG_SEM:/*(-0x23)*/

return(sig_sem((ID)arg_arr[0])-ITRON_BAS_ERR);

default:

return INVALID_FUNCTION }

}

----------
```

세컨드리 커널들이 동적 런타임 모듈로서 가장 먼저 활성화(로딩)될 때, 더미 API 호출이 실제 API에 링크된다. iTRON이 동적 모듈로서 리눅스하에서 활성화될 때, 더미 API 호출은 실제 API 호출에 의해 대체된다. 이러한 방식으로, 전체 세컨드리 커널(예를 들면 본 예에서 iTRON) API가 하기의 유사 코드에서 제한이 아닌 예시된 바와 같이 프라이머리 커널(예를 들면 본 예에서 리눅스)에 사용가능하게 된다:

```
asmlinkage int sys_duet_sys_call(signed long kernel_id, signed long function_id, unsigned long argc, unsigned long *arg_type, unsigned long *arg_arr)

{

if(duetptr2arr[kernel_id][2])

return duetptr2arr[kernel_id][2](function_id, argc, arg_type, arg_arr);

else

return INVALID_KERNEL_;/* 유효하지 않은 커널 */

}

duetptr2 duetptr2arr[DUET_NUM_KERNELS][DUET_NUM_POINTERS]=

{

{linux_sys_call, 0, 0},

{0, 0, 0}

};
```

본 실시예를 리눅스의 런타임 동적 모듈에 대해 활성화하기 위해, 하기의 유사 코드가 제한이 아닌 예시의 방식으로 이용될 수 있다:

```
void duet_init_itron(void)

/* 듀엣 스케줄, IRQ 설정*/

duetptrarr[1][0]=itron_schedule;

duetptrarr[1][1]=itron_IRQ;

duetptr2arr[1][2]=itron_syscall;/*itron_syscall 설치*/

duet_imaskc=0x000000D0;

duet_imasks=0x00000000;}
```

RTOS 모듈, 예를 들면 iTRON이 제거될 때, 더미 API는 하기의 유사코드에서 제한이 아닌 예시된 바와 같이 제거된다:

```
void duet_deinit_itron(void)

{
```

duetptr2arr[1][2]=0;/*itron_syscall을 언인스톨*/

duet_imaskc=0x000000F0;

duet_imasks=0x00000000;

}

더미 API 호출을 사용함으로써 이러한 방식 또는 유사한 방식으로, 프라이머리 커널은 상기 더미 API를 통해 프라이머리 커널에 한정하여 사용가능하게 만들어지는 세컨드리 커널 함수를 실행할 수 있다. 이러한 메커니즘은 데이터 공유, 태스크 동기화 및 통신 기능(세마포어, 이벤트 플래그, 데이터 큐, 메일 박스)을 포함하지만 그에 한정되지 않는 2 개 커널들 사이의 복잡한 상호작용을 사용할 수 있게한다. 더미 API와 GPOS(예를 들면 리눅스) 시스템 호출을 사용함으로써, 당업자는 본 발명의 교안에 따라, 실시간 임베디드 프로그램에서의 리눅스의 풍부한 특징(예를 들면, 파일 시스템, 드라이버, 네트워크 등)에 엑세스할 수 있는 프로그램을 개발할 수 있다. 본 발명의 일부 실시예들은 상기의 공통 스케줄러 및/또는 공통 더미 API를 포함하지 않을 수 있는데, 그것들은 선택사항이기 때문이다. 즉, 본 발명의 공통 인터럽트 핸들러를 가지고 다중 커널이 공통 스케줄러 및/또는 공통 더미 API없이 실행될 수 있다. 그러나, 많은 애플리케이션에서, 공통 스케줄러는 증가된 성능과 보다 나은 오류 핸들링을 제공한다. 다중 커널들 사이의 리소스 공유를 필요로하지 않는 애플리케이션들은 본 발명의 상기 공통의 더미 API 측면을 실시하지 않을 수 있다.

도 12는 적절하게 설정되거나 설계되었을 때, 본 발명이 실시될 수 있는 컴퓨터 시스템으로서 기능을 할 수 있는 전형적인 컴퓨터 시스템을 도시한다. 컴퓨터 시스템(1200)은 주기억장치(1206)(일반적으로 랜덤 엑세스 메모리, 또는 RAM), 주기억장치(1204)(일반적으로 읽기전용 메모리, 또는 ROM)를 포함하는 스토리지 디바이스에 결합되는 임의의 수의 프로세서(1202)(중앙처리 장치 또는 CPU라고도 함)를 포함한다. CPU(1202)는 프로그래밍가능한 디바이스(예를 들면 CPLD 및 FPGA)와 같은 마이크로컨트롤러 및 마이크로프로세서와 게이트 어레이 ASIC 또는 범용 마이크로프로세서와 같은 프로그래밍가능하지 않은 디바이스를 포함하는 다양한 유형의 것이 있을 수 있다. 종래기술에 공지된 바와 같이, 주기억장치(1204)는 단방향으로 CPU에 데이터 및 명령어를 전송하도록 동작하고, 주기억장치(1206)는 일반적으로 양방향 방식으로 데이터 및 명령어를 전송하도록 사용된다. 이러한 주기억장치 모두는 상술한 바와 같은 적절한 컴퓨터-판독가능한 매체를 포함한다. 대용량 스토리지 디바이스(1208)는 또한 양방향으로 CPU(1202)에 커플링될 수 있고, 추가적인 데이터 스토리지 용량을 제공하고, 상술한 컴퓨터 판독가능한 매체 모두를 포함할 수 있다. 대용량 스토리지 디바이스(1208)는 프로그램, 데이터 등을 저장하는 데에 사용되고, 일반적으로 하드 디스크와 같은 보조기억장치 매체이다. 적절한 경우에, 상기 대용량 스토리지 디바이스(1208) 내에 포함된 정보는 가상 메모리로써 주기억장치(1206)의 일부와 같은 표준 형태로 결합될 수 있다는 것이 이해될 것이다. 또한 CD-ROM(1214)과 같은 특정한 대용량 스토리지 디바이스는 데이터를 단방향으로 CPU에 전송할 수 있다.

CPU(1202)는 또한 비디오 모니터, 트랙볼, 마우스, 키보드, 마이크로폰, 터치-감지 디스플레이, 트랜스듀서 카드 판독기, 자기 또는 페이퍼 테이프 판독기, 태블릿, 스타일러스, 음성 또는 필기 인식기, 또는 기타의 컴퓨터와 같은 다른 공지의 입력 디바이스와 같은 하나 이상의 입/출력 디바이스에 연결하는 인터페이스(1210)에 결합될 수 있다. 마지막으로, CPU(1202)는 데이터베이스 또는 컴퓨터, 또는 원격통신, 또는 일반적으로 (1212)에서 도시된 외부 연결을 이용하는 인터넷 네트워크등의 외부 디바이스에 선택적으로 결합될 수 있다. 이러한 연결로, CPU가 네트워크로부터 정보를 수신하고, 또는 본 발명의 교안에서 기술한 방법의 스텝을 실시하면서 네트워크로 정보를 출력할 수 있다.

당업자는 본 발명의 교안에 따라, 상기 스텝 및/또는 시스템 모듈 중 임의의 것을 적절히 대체하고, 재정리, 및 제거하며, 추가적인 스텝 및/또는 시스템 모듈을 특정 애플리케이션의 필요에 따라 삽입하며, 본 실시예의 방법 및 시스템은 폭넓은 적절한 프로세스와 시스템 모듈을 이용하여 구현될 수 있고, 특정한 컴퓨터 하드웨어, 소프트웨어, RTOS, GPOS, 펌웨어, 마이크로코드 등에 한정되지 않음을 쉽게 인지할 것이다.

본 발명의 적어도 하나의 실시예를 완전히 기술하였지만, 본 발명에 따라 다중 커널 사이에서 동시에 실행하고 리소스를 공유하는 다른 동등한 또는 대안의 방법이 당업자에게는 명백할 것이다. 본 발명은 예시의 방식으로 상술되었고, 개시한 특정한 실시예는 상술한 특정한 형태로 본 발명을 한정하도록 의도되지는 않았다. 본 발명은 따라서 하기의 청구범위의 취지와 범위에 있는 모든 변형, 동등물 및 대안을 포함한다.

도면의 간단한 설명

본 발명은 첨부도면으로 예시의 방법에 의해 그러나 그에 한정되지 않으면서 설명되었고, 동일한 참조번호는 유사한 엘리먼트를 가리킨다.

도 1은 본 발명의 일 실시예에 따라 하나의 하드웨어 플랫폼 상에서 다중 커널들을 실행시킬 수 있는 예시적인 아키텍처의 도표를 도시한다.

도 2는 본 발명의 일실시예에 따라 다중 커널을 동시에 실행하기 위한 방법의 플로우 차트를 도시한다.

도 3은 본 발명의 일실시예에 따라 도 2에서 기술된 프라이머리 커널의 선택을 위한 예시적인 방법의 플로우 차트를 도시한다.

도 4는 본 발명의 일실시예에 따라 도 2에서 기술된 프라이머리 커널을 시작하기 위한 예시적인 방법의 플로우 차트를 도시한다.

도 5는 본 발명의 일실시예에 따라 도 2에서 기술된 세컨드리 커널(들)을 선택하고 추가하기 위한 예시적인 방법의 플로우 차트를 도시한다.

도 6은 본 발명의 일실시예에 따라 도 2에서 기술된 세컨드리 커널을 시작하기 위한 예시적인 방법의 플로우 차트를 도시한다.

도 7은 본 발명의 일 실시예에 따라 다중 커널을 위한 공통의 인터럽트 핸들러와 공통의 스케줄러를 위한 예시적인 아키텍처의 블록도를 도시한다.

도 8은 본 발명의 일 실시예에 따라 도 7의 문맥에서 다중 커널용의 인터럽트 마스크 레벨의 예시적인 도표를 도시한다.

도 9는 본 발명의 일실시예에 따라, 도 2의 블록도 및 도 8의 바 차트에 도시된 멀티 다중 커널용 인터럽트 마스크 레벨의 추가적인 측면을 도시한다.

도 10은 본 발명의 일 실시예에 따라 주기적인 신호에 의해 커널의 스위칭을 하는 예시적인 방법의 플로우 차트를 도시한다.

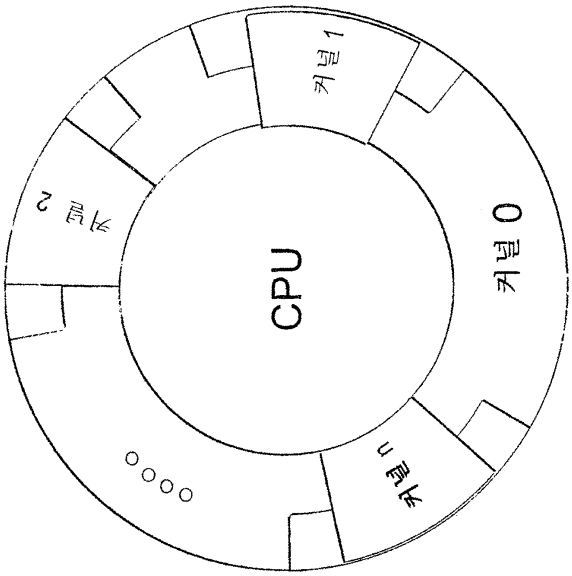
도 11은 공통의 시스템 호출이 다중 커널 리소스 공유에 이용되는 본 발명의 일실시예의 예시적인 블록도를 도시한다.

도 12는 적절하게 설정되고 설계될 때, 본 발명이 구현되는 컴퓨터 시스템으로서 기능할 수 있는 일반적인 컴퓨터 시스템을 도시한다.

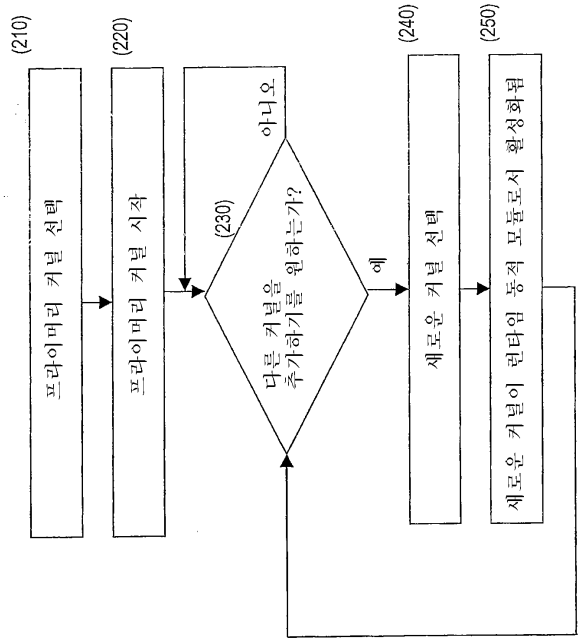
달리 표시되지 않으면, 도면의 축척은 문제되지 않는다.

도면

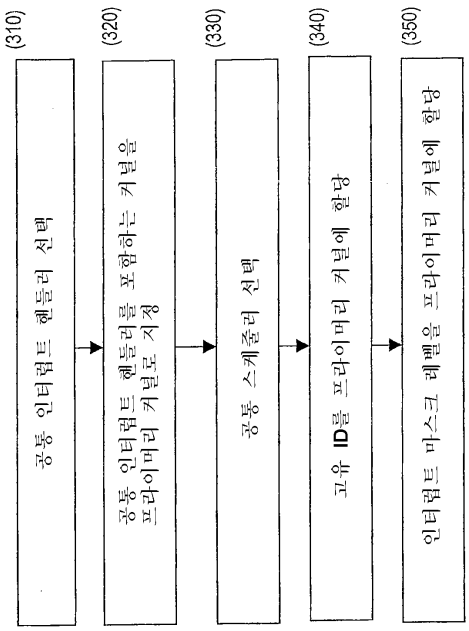
도면1


CPU
커널 0
커널 1
커널 2
커널 n
○○○○

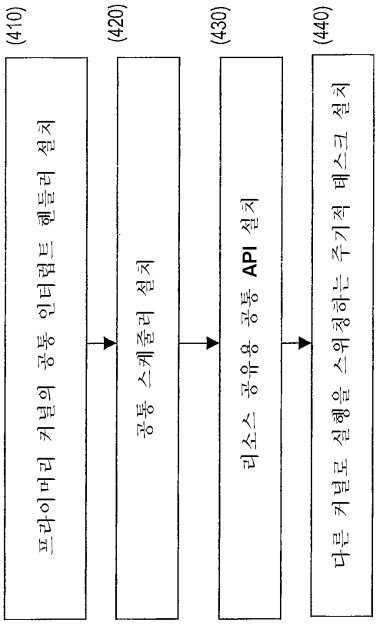
도면2


(210)
프라이머리 커널 선택
(220)
프라이머리 커널 시작
(230)
다른 커널을 추가하기를 원하는가?
아니오
예
(240)
새로운 커널 선택
(250)
새로운 커널이 런타임 동적 모듈로서 활성화됨

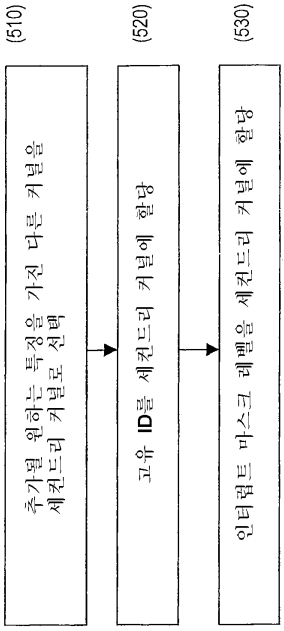
도면3


(310)
공통 인터럽트 핸들러 선택
(320)
공통 인터럽트 핸들러를 포함하는 커널을 프라이머리 커널로 지정
(330)
공통 스케줄러 선택
(340)
고유 ID를 프라이머리 커널에 할당
(350)
인터럽트 마스크 레벨을 프라이머리 커널에 할당

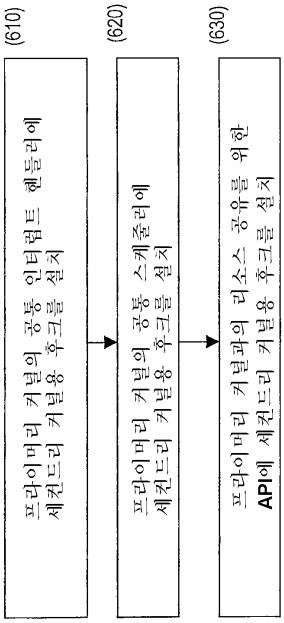
도면4


(410)
프라이머리 커널의 공통 인터럽트 핸들러 설치
(420)
공통 스케줄러 설치
(430)
리소스 공유용 공통 API 설치
(440)
다른 커널로 실행을 스위칭하는 주기적 테스크 설치

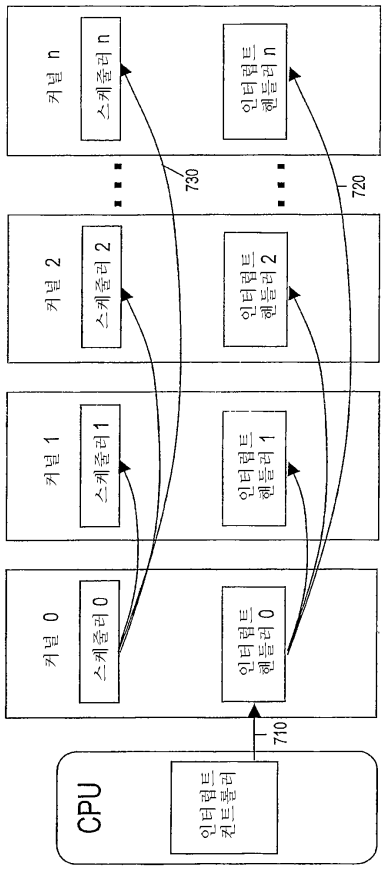
도면5


(610)
추가될 원하는 특징을 가진 다른 커널을
세컨드리 커널로 선택
(620)
고유 ID를 세컨드리 커널에 할당
(530)
인터럽트 마스크 레벨을 세컨드리 커널에 할당

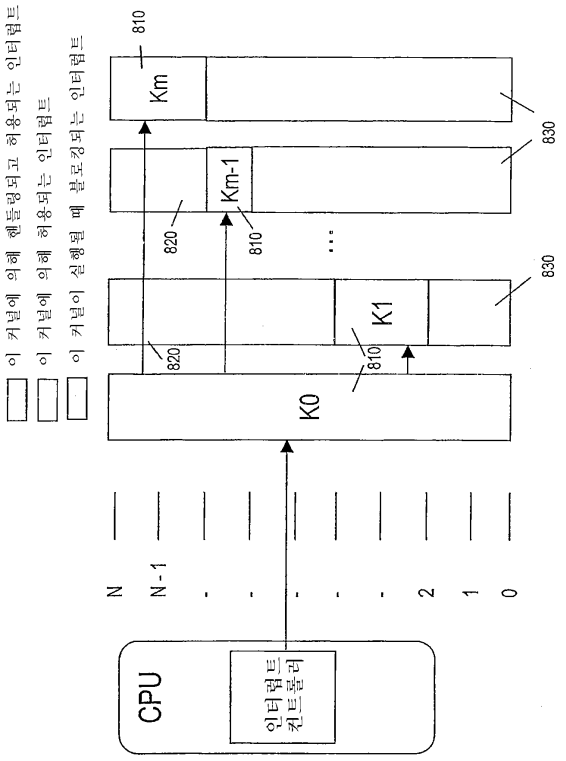
도면6


(610)
프라이머리 커널의 공통 인터럽트 핸들러에
세컨드리 커널용 후크를 설치
(620)
프라이머리 커널의 공통 스케줄러에
세컨드리 커널용 후크를 설치
(630)
프라이머리 커널과의 리소스 공유를 위한
API에 세컨드리 커널용 후크를 설치

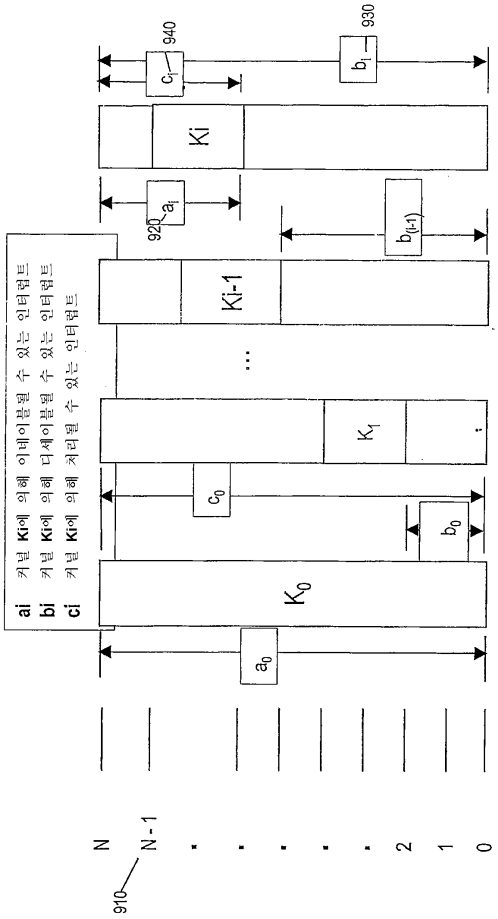
도면7


CPU
인터럽트 컨트롤러
710
커널 0
스케줄러 0
인터럽트 핸들러 0
커널 1
스케줄러 1
인터럽트 핸들러 1
커널 2
스케줄러 2
인터럽트 핸들러 2
커널 n
스케줄러 n
인터럽트 핸들러 n
730
720

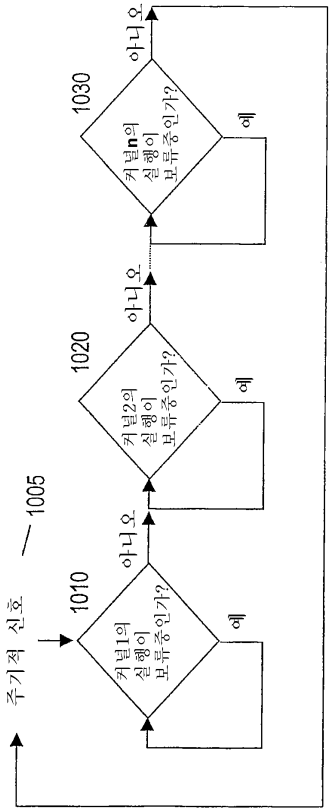
도면8


이 커널에 의해 핸들링되고 허용되는 인터럽트
이 커널에 의해 허용되는 인터럽트
이 커널이 실행될 때 블로킹되는 인터럽트
CPU
인터럽트 컨트롤러
N
N-1
2
1
0
K0
K1
Km-1
Km
810
820
830

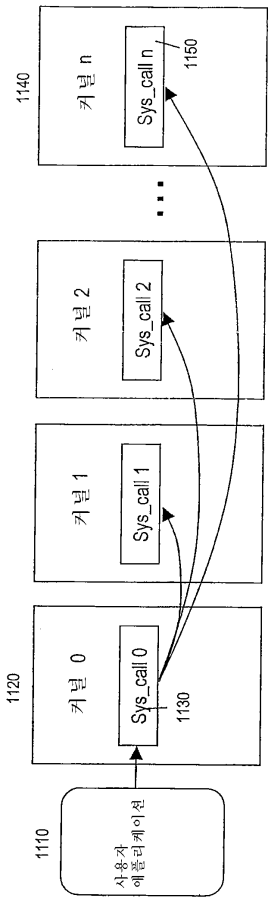
도면9


ai 커널 Ki에 의해 이네이블될 수 있는 인터럽트
bi 커널 Ki에 의해 디세이블될 수 있는 인터럽트
ci 커널 Ki에 의해 처리될 수 있는 인터럽트
910
N
N - 1
2
1
0
920
930
940

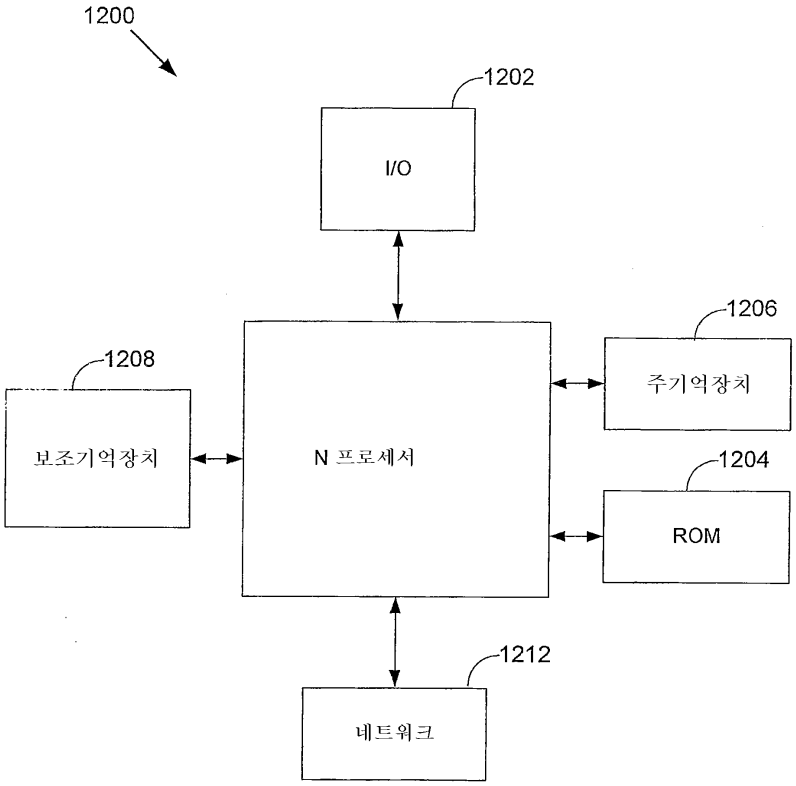
도면10


주기적 신호
1005
1010
커널1의 실행이 보류중인가?
아니오
예
1020
커널2의 실행이 보류중인가?
아니오
예
1030
커널n의 실행이 보류중인가?
아니오
예

도면11


1140
커널 n
Sys_call n
1150
커널 2
Sys_call 2
커널 1
Sys_call 1
1120
커널 0
Sys_call 0
1130
1110
사용자
에플리케이션

도면12


1200
1202
I/O
1206
주기억장치
1208
보조기억장치
N 프로세서
1204
ROM
1212
네트워크