11장. 커널 메커니즘 (Kernel Mechanism)

이 장에서는 커널의 여러 부분들이 함께 효과적으로 동작할 수 있도록 리눅스 커널이 제공 하는 몇가지 일반적인 작업과 메커니즘에 대해서 설명한다.

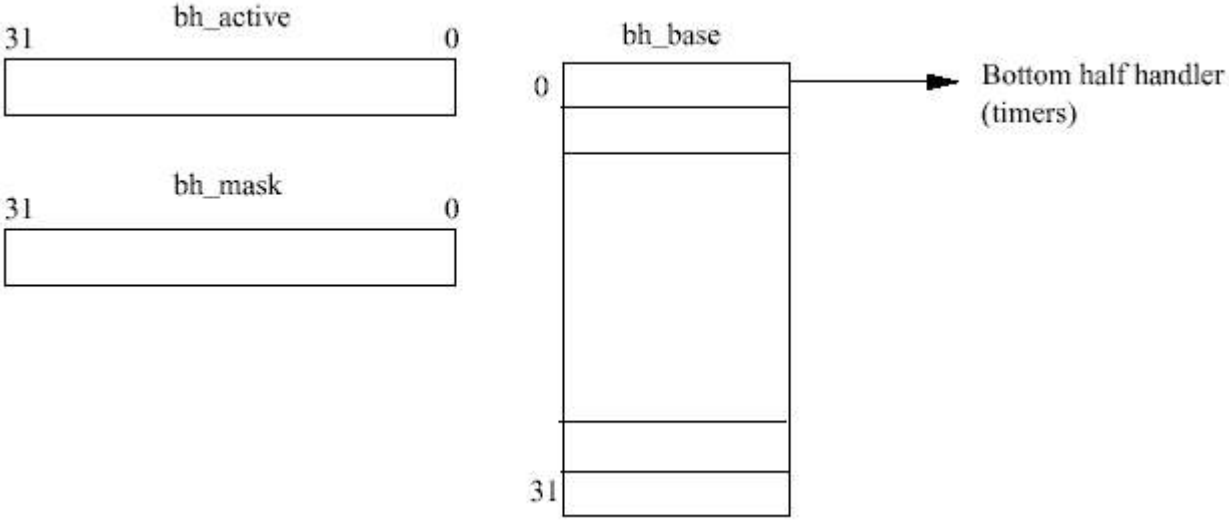

그림 11.1 : 하반부 처리 자료구조

11.1 하반부 처리(Bottom Half Handling)

커널에서는 종종 꼭 그 시점에서 일을 처리하길 바라지 않는 경우가 있다. 이의 대표적인 예로 인터럽트를 처리하는 도중이다. 인터럽트가 발생했을 때 프로세서는 자신이 하던 일을 중지하고 운영체제는 인터럽트를 해당하는 디바이스 드라이버에게 전달한다. 인터럽트를 처 리하는 동안에는 시스템의 다른 부분을 실행할 수 없으므로, 디바이스 드라이버는 인터럽트 처리에 너무 많은 시간을 보내면 안된다. 여기에는 당장이 아니라 나중에 처리해도 되는 일 들이 종종 있다. 리눅스의 하반부 핸들러(bottom half handler)[1]는 디바이스 드라이버나 리눅스 커널의 다른 부분들이, 할 일을 나중에 실행되는 큐에 넣을 수 있도록 하기 위해 개발되었 다. 그림 11.1은 하반부 처리와 관련된 커널의 자료구조를 보여준다. 모두 32개까지의 서로 다른 하반부 핸들러가 있을 수 있다[2]. bh_base는 커널의 하반부 핸들러 루틴을 가리키고 있는 포인터들의 벡터이다. bh_mask와 bh_active는 어떤 핸들러가 설치되어 있고 액티 브 한 지 나타내는 비트들의 집합이다. bh_mask의 비트 N이 설정되어 있다면 bh_base의 N번째에 하반부 루틴이 담겨 있는 것이다. bh_active의 N번째 비트가 설정되어 있으면, 스케쥴러가

가능하다고 판단할 때 N번째 하반부 핸들러 루틴을 되도록 빨리 불러주어야 한 다는 것이다. 이들 인덱스들은 정적으로 정의된 것이다[3]. 타이머 하반부 핸들러는 가장 높은 우선순위를 가지며(인덱스 0), 콘솔 하반부 핸들러는 다음 우선순위(인덱스 1)를 가진다. 일 반적으로 하반부 핸들러 루틴들은 자신과 연결된 작업들의 목록을 가지고 있다. 예를 들어, 즉시실행(immediate) 하반부 핸들러는 바로 수행해야 하는 작업들의 목록인 즉시실행 작업 큐(tq_immediate)를 가지고 동작한다.

커널의 하반부 핸들러 중에 어떤 것들은 장치에 고정되어 있지만 다른 것들은 보다 일반적 으로 쓸 수 있다[4] :

- **TIMER** 이 핸들러는 시스템의 주기적으로 발생하는 타이머 인터럽트가 발생할 때마다 액티 브로 표시가 되고, 커널의 타이머 큐 메커니즘을 위해 사용된다.
- **CONSOLE** 이 핸들러는 콘솔 메시지를 처리하는데 사용된다.
- **TQUEUE** 이 핸들러는 tty 메시지를 처리하는데 사용된다[5].
- **NET** 이 핸들러는 일반적인 네트웍 처리에 사용된다.
- **IMMEDIATE** 이는 여러 디바이스 드라이버들이 나중에 실행될 작업들을 쌓아두는데 사용 된다.

디바이스 드라이버나 커널의 어떤 부분이 나중에 수행될 작업을 스케쥴할 필요가 있을 때, 이들은 작업을 적당한 시스템 큐에 - 예를 들어 타이머 큐같은 - 넣고, 커널에 하반부 핸들 러가 수행될 필요가 있다고 신호를 보낸다. 이는 bh_active의 해당하는 비트를 설정하게 된다[6]. 만약 드라이버가 어떤 일을 즉시실행 큐에 넣고 이 즉시실행 하반부 핸들러가 실행 되어 이를 처리하길 바란다면 8번 비트를 설정할 것이다. 각 시스템 콜이 끝나서 제어권이 이를 부른 프로세스로 돌아가기 바로전에 bh_active 비트마스크를 검사하며, 만약 어떤 비트가 설정되어 있으면, 액티브로 표시된 하반부 핸들러 루틴들이 불린다. 비트 0을 먼저 검사하고, 1번을 다음에, 이런 식으로 31번 비트까지 검사한다. 각 하반부 핸들러 루틴을 부 르고 난 후에 bh_active의 해당 비트는 0으로 설정된다. bh_active는 일시적인 것이다. 이는 단시 스케쥴러 호출 사이에만 의미가 있으며, 하반부 핸들러에서 더이상 할 일이 없을 때 이들을 부르지 않게 하는 방법이다.

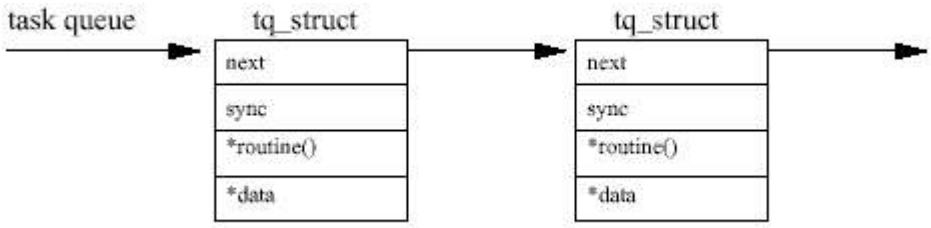

그림 11.2 : 작업큐

11.2 작업큐(Task Queue)

작업큐는 커널이 작업을 나중으로 미루는데 사용하는 방법이다. 리눅스는 작업을 큐에 쌓아 두고 이를 나중에 처리할 수 있도록 하는 일반적인 메커니즘을 가지고 있다[7]. 작업큐는 종 종 하반부 핸들러와 연결되어 쓰이기도 한다. 타이머 작업큐는 타이머 하반부 핸들러가 실 행될 때 처리된다[8]. 작업큐

는 그림 11.2에서 보는 것과 같이, 함수의 주소와 다른 데이터를 가리키는 포인터를 가진 tq_struct 자료구조의 단일 연결 리스트로 이루어진 아주 간단한 자료구조이다. 작업큐에 있는 한 원소가 처리가 될 때 데이터 포인터와 함께 여기에 지정된 함수가 불린다.

커널에 있는 어떤 것이든 (예를 들어 디바이스 드라이버같은) 작업큐를 만들고 사용할 수 있지만, 실제로 커널이 만들고 관리하는 작업큐로는 다음 세가지가 있다[9].

- **타이머(timer)** 이 큐는 다음 시스템 클럭 틱이 발생하였을 때 가능한 빨리 처리되어야 하는 일들을 큐에 넣기 위해서 사용된다. 각 클럭 틱이 발생할 때마다 여기에 무언가 있는지 검사하며, 이 큐에 무언가 있다면 타이머 큐 하반부 핸들러[10]가 액티브 상태로 바뀌게 된 다. 타이머 큐 하반부 핸들러 역시 다른 하반부 핸들러와 마찬가지로 스케쥴러가 다음에 실행될 때 처리가 된다. 이 큐는 훨씬 복잡한 구조를 가지고 있는 시스템 타이머하고 혼 동하지 말아야 한다[11].

- **즉시실행(immediate)** 이 큐 역시 스케쥴러가 액티브 하반부 핸들러를 처리할 때 같이 처리 된다. 즉시실행 하반부 핸들러는 타이머 큐 하반부 핸들러보다 우선순위가 낮으므로 이 보다는 더 늦게 실행이 된다.

- **스케쥴러(scheduler)** 이 큐는 스케쥴러에 의해 직접 처리된다. 이는 시스템에 있는 다른 작 업큐를 지원하기 위해서 사용되며, 이 경우 실행되는 작업은 디바이스 드라이버같은 것 들을 위한 작업큐를 처리하는 루틴일 것이다.

작업큐가 처리되면 큐에 있는 첫번째 원소에 대한 포인터는 큐에서 제거되어 null 포인터로 바뀐다. 사실, 이 제거하는 과정은 하나의 쪼개질 수 없는 연산으로 처리되며, 중단될 수 없 는 것이다. 이렇게 큐에 있는 각각의 원소들에 등록된 처리 루틴들이 차례로 호출이 된다. 큐에 있는 각 원소는 종종 정적으로 데이터를 할당받기도 한다. 그런데 여기에는 할당된 메 모리를 알아서 해제하는 메커니즘이 본래 포함되어 있지 않다. 작업큐를 처리하는 루틴은 단지 리스트의 다음 원소로 이동할 뿐이다. 할당받은 커널 메모리를 제대로 해제하는 것은 큐에 있던 작업이 해야 할 일이다.

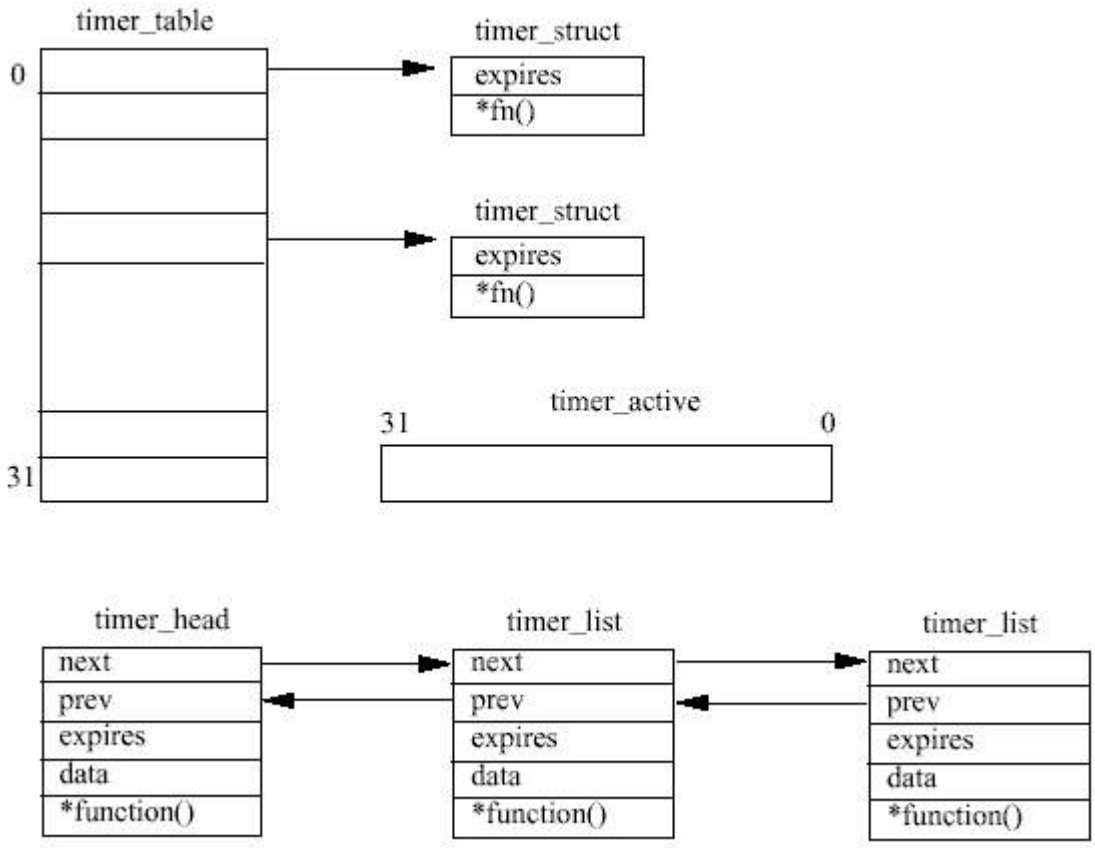

그림 11.3 : 시스템 타이머

11.3 타이머(Timer)
운영체제는 미래의 어떤 시간에 해야할 행동들을 스케쥴할 수 있는 능력을 필요로 한다. 이 들 행동들을 상대시간으로 정확하게 얼마간의 시간 후에 실행하도록 스케쥴하기 위한 메커 니즘이 필요하다. 운영체제를 지원하기를 바라는 마이크로프로세서들은, 반드시 정기적으로 프로세서에게 인터럽트를 발생하는 프로그래밍 가능한 간격 타이머(interval timer)를 가지고 있어야 한다. 이렇게 정기적으로 발생하는 인터럽트를 시스템 클럭 틱(clock tick)이라고 하며, 이는 시스템의 행동들을 결집시키는 메트로놈과 비슷한 일을 하는 것이다. 리눅스는 현재 시간을 아주 단순하게 표현한다. 리눅스는 시간을 시스템이 부팅한 때부터 발생한 클럭 틱 의 횟수 단위로 표현한다. 모든 시스템 시간은 이 단위로 되어 있으며, 이는 jiffies라고 하며, 이와 똑같은 이름의 전역 변수가 존재한다[12].

리눅스는 두가지 형태의 시스템 타이머를 가지고 있으며, 이 두 큐의 루틴들은 똑같은 시스 템 타임에 호출되지만[13], 구현방식에 있어서 약간의 차이가 있다. 그림 11.3은 이 두가지 메 커니즘을 보여준다. 앞의 것은 예전의 타이머 방식으로서, 정적변수로 timer_struct 자료 구조에 대한 포인터 32개를 배열로 가지고 있으며, 액티브 타이머의 마스크인 timer_active를 가지고 있다. 타이머가 타이머 테이블에 들어가는 것은 정적으로 정의된 다 (하반부 핸들러 테이블인 bh_base에 더 가깝다[14]). 각 항목들은 시스템 초기화 때 대부 분 이 테이블에 추가된다. 두번째 방식은 더 새로운 것으로서 timer_list 자료구조를 만 료시간의 올림순으로 가지고 있는 연결 리스트를 사용한다.

이 두가지 방식 모두 만료시간을 jiffies 단위로 가지고 있는 시간을 이용하므로, 5초 후 에 실행되길 바라는 타이머라면, 5초를 jiffies 단위로 변환한 후 현재 시스템 시간에 더 하여 만료시간을 시스템 시

간의 jiffies로 나타내야 한다[15]. 모든 시스템 클럭 틱마다 타 이머 하반부 핸들러는 액티브로 표시되고, 다음에 스케쥴러가 실행될 때 타이머 큐가 처리 될 수 있도록 한다. 타이머 하반부 핸들러는 이 두 가지 방식의 시스템 타이머를 모두 처리 한다. 예전 방식의 시스템 타이머에 대해서는 timer_active 비트마스크를 검사하여 설정 이 되어 있는 비트를 검사하게 된다. 만약 현재 액티브한 타이머의 만료시간이 지나면 (만 료시간이 현재 시스템의 jiffies보다 작으면), 타이머 루틴이 호출되고, 액티브 비트는 지 워지게 된다. 새로운 방식의 타이머에서는, timer_list 자료구조의 연결 리스트에 있는 각 원소를 검사하여, 만료된 모든 타이머들은 리스트에서 제거되고, 등록된 함수가 호출된다. 새로운 타이머 방식은 타이머 루틴에 인자를 넘길 수 있다는 장점이 있다.

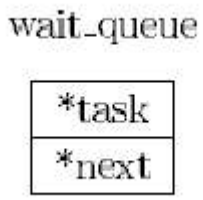

그림 11.4 : 대기큐

11.4 대기큐(Wait Queue)

프로세스가 시스템 자원을 기다려야 하는 경우가 많이 있다. 예를 들어, 프로세스가 파일 시 스템에 있는 한 디렉토리를 나타내는 VFS inode를 필요로 하는데 이 inode가 버퍼 캐쉬에 있지 않은 경우, 프로세스는 파일 시스템을 가지고 있는 물리적인 장치에서 그 inode를 가 져오는 것을 기다려야 한다.

리눅스 커널은 대기큐(그림 11.4을 보라)라는, 프로세스의 task_struct에 대한 포인터와 대기큐에 있는 다음 원소에 대한 포인터를 가지고 있는, 아주 단순한 자료구조를 사용한다.

프로세스가 대기큐의 끝에 추가가 되면, 이들은 인터럽트 가능(interruptible), 또는 인터럽트 불가능(uninterruptible) 상태가 된다. 인터럽트 가능한 프로세스는 대기큐에 있는 동안 발생하 는 타이머 만료나 시그널같은 이벤트들에 의해서 인터럽트가 될 수 있다. 대기중인 프로세 스의 상태는 이를 반영하여 INTERRUPTIBLE 또는 UNINTERRUPTIBLE 둘 중의 하나가 될 것이다. 이 프로세스는 지금 당장 계속 실행할 수 없기 때문에 스케쥴러가 실행되어, 새로 실행할 프로세스를 선택하게 되면 대기 프로세스는 중단이 된다[16].

대기큐가 처리가 될 때[17] 대기큐에 있는 모든 프로세스들의 상태는 RUNNING으로 바뀌게 된다. 만약 그 프로세스가 실행큐에서 제거된 것이었다면 다시 실행큐에 넣게 된다. 다음에 스케쥴러가 실행될 때 대기큐에 있던 프로세스들은, 더 이상 기다리고 있는 것이 아니기 때 문에 실행될 수 있는 후보가 된다. 대기큐에 있는 프로세스가 스케쥴이 되면 제일 먼저 하 는 일은 자신을 대기큐에서 제거하는 것이다. 대기큐는 시스템 자원에 대한 접근을 동기화 하는데 사용할 수 있고, 리눅스가 세마포어를 구현하는데에도 사용한다. (아래를 보라)

11.5 버저락(Buzz Lock)

이것은 스핀락(spin lock)이라고 더 잘 알려져 있는데, 자료구조나 코드의 한 부분을 보호하 는 가장 기본적인 방법이다. 이것은 코드의 임계지역 안에서 동시에 하나의 프로세스만 있 도록 허용한다. 리눅스에서는 하나의 정수 항목을 락으로 사용하여 자료구조에 있는 항목에 대한 접근을 제한하는 목적으로 사용한다[18]. 임계지역으로 들어가고자 하는 각 프로세스들은 락의 초기값을 0에서 1로 바꾸려고 한다. 만약 현재 값이 1이라면 프로세스는, 코드의 루 프 안에서 계속 빙글빙글 돌면서 다시 시도하게 된다. 락을 가지고 있는 메모리 위치에 대 한 접근은 반드시 한번에 이루어져야 한다(atmoic). 값을 읽고 그 값이 0인지 확인하고, 0이 면 값을 1로 바꾸는 것은 다른 어떤 프로세스에 의해서 중단되어선 안된다. 대부분의 CPU 구조들은 이를 특별한 명령어로 지원하지만, 캐시되지 않은 메인 메모리를 이용하여 버저락 을 구현할 수도 있다.

락을 소유하고 있던 프로세스가 코드의 임계지역을 벗어날 때 버저락의 값을 감소시켜 0이 되게 한다. 락을 검사하며 계속 돌고 있던 어떤 프로세스든지 이 값이 0인 것을 알 수 있겠 지만, 처음 읽은 프로세스가 이를 1로 증가하고 임계지역으로 들어가게 될 것이다.

11.6 세마포어(Semaphore)[19]

세마포어는 코드나 자료구조의 임계구역을 보호하는데 사용된다. 디렉토리를 나타내는 VFS inode 같은 임계 자료에 접근하는 것은, 프로세스의 다른 한 편에서 돌아가는 커널 코드에 의해서 이루어진다. 한 프로세스가 사용하고 있는 이런 중요한 자료구조를 다른 프로세스에 서 고칠 수 있게 하는 것은 매우 위험하다. 이런 목적을 달성할 수 있는 한 방법은 임계자 료에 접근하는 곳 주위에 버저락을 사용하는 것이지만, 이는 그다지 시스템 효율성이 좋지 않은 단순한 접근 방법이다. 대신 리눅스는 동시에 한 프로세스만이 코드나 데이터의 임계 구역에 접근할 수 있도록 세마포어를 사용한다. 이 구역에 접근하려는 다른 모든 프로세스 는 이 세마포어가 해제될 때까지 기다리게 될 것이다. 대기하게 되는 프로세스는 중단되지 만, 시스템의 다른 프로세드들은 정상적으로 계속 동작할 수 있다.

리눅스 semaphore 자료구조는 다음과 같은 정보를 가지고 있다[20].

- **카운트(count)** 이 항목은 이 자원을 사용하려고 하는 프로세스들의 갯수를 관리한다. 양수는 이 자원이 사용가능하다는 것을 의미한다. 음수 또는 0은 프로세스들이 그것이 해제되기를 기다리고 있다는 것을 의미한다. 초기값으로 1을 주는 것은 단지 동시에 한 프로세스만이 이 자원을 사용할 수 있다는 것을 말한다. 프로세스가 자원을 얻고자 하면 카운트를 1 감소 시키고, 자원의 사용이 끝나면 카운트를 1 증가시킨다.
- **깨울(waking)** 이 자원을 기다리고 있는 프로세스의 수이며, 이 자원이 해제될 때 깨어나게 될 프로세스의 수이기도 하다.
- **대기큐(wait queue)** 프로세스가 어떤 자원을 기다리면 그 자원의 대기큐에 들어간다.
- **락(lock)** waking 항목을 접근할 때 사용하는 버저락이다.

세마포어의 초기 카운트가 1이라고 할 때, 처음 사용하는 프로세스는 그 값이 양수라는 것 을 알고, 1을 감소시켜 0으로 만든다. 이 프로세스는 이제 세마포어에 의해 보호되는, 코드 나 자원의 임계부분을 "소유"하게 된다. 프로세스가 임계지역을 벗어나게 되면 세마포어의 카운트를 증가시킨다. 가장 최선인 경우는 임계지역을 소유하고자 하는 다른 프로세스가 없 는 경우이다. 리눅스의 세마포어는 이 경우(가장 흔한 경우이기도 하다)에 대해 효율적으로 동작하도록 구현되었다[21].

만약 다른 프로세스가 소유하고 있는 임계지역에 한 프로세스가 들어가려고 할 때, 이 프로 세스도 역시 카운트를 1 감소시킨다. 이번엔 카운트가 음수(-1)이므로 프로세스는 임계지역 에 들어가지 못한다. 대신 영역을 소유하고 있는 프로세스가 영역을 빠져나갈 때까지 기다 려야 한다. 리눅스에서는 기다리는 프로세스를 재우고, 임계지역을 소유하고 있는 프로세스 가 임계지역을 빠져나갈 때 이를 깨우도록 한다. 기다리는 프로세스는 자신을 세마포어에 있는 대기큐에 추가하고, 루프를 돌면서 waking 항목의 값을 검사하고, waking이 0이 아 닌 값이 될 때까지 스케쥴러를 호출하는 일을 반복한다[22].

임계지역의 소유자는 세마포어의 카운트를 증가시키는데, 그 값이 0보다 작거나 같으면 잠 들어서 이 자원을 기다리는 프로세스가 있다는 것이다. 가장 최선의 경우는 세마포어의 카 운트가 다시 초기값인 1이 되어서, 더이상 필요한 일이 없는 것이다. 소유하는 프로세스는 waking 카운터를 증가시키고, 세마포어의 대기큐에서 잠들어 있는 프로세스를 깨운다. 기 다리는 프로세스가 깨어났을 때 waking 카운터는 이제 1이 되어 있을 것이고, 이 프로세 스는 이제 임계지역에 들어갈 수 있게 된다. 이 프로세스는 waking 카운터를 0으로 감소 시키고, 자신의 작업을 계속하게 된다. 세마포어의 waking 항목에 대한 접근은 세마포어의 락 항목을 이용한 버저락에 의해 보호된다.

번역 : 이호, 심마로
정리 : 이호

역주 1) 하반부(bottom half)라는 말은 인터럽트 핸들러를 상반부(top half)라고 생각하여 인터 럽트 핸들러에서 처리되지 않고 나중으로 미뤄진 작업을 대비시켜 붙인 이름이다. (flyduck)

역주 2) 이는 하반부 처리에 관련된 자료구조가 4바이트 크기의 마스크와 고정된 크기의 배 열로 되어 있기 때문이다. 따라서 하반부 처리를 사용할 수 있는 것은 한정되어 있으며, 이것보다 좀 더 개선된 구조의 작업큐가 나오게 된다. (flyduck)

역주 3) 정적(static)으로 정의되었다는 의미는, 하반부 핸들러를 사용하겠다고 동적으로 인덱 스를 얻어서 사용하는 것이 아니라, 미리 각 인덱스에는 무엇이 담길 것이며 이 인덱스 를 정의하는 상수(아래에 나오는)가 정의되어 있다는 것이다. (flyduck)

역주 4) 아래에 나오는 하반부 핸들러는 각각 TIMER_BH, CONSOLE_BH, TQUEUE_BH, NET_BH, IMMEDIATE_BH로 정의되어 있다. 이들 외에도 다른 하반부 핸들러도 있으며, include/linux/interrupt.h에서 확인할 수 있다. (flyduck)

역주 5) 이는 원문의 내용이 틀린 것이라고 생각하지만, TQUEUE는 각 타이머 틱마다 활성 화되는 하반부 핸들러로, tq_timer 작업큐를 처리하는 역할을 한다. 앞의 TIMER 하반 부 핸들러 역시 각 타이머 틱마다 활성화되지만 11.3에 나오는 커널 타이머를 처리하는 역할을 한다. 하지만 둘 다 타이머 틱이 발생했을 때 활성화된다는 점은 동일하지만 맡 은 역할은 다르다. (kernel/sched.c의 do_timer() 참조) (flyduck)

역주 6) 이는 mark_bh() 함수를 해당하는 하반부 핸들러 상수와 함께 부르면 된다. (flyduck)

역주 7) 이런 용도로 앞에 하반부 핸들러를 설명했는데, 둘의 역할은 비슷하지만 메커니즘 과 사용하는 경우는 서로 다르다. 하반부 핸들러는 한정된 자원인 반면에, 작업큐는 작업 의 목록을 연결 리스트로 가지고 있으며, 별도의 작업큐를 정의하여 사용할 수 있기 때 문에 확장이 가능하다. 작업큐는 타이머같이 타이머 인터럽트가 발생했을 때 처리될 작 업 목록을 쌓아두기 위해서 사용되기도 하고,

디바이스 드라이버에서 작업을 미루기 위 해서 사용한다. 모듈로 만들어진 디바이스 드라이버는 하반부 핸들러를 사용할 수 없으 며, 작업큐 메커니즘을 사용해야 한다. (flyduck)

역주 8) 작업큐를 처리하는 함수는 run_task_queue()이며, kernel/sched.c에서 보면 schedule() 함수에서 run_task_queue(&tq_scheduler)를 부르며, TQUEUE 하반부 핸들러에서 tq_timer를, IMMEDIATE 하반부 핸들러에서 tq_immediate를 처리하는 것 을 볼 수 있다. (flyduck)

역주 9) 아래 나오는 세가지 작업큐 외에 tq_disk가 있지만 이는 메모리 관리 서브시스템 에서 내부적으로 사용하는 것이며, 다른 부분에서 사용할 수 없는 것이다. 이 세 작업큐 는 각각 tq_timer, tq_immediate, tq_schedule로 정의되어 있다. (flyduck)

역주 10) 이 타이머 큐 하반부 핸들러는 앞에서 이야기한 바와 같이 TQUEUE_BH이다. (flyduck)

역주 11) 시스템 타이머는 TIMER_BH에서 처리하는 11.3장에서 설명할 타이머를 말한다. (flyduck)

역주 12) 이 jiffies 단위의 시간이 정확히 어느정도의 시간인지는 시스템마다 다르다. arch/*/param.h에 HZ라는 상수가 정의되어 있는데, 클럭 틱은 초당 이 HZ 횟수만큼 발생 하므로 1 jiffie = 1 / HZ 초라고 할 수 있다. 현재 커널에서 HZ는 알파 시스템에서는 1024로 다른 시스템에서는 100으로 정의되어 있다. 이 값을 바꾸어서 컴파일 할 수 있는 데, 이 값이 커지면 시스템의 속도는 느려지겠지만 반응 속도는 더 빠를 것이며, 값이 작 아지면 속도는 빨라지지만 반응 속도는 더 느려지게 된다. (flyduck)

역주 13) 예전의 타이머는 run_old_timer()에서, 새로운 타이머는 run_timer_list() 에서 처리하며, 둘 다 timer_bh()에서 불린다. kernel/sched.c 참조 (flyduck)

역주 14) 구현방식으로 본다면 예전의 타이머는 정적으로 정의되고 부팅시에 핸들러가 등록 되는 하반부 핸들러와, 새로운 타이머는 동적으로 사용하는 작업큐와 비슷하다고 할 수 있다. (flyduck)

역주 15) 즉 jiffies(현재 시간을 나타내는 전역변수) + 원하는 간격 * HZ로 계산한다. (flyduck)

16) REVIEW NOTE : 다음번에 스케쥴러가 실행될 때 INTERRUPTIBLE 상태에 있는 태스크 가 실행되는 것을 막는 것은 무엇인가? 대기큐의 프로세스는 깨어날 때까지 절대로 실행 되지 않는다.

역주 17) 대기큐가 처리가 될 때라는 것은, 기다리고 있던 자원을 사용할 수 있게 되어 이 자원을 기다리는 대기큐를 처리할 때라는 것이다. (flyduck)

역주 18) 다음 세마포어에서 이 스핀락을 semaphore 자료구조의 waking 항목에 대한 접 근을 제어할 때 사용하는 것을 볼 수 있다. (flyduck)

역주 19) 이 세마포어는 IPC에서 나온 세마포어와 다르다. 이 세마포어는 SMP에서 한 프로 세서만이 커널 모드로 들어갈 수 있도록 사용하는 것이다. 리눅스에서 SMP는 현재 효율 적으로 만들어지지 않았다. 리눅스 커널은, 커널 모드에서 자신이 제어권을 놓지 않는 한 다른 프로세스에 의해 중단되지 않으며, 인터럽트 처리 루틴도 자신보다 높은 우선순위 를 가진 인터럽트가 아닌 다른 프로세스에 의해 중단되지 않는다는 가정을 가지고 있다. 즉 커널모드에서 자료구조를 수정하는 것이 다른 것에 의해 중단되지 않는다는 가정을 가지고 있는 것이다. 이는 SMP에서 문제가 되는데, 왜냐하면 한 프로세서에서 커널 모드 로 들어가 자료구조를 수정하고 있을 때, 다른 프로세서에서 커널모드로 들어가면 커널 이 유지하는 자료구조를 동시에 여러 프로세서가 수정하게 되기 때문이다. 이의 가장 올 바른 해결책은 당연히 자료구조를 수정하기 전에 임계지역을 표시하고 다른 프로세서가 접근하지 못하게 하는 것이지만, 이는 현재 구조상 너무 방대한 작업을 필요로 한다. 그 래서 현재 SMP 구현은 하나의 세마포어를 사용하여 동시에 한 프로세서만이 커널모드에 있을 수 있게 하며, 이 장에서 설명하는 세마

포어는 이런 용도를 위해 사용하는 것이다. 그래서 SMP에서도 커널 모드에서 동작하는 프로세스가 다른 프로세스에 의해 중단되지 않게하는 것이다. 이는 커널 모드에서 잡아먹는 CPU 시간이 전체 시스템 효율성의 병목 으로 작동하게 되며, 커널 모드에 많이 진입하는 I/O 중심의 시스템에서는 더욱 병목현상 이 더 심해지게 된다. 앞으로 효율적인 SMP 시스템을 구현하려면 필요한 경우에만 락을 걸 수 있도록 수정되어야 할 것이다. (flyduck)

역주 20) 이 책의 바탕인 2.0.33 소스에는 lock 항목이 있지만, 2.0.2x 버전이나 2.2.x 버전에 서 lock 항목을 찾을 수 없다. (flyduck)

역주 21) 즉 한 프로세서만이 커널 모드에 진입할 때 가장 효율적으로 동작하도록 설계되었 다는 뜻이다. (flyduck)

역주 22) 커널 모드에 진입하기 위하여 세마포어를 얻으려고 했는데 이를 얻을 수 없다면, 자신은 세마포어를 사용할 수 있게 될 때까지 스케쥴러를 호출하며(이것은 다른 프로세 스가 자신 대신에 실행될 수 있게 만든다), 세마포어를 얻을 수 있을 때까지 기다린다는 것이다. (flyduck)