Project #6: 스케줄러 구현, 동기화문제해결, 멀티스레딩, 추첨스케줄러 추가구현

신정은, 조재호, 조한주
School of Software, Soongsil University

1. 구현 목표

1.1 라운드 로빈 스케줄러 추가

멀티태스킹 처리 도구들, TCB, 컨텍스트 그리고 스택과 태스크 전환 함수를 만들었다. 이제 범용 리스트와 컨텍스트 전환 함수를 사용하여 라운드 로빈 스케줄러를 구현한다. 라운드 로빈 스케줄러는 모든 태스크를 순차적으로 실행하여, 자료구조가 단순하고 알고리즘이 간단하다. 그래서 가벼운 OS를 만드는데 많이 사용한다. 그리고 태스크 풀과 스택 풀을 할당하여 태스크를 더 편리하게 생성하는 방법을 알아본다. MINT64 OS의 최종 목표는 태스크의 우선순위를 3단계로 나누어 각 레벨 별로 라운드 로빈을 수행하는 멀티레벨 큐 스케줄러를 구현하는 것이다.

1.2 멀티레벨 큐 스케줄러로 업그레이드 및 태스크 종료 기능 추가

라운드 로빈 스케줄러는 태스크 수가 많고 프로세서를 스스로 반환하지 않으면 콘솔셸이 느려지게 된다. 태스크가 많아도 작업과 직접적인 관련이 있는 태스크는 영향을 받지 않도록 태스크에 우선순위를 부여하고 그에 따라 태스크 실행 빈도를 조절함으로써, 태스크 수에 관계없이 콘솔셸의 수행 속도를 보장한다. 그리고 대기 큐와 유휴 태스크 구현을 통해 태스크 종료기능과 프로세서 사용률에 따라 대기상태로 전환하는 기능을 추가하여 MINT64 OS의 전체적인 태스크 구조를 완성해 본다.

1.3 태스크와 인터럽트, 태스크와 태스크 사이의 동기화 문제 해결

준비 리스트나 대기 리스트를 태스크 코드와 인터럽트 코드에서 동시에 사용하면 태스크를 생성하거나 종료하는 과정에서 리스트에 작업을 수행하다가 PIT인터럽트에 의해 태스크가 스케줄링될 경우 문제가 발생한다. 따라서 이와 같이 스케줄러 문제와 태스크 사이의 자원 공유 문제 해결을 위한 동기화에 대해 알아보고 시스템 전역에서 사용하는 데이터용 동기화 코드와 태스크들만이 공유하는 데이터용 동기화 코드를 각각 구현하여 데이터별로 동기화를 처리해 본다.

1.4 멀티스레딩 기능 추가

기존의 코드는 태스크만 지원을 하지만 태스크를 수정하여 프로세스나 스레드로 바꾸어 멀티스레딩을 지원하도록 한다. 멀티태스킹과 멀티스레딩의 개념을 알아보고 멀티스레딩 기법을 적용할 때의 주의점인 프로세스 종료 처리에 대한 문제와 동기화 문제에 대해서 공부한다. 개념을 바탕으로 기존에 작성한 태스크 코드를 멀티스레드를 지원하도록 바꾼다.

1.5 추첨 스케줄러 추가 구현

기존의 라운드 로빈 스케줄러를 추첨 스케줄러로 대체하고 그 결과를 확인한다. 우선순위를 고려하여 추첨 스케줄러의 추첨 방식을 충분히 고민하고 구현해 본다. 그리고 구현한 추첨 스케줄러가 starvation문제를 방지하는지 확인하기 위해 각 프로세스가 가진 추첨권에 비례하여 CPU 점유 시간이 증가하는지 그래프를 통해 표현한다.

2. 구현 내용

2.1 라운드 로빈 스케줄러 추가

2.1.1 스케줄러를 위한 리스트 구현

스케줄러가 태스크를 선택하려면 태스크를 모아 놓을 자료구조가 필요하다. 라운드 로빈 방식은 태스크에 일정 시간 할당하고 시간이 만료되면 다른 태스크로 전환하는 방식으로 동작하므로 큐 자료구조로 구현한다. 하지만 스케줄러를 구현하려면 데이터를 검색하거나 임의의 위치에 데이터를 제거하는 기능이 필요하다. 이를 지원하는 리스트 자료구조를 구현한다. MINT 64 OS에서는 리스트가 많이 쓰이기 때문에 범용으로 설계하여 다양한 환경에 쓰일 수 있게 한다.

리스트 구현의 핵심은 다음 데이터의 어드레스를 관리하는 것이다. 리스트의 삽입, 삭제, 검색 기능 모두 다음 데이터의 어드레스를 사용하여 처리한다. 명시적으로 다음 데이터의 어드레스 필드 위치를 데이터의 시작 위치로 고정하게 되면 타입 변환 만으로 간단히 처리할 수 있어서 편리하다. 다만 자료구조를 생성할 때 반드시 다음 데이터의 어드레스 필드를 가장 앞쪽에 정의해야 한다.

이번에 만드는 리스트는 리스트의 앞과 뒤 모두에 데이터를 추가, 제거할 수 있도록 하고, head 뿐만 아니라 tail을 관리하는 포인터를 두어 사용 편의성을 높인다.

앞에서 만든 리스트 자료구조는 전체를 0으로 설정하는 방법으로 간단히 초기화한다. 리스트에서 데이터를 추가하는 경우 데이터가 없을 때는 head와 tail이 새로운 데이터를 가리키게 한다. 데이터가 있었을 때는 tail이 새로운 자료를 가리키고 원래 tail의 자료 뒤에 연결한다. 데이터를 제거할 때는 네 가지 경우가 있다. 데이터가 하나 들어 있을 때는 head와 tail을 NULL로 만든다. 가운데 있는 데이터의 경우 앞의 데이터의 다음을 가리키는 포인터가 지울 데이터 다음의 데이터를 가리키게 한다. 데이터가 2개 이상 있는 상태에서 head를 지울 때는 head가 두 번째 자료를 가리키게 한다. tail 을 지울 때는 tail이 마지막에서 두 번째 데이터를 가리키게 한다. 임의의 데이터를 지울 때는 id를 비교하여 제거하는 방식으로 구현한다.

2.1.2 태스크 풀과 스택 풀 설계

태스크 풀은 태스크 자료구조를 모아 놓은 공간, 즉 TCB가 모여 있는 특정 메모리 영역이다. TCB 한 개가 192 바이트 이상 사용하고 최대 1,024개의 태스크를 지원할 때 적어도 192KB 이상의 메모리가 필요하다. 또한 태스크 마다 별도의 스택이 필요하므로 9KB를 스택으로 할당하면 최대 8MB가 추가로 필요하다. MINT64 OS의 커널 공간은 스택을 제외한 영역이 최대 4MB라서 공간이 부족하다. 따라서 IST 영역 8MB 이후 영역에 할당해서 태스크와 스택을 위해 사용한다. TCB를 관리하려면 TCB 풀의 시작 어드레스와 TCB의 최대 개수, 사용한 TCB의 개수 정보가 필요하다. 태스크 풀 초기화 함수, TCB 할당 함수, 해제 함수를 구현한다. 태스크 풀 초기화는 TCB의 최대 개수를 1024개로 설정하고 현재 할당된 태스크의 수를 0으로 설정한다. 그리고 각 TCB에 ID를 할당한다. TCB 할당은 iAllocateCount 필드를 검사해서 태스크 ID의 상위 32비트를 OR하여 유일한 ID를 만들어 할당한다. 해제는 사용한 TCB와 ID를 초기화하면 된다.

스택은 TCB를 할당 받을 때 스택 풀에서 TCB ID의 하위 32비트에 맞는 스택을 할당한다. 그 외 별다른 처리는 하지 않는다.

2.1.3 라운드 로빈 스케줄러 설계 및 구현

스케줄러는 태스크의 실행 순서를 관리한다. 대기 중인 태스크의 목록과 현재 수행 중인 태스크의 정보만 있으면 된다. 라운드 로빈 스케줄러는 모든 태스크가 완료될 때 까지 태스크를 순환하며 실행한다. 순차적으로 실행하는 특성 때문에 큐와 리스트를 사용한다. 앞서 구현한 범용 리스트를 사용한다.

스케줄러 자료구조에는 현재 수행 중인 태스크, 태스크가 사용할 수 있는 프로세서 시간, 실행할 태스크가 준비 중인 리스트가 담겨있다. 프로세서 시간은 스케줄러가 태스크에 허락하는 프로세서 사용 시간이다.

스케줄러는 초기화 함수, 현재 수행 중인 태스크와 태스크 리스트를 관리하는 함수, 태스크를 전환하는 함수를 구현한다. 스케줄러 초기화 함수는 부팅 과정에서 호출되며 태스크 풀과 준비 리스트를 초기화한다. 그리고 TCB를 할당 받아 현재 부팅 과정을 실행하는 태스크를 스케줄러의 실행 중인 태스크 필드에 설정한다.

태스크를 전환하는 함수는 태스크 실행 중에 전환하는 코드와 인터럽트가 발생했을 때 전환하는 코드로 구분된다. 태스크 수행 중에 전환하는 방법은 앞서 멀티태스킹을 구현할 때 전환하는 방법에 일부 코드를 추가한다. 태스크를 전환하는 도중에 인터럽트가 발생하지 못하도록해야 한다. 태스크 전환 중에 인터럽트가 발생되면 인터럽트 핸들러에서 다시 태스크가 전환된다. 이 때 스케줄러 자료구조의 RunningTask 값이 달라서 문제가 생길 수 있다. 이를 막기 위해 태스크 전환을 완료할 때 까지 인터럽트를 비활성화 한다. 따라서 태스크 전환 함수에서는 전환 시작 전에 프로세서 시간을 활성화하고 태스크 전환 직전에 인터럽트를 비활성화 한다. 그리고 나서 복원할 때 인터럽트 발생 여부도 다시 되돌린다.

인터럽트가 발생했을 때는 IST에 이미 컨텍스트가 저장되어 있어 IST에서 TCB로 바로 복사한다. 전체적으로 태스크 실행 중에 태스크를 전환하는 코드와 같고 컨텍스트의 처리만 IST를 사용하도록 한다.

인터럽트 방식을 사용하여 시분할 멀티태스킹을 구현한다. 일정한 주기로 PIT 컨트롤러가 IRQ 0 인터럽트를 발생시킨다. 기존의 공용 인터럽트 핸들러 코드에 인터럽트용 태스크 전환 함수를 추가하여 타이머 인터럽트 핸들러를 구현한다. 태스크를 전환하는 함수 외에 타이머 인터럽트가 발생한 횟수도 기록한다. 할당된 프로세서 시간이 다 되었을 때 스케줄러 인터럽트를 발생시켜 태스크 전환을 수행한다. 이렇게 태스크에게 일정한 수행 시간을 보장한다. 이제 타이머 ISR 함수에 타이머 핸들러를 추가하면 시분할 멀티태스킹을 구현할 수 있다.

앞서 구현한 함수들을 조합해 태스크를 생성하는 함수를 추가한다. CreateTask() 에서는 태스크의 엔트리 포인트와 플래그를 전달받아 태스크를 생성한다. 태스크를 생성함과 동시에 준비 리스트에 삽입하여 스케줄링될 수 있게 한다.

2.1.4 빌드 및 구동 확인

리스트 파일을 추가한다. 리스트 파일에는 라운드 로빈 스케줄러에서 사용하는 범용 리스트의 함수와 자료구조 매크로를 정의한다. 태스크 파일을 수정한다. 라운드 로빈 스케줄러, 태스크 풀, 스택 풀에 대한 함수, 자료구조와 매크로를 정의한다. 인터럽트 관련 파일과 유틸리티 파일을 수정한다. 시분할 멀티태스킹 기능에 관련된 함수를 추가한다. 유틸리티 파일에는 타이머 인터럽트 핸들러에서 증가시키는 카운터와 관련된 함수와 변수를 추가한다. C 언어 커널 엔트리 포인트 파일에는 스케줄러를 초기화하고, PIT 컨트롤러의 인터럽트 주기를 1초에 1000번으로 설정한다. 콘솔 셸 파일에는 라운드 로빈 스케줄러와 시분할 멀티태스킹 기능 테스트를 위해 두 개의 태스크를 추가한다. 멀티태스키의 효과를 시각적으로 확인하기 위해 콘솔 화면의 주변을 돌면서 문자를 출력하는 태스크와 자신의 ID에 해당하는 위치에 바람개비를 출력하는 태스크를 작성한다. 테스트용 태스크 외에도 createtask 커맨드를 수정하여 생성할 태스크의 번호와 개수를 입력한다. 최대 1024개 까지 생성 가능하도록 한다.

코드를 작성하고 빌드하고 확인한다. createtask 2 800으로 800개의 태스크를 실행해보면 800개의 바람개비가 돌아가는 것을 확인할 수 있다. createtask2 800을 실행하면 800개의 태스크 1이 동시에 실행되어 고장 난 TV 화면 같은 모습을 보여준다. 이렇게 많은 태스크를 수행해도 버벅거리지 않는 이유는 kSchedule() 함수가 문자를 출력하고 프로세서를 반환하기 때문이다. 하지만 실제 환경에서는 각 태스크 마다 우선 순위가 있을 수 있기 때문에 중요도에 따라 다른 스케줄을 부여해야한다. 따라서 이후에 태스크에 레벨을 부여하고 각 레벨에 따라 우선순위를 할당해 태스크 수에 관

계 없이 중요한 콘솔 셸의 수행을 보장하는 방법을 구현한다.

2.2 멀티레벨 큐 스케줄러로 업그레이드 및 태스크 종료기능 추가

2.2.1 태스크 우선순위와 멀티레벨 큐 스케줄러 알고리즘

멀티레벨 큐 스케줄러는 우선순위에 따라 구분된 여러 개의 큐를 사용하여 태스크를 수행하는 스케줄러이다. 태스크가 대기하는 공간이 여러 개이므로, 멀티레벨 큐 스케줄러에는 두가지의 스케줄링 정책이 필요하다. 첫번째는 여러 개의 큐 중 특정 큐를 선택하는 정책이다. 이는 시스템이 추구하는 목표에 따라 달라질 수 있는데, MINT64 OS에서는 전체 태스크를 고르게 실행하는 것을 목표로 하므로 높은 우선순위 큐에 존재하는 태스크를 모두 1회 수행한 후 낮은 우선순위의 태스크를 1회 수행하는 알고리즘을 사용한다. 두번째는 하나의 큐에서 태스크를 선택하는 정책이다. 이는 앞에서 라운드로빈 스케줄러로 구현을 이미 했다. 따라서 큐 스케줄링을 구현한다.

태스크의 우선순위를 5단계로 구분한다. 가장 높음 단계의 태스크는 다른 태스크보다 자주 스케줄링되므로 시스템 차원에서 긴급하거나 주기적으로 수행되어야 하는 태스크에 할당한다. 태스크 수행시간을 보장해야 한다면 높음으로 설정하고, 다른 태스크와 동등한 수행시간만 보장하면 된다면 중간을, 다른 태스크에 비해 시간적 제약이 적다면 낮음으로 설정하고, 작업을 완료하는데 시간적 제약이 거의 없다면 가장 낮음을 할당한다.

라운드 로빈 방식에 기반하여 큐를 선택함으로써 기아상태가 발생하는 것을 피하고, 우선순위에 따라 태스크 수행 시간을 보장해야 하므로 높은 순위의 태스크를 1회씩 수행한 후, 낮은 우선순위의 태스크를 수행하는 조건을 추가한다.

2.2.2 멀티레벨 큐 스케줄러 업그레이드

MINT64 OS의 멀티레벨 큐 스케줄러는 5개의 준비 큐와 1개의 대기 큐로 구성된다. 5개의 준비 큐는 실행할 태스크를 선택하는데 사용되므로, 관리하기 쉽도록 리스트의 배열 형태로 정의한다. 그리고 TCB자료구조에 우선순위를 추출하고 변경하는 매크로를 추가한다.

그리고 준비 리스트와 대기 리스트, 큐별 태스크 실행 횟수를 초기화한다. 그리고 태스크를 선택하는 함수인 kGetNextTaskToRun()함수를 수정하여 멀티레벨 큐 스케줄러로 동작하게 한다. 이를 위해서는 현재 큐에 있는 태스크의 개수와 현재 큐의 태스크를 수행한 횟수를 알아야 한다. 현재 큐의 태스크를 수행한 횟수가 현재 큐 큐에 있는 태스크의 개수보다 크다면 모든 태스크가 1회 이상 수행된 것이므로 다음 큐로 프로세서를 양보하면 된다. 여기서 큐에 태스크가 있지만 모든 큐의 태스크가 1회씩 실행된 경우, 모든 큐가 프로세서를 양보하여 태스크를 선택하지 못할 수 있으므로 NULL일 경우 한번 더 수행한다.

태스크에 우선순위가 추가되었으므로 준비 리스트에 태스크를 등록하는 kAddTaskToReadyList()함수도 우선순위에 맞춰 태스크를 삽입하도록 수정한다. TCB의 하위 8비트를 통해 우선순위를 알 수 있으므로 이를 통해 우선순위의 큐에 삽입하도록 한다.

태스크 우선순위는 실행중인 자신의 우선순위를 변경하거나 우선순위를 변경할 태스크가 준비 큐에 있는 경우의 두가지로 나뉠 수 있다. 첫번째의 경우 플래그 필드의 우선순위만 변경하면 수행이 끝난 직후 바로 우선순위가 적용되고, 두번째 경우 ID로 TCB를 직접 참조하여 태스크를 제거하고 플래그 필드의 우선순위 값을 변경하고 해당 큐에 삽입하면 된다. 그리고 콘솔 셸의 우선순위를 높게 등록한다.

2.2.3 태스크 종료와 유휴 태스크

태스크가 종료할 때 할당받은 자원을 해제하는 작업이 필요하므로 이를 위한 별도의 태스크가 필요하다. 지금까지의 태스크는 스택 영역을 제외하고는 별도의 메모리를 할당하지 않기 때문에 TCB와 스택만 반환하면 됐지만 앞으로는 할당된 별도의 메모리를 종료될 때 반환해야 한다. 하지만 태스크가 자신이 수행중인 메모리를 스스로 반환할 수 없으므로 가장 낮은 우선순위를 갖는 유휴 태스크와 대기 큐를 통해 이 문제를 해결한다. 대기 큐는 종료시킬 태스크를 저장하는 용도로 사용하는 큐이므로 유휴 태스크는 매번 대기 큐를 검사하여 태스크가 삽입된 경우 태스크 종료 작업을 수행한다. TCB의 플래그 필드와 스케줄러 자료구조의 실행 태스크 필드를 조합하면 태스크의 상태를 알 수 있다.

태스크를 종료하는 작업은 다른 태스크를 종료하는 경우와 자신을 종료하는 경우가 있다. 다른 태스크를 종료하려면 태스크 플래그를 종료로 설정하고 우선순위를 0xFF로 변경하여 대기 큐에 삽입하면 된다. 자신을 종료하려면 종료 플래그 설정을 하고 자신을 대기 큐에 연결하고 다른 태스크로 전환하여야 한다. 이 과정에서 호출되는 kSchedule()함수와 kScheduleInInterrupt()함수를 수정하여 종료할 태스크를 대기 큐에 삽입하도록 한다. 또한 종료할 태스크의 콘텍스트를 저장하지 않도록 한다.

유휴 태스크를 추가하여 대기 큐에서 태스크를 종료시키고, 프로세서의 사용률을 계산하여 바쁘지 않다면 프로세서를 쉬게 한다. Tick의 변화량으로 프로세서의 사용 시간을 계산한다. 그리고 HLT명령어를 통해 프로세서가 인터럽트나 예외 등이 발생할 때까지 대기하게 한다. 프로세서 사용률이 95%이하라면 프로세서를 대기모드로 진입시켜 쉬게 한다.

kGetCh()함수는 키 큐에 키가 입력될 때까지 무한 루프를 돌게 되므로 프로세서 사용시간을 전부 사용한다. 따라서 키를 대기하는 동안 다른 태스크에 프로세서를 양보하여 프로세서 사용률을 낮춘다.

2.3 태스크와 인터럽트, 태스크와 태스크 사이의 동기화 문제 해결

2.3.1 태스크 생성 문제와 동기화 처리

태스크를 추가하는 함수와 태스크를 스케줄링하는 함수의 공통점은 준비 큐를 사용한다는 것이며, 두 함수는 공통적으로 kAddTaskToReadyList함수를 사용하고 kAddListToTail함수를 호출하여 리스트의 마지막에 데이터를 추가한다. 이 작업은 한번에 이루어지지 않는다. 따라서 태스크가 추가되는 도중 인터럽트가 발생하여 태스크 링크가 변경되면 태스크가 사라지는 문제가 발생하게 된다. 따라서 이 문제를 해결하기 위해 태스크가 리스트에 작업을 완료할 때까지 인터럽트가 발생하여 리스트에 접근하는 것을 막아야 한다. 만약 두 태스크가 서로 데이터를 주고받는 용도로 리스트를 공유한다면 태스크 사이에서도 같은 문제가 발생할 수 있다. 이를 해결하기 위해 동기화를 수행한다.

첫째로, 태스크와 인터럽트 간의 동기화를 수행해야 한다면 태스크 전환을 수행하는 인터럽트를 비활성화 해야한다. 인터럽트는 외부 디바이스에서 전달하는 긴급한 신호이므로 비활성화 구간은 짧게 한다. 두번쨰로, 태스크 사이의 동기화 문제를 해결하기 위해 뮤텍스와 세마포어를 사용한다. 뮤텍스는 임계 영역에 단 하나의 태스크만 접근할 수 있도록 제어하는 동기화 객체이고, 세마포어도 뮤텍스와 비슷한 역할을 하지만 다수의 태스크가 동시에 자원을 사용할 수 있다는 점이 다르다.

2.3.2 인터럽트 제어와 동기화 객체를 통한 동기화

인터럽트 플래그는 kSetInterruptFlag() 함수로 제어할 수 있으므로 이 함수로 시스템 전역에서 사용하는 자료구조용 잠금 및 해제 함수를 구현한다. 여기서 중요한 것은 임계영역이 끝났을 때 이전 플래그의 상태를 복원해 주는 것이다. 따라서 파라미터로 이전 플래그 값을 받아 잠금 해제시에 인터럽트 플래그로 설정한다.

동기화 함수를 구현했으니 스케줄러에 관련된 함수와 키보드에 관련된 함수에 동기화 코드를 적용한다. 내부 함수는 호출관계가 서로 얽혀 있으므로 외부에서 사용하는 함수에만 동기화 코드를 삽입하는 것이 좋다. 스케줄러에 관련된 함수에서는 kCreateTask()함수와 kScheduleInInterrupt()함수가 있다. 다른 태스크나 인터럽트 핸들러와 데이터를 공유하는 부분을 동기화 처리해주면 된다. 키보드에 관련된 함수 중 키 큐를 공유하는 함수인 kConvertScanCodeAndPutQueue()함수와 kGetKeyFromKeyQueue()함수에 동기화처리를 한다.

이제 태스크 사이의 자료 공유를 위한 동기화 객체를 설계하고 구현해 본다. 코드가 복잡해지면 의도치 않은 중복 잠금 및 해제가 발생할 수 있으므로 뮤텍스를 구현해 본다. 태스크에서만 사용하는 데이터는 인터럽트 핸들러에서 사용하지 않으므로 굳이 인터럽트를 불가능하게 설정할 필요가 없다. 임계 영역에 진입하는 태스크 수를 인터럽트를 사용하지 않고 제어해야 한다. 태스크 수는 임계영역 진입여부를 나타내는 플래그와 잠금 횟수를 나타내는 카운터, 그리고 임계영역에 진입한 태스크의 ID만 있으면 제어할 수 있으므로 이를 토대로 뮤텍스의 자료구조를 정의해 본다. 패딩을 추가하여 8바이트의 배수로 맞춘다. 자료구조를 정의했으면 뮤텍스 자료구조를 초기화한다. 뮤텍스는 중복 잠금을 허락하므로 잠금을 수행한 태스크와 현재 태스크를 비교해 같다면 한번 더 잠금을 하고 다르다면 잠금이 해제될 때까지 대기한다. 해제 함수 역시 같은 흐름이다.

하지만 잠금 상태를 판단하고 설정하는 과정이 하나의 명령어로 처리되지 않았기 때문에 이 과정에서 인터럽트가 끼어들 수 있다. 따라서 LOCK명령어와 CMPXCHG명령어를 사용하여 잠금 테스트와 설정 과정을 하나의 명령어로 처리하여 해결한다. CMPXCHG명령어는 두개의 오퍼랜드가 필요하며, AX레지스터 값과 첫번째 오퍼랜드를 비교한 결과에 따라 두가지 방식으로 동작한다. 일치한다면 RLFAGS레지스터의 ZF비트를 1로 설정하고 첫번째 오퍼랜드에 두번째 오퍼랜드의 값을 대입한다. 하지만 다르다면 ZF비트를 0으로 설정하고 첫번째 오퍼랜드의 값을 AX레지스터로 옮긴다. 따라서 kTestAndSet()함수로 테스트와 설정 과정을 태스크 전환 없이 처리하고 이 함수를 kLock()함수와 kUnlock()함수의 잠금 플래그 설정 부분에 적용한다.

멀티태스킹 환경에서 kPrintf()함수도 동기화 문제가 생길 수 있다. 화면에 메시지를 출력할 경우 화면 내의 커서 위치를 나타내는 필드를 먼저 사용하려고 경쟁할 것이다. 한 태스크가 메시지를 출력 후 커서 위치 필드를 갱신하기 전 다른 태스크로 전환되고 전환된 태스크가 화면에 메시지를 출력하게 되면 덮어써지는 문제가 생길 수 있다. 따라서 kPrintf()함수와 gs_qwAdder변수를 사용하는 부분을 뮤텍스 처리한다.

ConsoleShell파일의 killtask커맨드의 기능을 수정하고 testmutex 커맨드를 추가한다. killtask커맨드의 기능을 입력받은 특정 태스크를 종료하는 기능에서 추가하여 태스크 ID로 0xFFFFFFFF이 들어오면 0번과 1번 태스크를 제외한 모든 태스크를 종료하게 하고, testmutex커맨드를 추가하여 간단하게 동기화 기능을 테스트한다.

2.4 멀티스레딩 기능 추가

2.4.1 멀티태스킹과 멀티스레딩

프로그램은 메모리에 로드하여 실행할 수 있는 코드와 데이터의 집합이다. 이러한 프로그램이 메모리에 로드되어 실행 가능한 상태가 되면 프로세스가 된다. 프로세서는 독립된 메모리 공간과 콘텍스트, 스택을 가지기 때문에 여러 개의 프로세스를 동시에 수행할 수 있다. 이를 멀티태스킹이라고 한다. 프로세스의 내부에 코드를 실행하는 실행 단위를 스레드라고 한다. 프로세스가 메모리에 로딩될 때 프로세스의 코드를 실행하는 스레드도 같이 생성되는데 이를 메인 스레드라고 한다. 프로세스 내부에서 여러 개의 스레드를 생성하여 동시에 작업을 진행하는 것을 멀티스레딩이라고 한다.

스레드는 개별적인 메모리 공간을 할당받지 않고 프로세스와 공유한다. 이것으로 인해 서브

스레드가 동작하는 동안 메인 스레드가 종료되면 프로세스의 메모리 공간이 사라지므로 서브스레드에 문제가 생긴다. 이것을 방지하기 위해 메인 스레드는 프로세스 내에 존재하는 다른 스레드가 종료될 때까지 무조건 대기해야 한다.

2.4.2 멀티스레딩 지원을 위한 재설계와 구현

프로세스는 메모리 영역과 실행중인 스레드의 정보가 필요하다. 프로세스가 실행되는 도중 얼마나 많은 스레드를 생성할지 모르기 때문에 리스트를 사용하여 자식 스레드를 처리한다. 기존의 태스크 자료구조에 생성한 태스크를 연결하는데 사용되는 별도의 링크가 필요하다. 스레드 링크(stThreadLink)와 자식 스레드 리스트(stChildThreadList)를 기존 태스크 자료구조에 추가한다. 태스크 자료구조의 플래그 필드에 프로세스와 스레드 플래그도 추가한다.

스레드를 생성하는 경우는 생성하는 스레드가 속한 프로세스의 정보를 가져와서 설정해야 하기에 태스크 생성 함수에 파라미터를 추가한다. 또한 스레드를 생성할 때 스레드 링크의 어드레스를 넘겨줘서 스케줄러 링크에 영향이 가지않도록 한다. 유휴 태스크는 스레드를 종료하면서 관계를 끊는 역할을 한다. 프로세스가 종료될 때 프로세스의 자식 스레드 리스트에 연결된 스레드를 모두 종료 시켜야 한다. 만약 종료하는 태스크가 스레드라면 스레드 종료와 동시에 프로세스의 자식 스레드 리스트에서 자신을 제거해야 한다. kExitTask()로 태스크 자기 자신을 종료하는데 kExitTask()를 호출하지 않고 엔트리 포인트 함수에서 리턴하면 어떤 어드레스로 이동하여 코드를 실행할지 알 수 없다. 따라서 이를 막기 위해 태스크 생성할 때 스택에 가장 마지막에 kExitTask()함수의 어드레스를 넣어줘서, 엔트리 포인터 함수에서 리턴할 때 kExitTask()함수로 이동하게 한다.

2.5 추첨 스케줄러 추가 구현

기존의 멀티레벨 큐를 기반으로 Lottery Scheduling을 구현한다. 시스템 전체의 티켓 수를 상수로 정의하지 않고 프로세스의 수와 우선순위에 따라 유동적으로 바뀔 수 있게 구현한다. 멀티 레벨 큐에는 5 단계의 우선순위가 있는데, 각 우선순위에 16, 8, 4, 2, 1 개의 티켓을 부여한다. 각 큐에 하나 이상의 프로세스가 존재할 때마다 그 큐의 우선순위의 티켓을 총 티켓 개수에 더하여 전체 티켓 개수를 구한다. random number를 총 티켓 갯수로 나누어 한 숫자, winner를 선택한다. 그리고 ready list를 순회하면서 카운터 값이 winner를 초과할 때까지 각 추첨권 개수를 카운터에 더한다. 값이 초과하게 되면 리스트의 현재 원소가 당첨자가 된다. <코드 1>

random number는 다음 논문의 마지막에 언급된 A Random Number Generator의 Park-Miller pseudo-random number generator를 C 언어로 구현하여 얻는다. Random number의 seed는 앞서 구현한 타임스탬프 카운터에서 얻은 값을 사용했다. <코드 2, 3>

```
/**
 *  태스크 리스트에서 다음으로 실행할 태스크를 얻음
 */
static TCB* kGetNextTaskToRun( void )
{
    TCB* pstTarget = NULL;
    int iTaskCount, i, j;

    int counter = 0;
    int total_ticket = 0;

    for ( int k = 0; k < TASK_MAXREADYLISTCOUNT ; k++)
    {
        if ( kGetListCount( &( gs_stScheduler.vstReadyList[ k ])) > 0)
            total_ticket += cal_ticket(k);
    }

    int winner = random_generator() % ( total_ticket + 1 );

    // 큐에 태스크가 있으나 모든 큐의 태스크가 1회씩 실행된 경우, 모든 큐가 프로세서를
    // 양보하여 태스크를 선택하지 못할 수 있으니 NULL일 경우 한번 더 수행
    for( j = 0 ; j < 2 ; j++ )
    {

        counter = 0;

        // 높은 우선 순위에서 낮은 우선 순위까지 리스트를 확인하여 스케쥴링할 태스크를 선택
        for( i = 0 ; i < TASK_MAXREADYLISTCOUNT ; i++ )
        {
            iTaskCount = kGetListCount( &( gs_stScheduler.vstReadyList[ i ] ) );

            if ( iTaskCount == 0 )
                continue;

            counter += cal_ticket(i);

            if (counter >= winner){
                pstTarget = ( TCB* ) kRemoveListFromHeader(
                                    &( gs_stScheduler.vstReadyList[ i ] ) );
                gs_stScheduler.viExecuteCount[ i ]++;
                pstTarget->got_time += 1;
                break;
            }
        }
```

<코드 1. 추첨 스케줄러 구현>

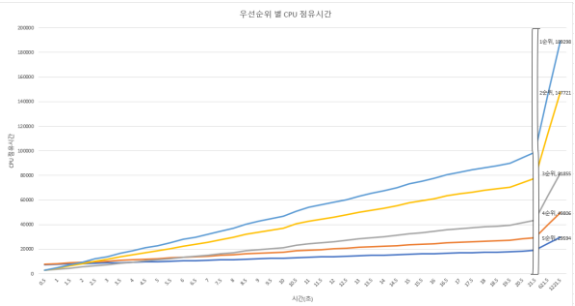
```
#ifndef __RANDOM_H__
#define __RANDOM_H__

#include "Types.h"
#include "AssemblyUtility.h"
#include "Task.h"

QWORD random_generator();
QWORD cal_ticket(BYTE priority);
QWORD rand();
void srand(QWORD seed);

#endif
```

<코드 2. 랜덤함수 헤더>

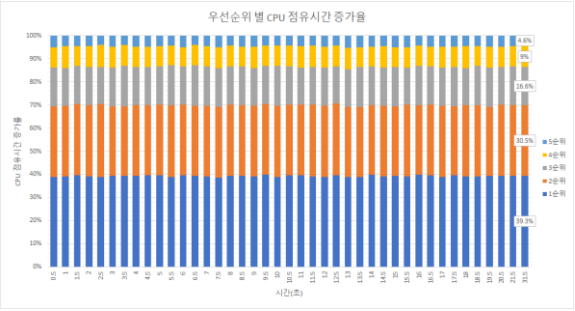
```
#include "random.h"

static QWORD next = 1;

QWORD random_generator(){
    QWORD seed;
    QWORD hi, lo;

    seed = kReadTSC();

    lo = 16807 * ( seed & 0xFFFF );
    hi = 16807 * ( seed >> 16 );

    lo += ( hi & 0x7FFF ) << 16;
    lo += hi >> 15;

    if ( lo > 0x7FFFFFFF ) lo -= 0x7FFFFFFF;


    return ( seed = ( QWORD ) lo );
}

QWORD cal_ticket(BYTE priority){
    switch(priority){
        case TASK_FLAGS_HIGHEST:
            return 16;
        case TASK_FLAGS_HIGH:
            return 8;
        case TASK_FLAGS_MEDIUM:
            return 4;
        case TASK_FLAGS_LOW:
            return 2;
        case TASK_FLAGS_LOWEST:
            return 1;
    }
}
```

<코드 3. 랜덤함수 구현>

구현한 추첨 스케줄러의 공정성을 평가하기 위해 우선순위 별 CPU점유시간과 우선순위 별 CPU점유시간 증가율을 표로 나타내었다. 시간이 지날 수록 CPU점유시간이 추첨권 개수(16 : 8 : 4 : 2 :1)에 비례하는 것을 확인 할 수 있고, 증가율 또한 40 : 30 : 16 : 9 : 4로 추첨권 개수에 비례하게 됨을 확인할 수 있다. 완벽하게 공정하진 않지만 추첨권에 따라 CPU점유시간이 달라짐을 확인해 보았다.



<표 1. 우선순위 별 CPU점유시간>



<표 2. 우선순위 별 CPU점유시간 증가율>

참고한 논문: Waldspurger, Carl A., and William E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management." (1994).

개발 시 경험한 문제점 및 해결점

처음에 추첨 스케줄링 구현 시 각 Task에 우선순위에 따라 ticketNum을 할당하였다. (각 TCB마다 LINK에 필드로 추가하여서 할당하였다.) 그리고 다음 Task를 선택할 때 그 ticket을 가지고 선택했는데, 계속 무한루프가 돌고 키보드가 안쳐졌다. 그래서 task마다 LINK에 필드를 추가하는 것이 아니고 vstReadyList에 각 큐마다의 task가 있으면, 우선순위에 따라 티켓을 부여하고, 당첨된 큐에 있는 task를 다음에 실행하는 방식으로 구현해서 공정성 있는 추첨 스케줄링을 구현할 수 있었다.

3. 조원별 기여도

신정은: 34%
조재호: 33%
조한주: 33%