Operating Systems

Assignment #1 KU_CFS

컴퓨터공학과

201811259

배수빈

■ Basic Design

-CFS 란 ?

Completely Fair Scheduler로 리눅스 스케줄러의 한 방식이다. 이 스케줄러에서 모든 task는 virtual runtime에 기반해서 해당 task의 PCB (process Control Block) 가 red-black tree에 저장이 된다. virtual runtime은 weight값을 고려한 time으로 다음 식을 통해 계산된다.

vruntime = vruntime + DeltaExec * (weight 0 / weight p)

weight값은 프로세스마다 가지는 nice값에 따라 결정이 되는데 nice값이 크면 weight p 의 값이 작아져 weight 0 / weight p 의 값이 커지게 되고 이는 vruntime 계산시 DeltaExac에 큰 가중치를 주기 때문에 vruntime이 커진다.

이는 cpu bound한 프로세스는 cpu를 할당할 확률을 작게 하고 I/O bound한 프로세스는 cpu를 할당할 확률을 크게 하기 위한 방식이다. CFS는 vruntime 이 작은 프로세스의 PCB는 red-black tree의 오른쪽에 위치하고 vruntime이 큰 프로세스의 PCB는 red-black tree의 왼쪽에 위치하도록 관리한다.

스케줄러는 red-black tree의 가장 왼쪽의 프로세스에게 cpu를 할당해 주고 할당 시 tree구조에서 해당 PCB를 뺀다. 스케줄러가 실행 중이던 프로세스에게서 cpu자원을 interrupt해서 다른 프로세스에게 cpu자원을 할당해야 할 때는 실행을 멈추게 된 프로세스의 vruntime을 위의 식대로 계산한 뒤 그 프로세스의 PCB를 vruntime에 따라 정렬된 tree의 알맞은 위치에 넣도록 한다.

-ku_cfs에서의 CFS 구현 방식

스케줄러가 vruntime을 계산하고 그 vruntime의 크기에 따라 자료구조에 정렬하는 방식은 CFS와 동일하다. 또한 timeslice만큼 하나의 프로세스가 실행되고 timeslice만큼의 시간이 지나면 운영체제가 cpu를 interrupt해서 프로세스를 스케줄링하는 방식도 구현하였다. 실제 cpu자원을 할당하고 interrupt해서 다른 프로세스에게 할당하는 것을 직접 구현할 수는 없어서 fork를 통해 멀티 프로세스를 구현하고 시그널을 통해 실행을 interrupt한 뒤 스케줄링 하는 것을 흉내를 내는 방식으로 구현했다.

ku_cfs에서 스케줄러는 응용프로그램, 즉 user process로 구현하였다. 그리고 CFS에서의 red-black tree는 linkedList로 구현하여 레디큐의 역할을 하도록 하였다. 또한 스케줄러가 관리해야 하는 여러 프로세스들은 스케줄러 역할을 하는 부모프로세스에서 fork를 통해 생성된 자식 프로세스들이다. 이러한 자식 프로세스들을 스케줄링하기 위한 interrupt는 시스템콜을 이용하여 부모프로세스에 시그널을 보내는 방식으로 구현하였다. 자식프로세스가 timeslice만큼 실행되면 부모프로세스에 시그널을 보내는 역할은 itimer가 하도록 했다. 해당 프로그램에서의 timeslice는 1초로 설정해 두었으므로 1초마다 부모에게 SIGALRM이 보내지는 것이고, 스케줄링 작업은 SIGALRM에 해당하는 시그널 핸들러가 수행하는 것이다.

-PCB구현

ku_cfs에서 PCB는 구조체로 구현하였다. 구조체의 이름은 Node이고 Node구조체의 멤버로는 Node구조체타입의 포인터 변수인 prev와 next가 있고, 어떤 프로세스에 대한 Node인지 알 수 있는 int타입의 myPid, 해당 프로세스의 Virtual runtime인 int타입의 vruntime 그리고 int타입의 nice 까지 총 다섯개의 멤버가 있다.

관리해야하는 프로세스가 많은 만큼 PCB도 많이 필요하기 때문에 구조체 변수를 선언하기 쉽도록 typedef로 구조체의 별칭또한 Node로 지정해주었다. 해당 구조체 변수나 포인터 변수를 선언할 때에는 Node * nodePointer 또는 Node node 와 같은 형식으로 선언할 수 있도록 하였다.

-Ready Queue구현

ku_cfs에서 레디큐는 red-black tree가 아닌 linkedList 자료구조의 형태이다. linkedList는 구조체로 구현하였다. 해당 구조체가 가지는 구조체 멤버로는 Node* 타입의 head와 tail이다. linkedList의 가장 앞부분에 해당하는 노드가 head이고 가장 뒷부분에 해당하는 노드가 tail이다. head와 tail에는 프로세스정보가 들어있는 노드가 지정되지는 않고 정말 linkedList의 'head'와 'tail'의 역할만 하도록 한다. 실제 프로세스가 linkedList에 연결될 때는 head의 next부터 tail의 prev까지 node가 지정되는 방식이다.

linkedList에 ready 상태인 프로세스에 해당하는 노드를 insert해야할 때는 linkedList의 head노드부터 차례대로 해당노드의 vruntime과 비교하고 해당노드의 vruntime보다 작거나 같은 값이 나오면 그 노드의 next에 해당노드를 지정해준다. linkedList에 vruntime에 따라 정렬된 상태로 저장이 되기 때문에 스케줄러가 linkedList->head->next로 실행해야할 프로세스를 바로 참조할 수 있기 때문에 유용한 방식이다.

-프로세스 구현

스케줄러에 해당하는 프로세스는 ku_cfs의 부모 프로세스이다. 또한 스케줄러가 관리하는 여러 프로세스들은 해당 부모프로세스가 fork하여 생성한 자식 프로세스들이다. ku_cfs를 실행할 때 첫 번째부터 다섯 번째 인자는 각각 nice값을 -2, -1, 0, 1, 2로 갖는 자식 프로레스들의 개수다. 그러므로 각각의 개수만큼 프로세스를 fork한다.

부모 프로세스는 fork한 프로세스에 대해서 node를 생성하고 node의 pid는 해당 자식 프로세스의 pid로, vruntime은 0으로, nice값은 해당 nice값으로 지정한다. ku_cfs에서 자식프로세스는 fork이후 바로 execl함수를 통해 ku_app프로그램을 실행하도록 한다. ku_app은 인자로 받아오는 문자 하나를 0.2초마다 출력하는 프로그램이다. 이때 먼저 생성되는 자식 프로세스부터 차례대로 ku_app에 넘기는 문자를 A부터 B, C .. Z순으로 지정한다. 이때 넘기는 문자의 타입은 char[]타입이고 크기는 1이며 char[0]에 "A"를 지정하고 fork이후 char[0]++를 하여 다음 fork되는 프로그램에는 그다음 문자열이 넘어갈 수 있도록 하였다.

-스케줄링 구현

ku_cfs에서 interrupt는 itimerval구조체 변수인 timer가 it_interval의 간격마다 SIGALRM을 울리고 SIGALRM에 연결된 시그널핸들러가 작동되는 방식으로 구현하였다. ku_cfs에서 timeslice는 1초로 정해져 있기 때문에 INTERVAL값을 1로 define해둔 뒤에 이를 timer의 it_interval 값으로 설정했다. 또한 timeslice가 실행되어야 하는 횟수는 ku_cfs실행 시 6번째 인자로 받아오는데 실행시간을 제어하기 위하여 timesliceCount라는 변수를 선언해두고 핸들러가 불릴때마다 +1하여 핸들러가 몇번 불렸는지 저장해 둔다. 만약 timesliceCount가 6번째 인자로 받아오는 숫자인 timesliceNum과 같아지면 부모프로세스의 실행을 종료하도록 한다. 이때 itimer는 부모프로세스가 종료하면 기능을 멈추고 더 이상의 시그널을 발생하지 않기 때문에 timesliceNum만큼의 timeslice 동안은 부모프로세스가 계속 작동해야 한다. 따라서 while문으로 무한루프를 실행하다가 timesliceCount가 timesliceNum와 같아지면 바로 무한루프를 빠져나오는 방식으로 구현했다.

itimer가 INTERVAL만큼의 시간마다 SIGALRM을 발생히면 해당 시그널에 연결된 시그널 핸들러인 timer_handler를 실행한다. 시그널 핸들러는 sigaction메소드를 이용하여 연결했다. 이를 위해서 sigaction구조체 변수인 action을 선언하고 action의 sa_handler로 timer_handler를 지정해주고 sigaction을 통해 SIGALRM시그널과 action을 연결하였다.

타이머 핸들러는 timer_handler함수로 구현하였다. 이 함수는 Node구조체의 포인터변수인 runningNode를 활용한다. 이는 해당 프로그램이 시작되어 아무 자식도 실행상태가 아닐 때에는 아무 값도 저장되어 있지 않고 그저 Node만큼 동적할당 되어 Node* 타입으로 형 변환 되어있는 상태이다. itimer가 setitimer메소드를 통해 실행이 되어 가장 처음 SIGALRM이 울리고 핸들러가 호출이 되면 runningNode는 mylinkedList->head->next노드로 지정이 되고 deleteNode()메소드를 통해 runningNode를 linkedList에서 빼낸다. 그리고 runningNode가 가지는 pid정보를 통해 해당 프로세스가 실행되도록 SIGCONT시그널을 보낸다.

핸들러가 호출이 되었을 때 runningNode가 어떤 노드로 설정이 되어있는 상태라면 (어떤 프로세스가 실행중인데 핸들러가 호출이 되었을 경우) runningNode의 vruntime을 계산해서 다시 저장한다. 이때 vruntime은 ***vruntime = vruntime + DeltaExec * 1.25^(nice)*** 식에 따라 계산이 된다. ku_cfs에서 DeltaExec은 timeslice인 INTERVAL, 즉 1초이고 1.25^(nice)는 미리 계산해서 constantValue라는 배열에 저장해 두었다. node의 nice값이 -2일 경우 constantValue[node->nice + 2] 로 값을 활용할 수 있도록 한 것이다. runningNode의 vruntime을 위의 식에 따라 증가시킨 뒤 insertNode()메소드를 이용해 runningNode를 다시 linkedList에 연결한다. 그리고 runningNode의 pid에 SIGSTOP시그널을 보내서 실행을 멈추도록 한다. 실행중이던 프로세스를 멈추었으니 다른 프로세스를 실행시켜야 한다. linkedList에는 가장 vruntime이 작은 노드부터 저장이 되어있기 때문에 runningNode를 linkedList->head->next로 지정한 뒤 runningNode를 deleteNode메소드를 통해 linkedList에서 삭제한다. 이후 runningNode에 SIGCONT시그널을 보내 실행시킨다. 핸들러를 실행할때마다 timesliceCount가 증가하기도 하는데 이는 위에서 언급한 방식대로 활용된다.

■ Description for important functions

newList	Funtionality	***linkedList변수를 생성하고 구조체 멤버들을 설정한 뒤 그 변수를 리턴하는 함수** -linkedList구조체 포인터 변수를 선언하고 동적할당 해준다. -생성한linkedList의 head와 tail을 newNode함수를 이용하여 생성한다. -head의 next노드를 tail로, prev노드를 NULL값으로, tail의 prev노드를 head로, next노드를 NULL값으로 설정한다.
	Parameters	X
	Return Value	linkedList*
newNode	Funtionality	*** Node변수를 생성하고 인자값들로 구조체 멤버들을 설정한 뒤 그 변수를 리턴하는 함수** -Node구조체 포인터 변수를 선언하고 해당 변수와 prev와 next노드를 동적할당 해준다. -인자로 받아오는 값들로 구조체 멤버들을 설정한다. -해당 노드의 prev와 next노드를 NULL값으로 설정한다.
	Parameters	int Pid, double vruntime, double nice
	Return Value	Node*
insertNode	Funtionality	*** 파라메터로 받아오는 노드를 linkedList의 알맞은 위치에 연결해준다.** -linkedList내부의 노드들은 linkedList의 head쪽부터 vruntime이 작은 노드부터 정렬이 된 상태로 link되어 있기 때문에 head노드부터 vruntime을 비교해가며 알맞은 위치에 노드를 연결해준다.
	Parameters	Node* node
	Return Value	void
deleteNode	Funtionality	*** 파라메터로 받아오는 노드를 linkedList에서 제거해준다.** -linkedList는 각 노드들이 서로 next 와 prev로 앞뒤의 노드를 설정해주며 link하는 방식이기 때문에 인자로 받아오는 노드의 next 와 prev를 NULL값으로 지정해주고 해당 노드의 앞뒤에 위치하던 노드들의 next와 prev도 서로를 연결하도록 해준다.
	Parameters	Node* node

	Return Value	void
timer_handler	Funtionality	*** signum에 해당하는 시그널이 발생했을 때 실행하는 핸들러. 실행중인 프로세스에게 SIGSTOP시그널을 보내고, 가장 vruntime 이 작은 프로세스에서 SIGCONT시그널을 보낸다.** -핸들러를 한번 호출할 때마다 timesliceCount를 1씩 증가시킨다. (해당 프로그램을 실행할 때 6번째 인자로 넘겨주는 숫자만큼만 핸들러가 불려야하기 때문이다. 이에 대한 처리는 main에서 이루어진다.) -실행중이던 프로세스를 멈춘다. runningNode의 vruntime을 증가시켜준다. runningNode를 insertNode메소드를 통해 linkedList에 다시 넣어준다. runningNode의 pid를 통해 실행중이던 프로세스에 SIGSTOP시그널을 보낸다. -vruntime이 작은 프로세스를 실행시킨다. runningNode를 linkedList의 head의 next로 설정한다. runningNode를 deleteNode메소드를 통해 linkedList에서 제거한다. runningNode의 pid를 통해 해당 프로세스에 SIGCONT 시그널을 보낸다.
	Parameters	int signum
	Return Value	void