Thread

Review of Process

Thread는 Process의 refined된 형태이다

refined?

Process – abstraction for

Execution unit – 스케줄링 단위

Protection domain – 소유하고 있는 자원에 대한 보호

implementation?

register들로 구현된다.

Process는 Text-Data-Stack으로 구성된다.

Program 과 Process관계

프로그램이 메모리에 올라와서 CPU에서 명령어를 수행시키는 상태

Executing program with a single thread of control

→ 지금까지는 프로그램을 실행시키는 것을 a single thread of control이라 생각했었다.

→ sequential하게 하나씩 실행된다는 의미이다. 이것을 전제했었다. (executional flow가하나이다.)

지금까지의 프로세스 개념

하나의 실행 흐름을 가지고 실행 중인 프로그램

→ 이 말은 하나씩 하나씩 수행한다는 의미이다.

→ 하나의 실행의 흐름을 가진다는 의미이다.

1개의 실행 흐름을 여러 개로 만든다면?

→ multicore때문에 이를 생각해야 하는 것이다.

core는 독립적인 수행이 가능

교수님 질문 : core의 수가 많아지면 성능도 좋아지나?

core가 4배로 많아지면 성능이 4배로 좋아지나?

A : 프로그램 자체의 성능은 빨라지지 않는다. 왜냐하면 프로그램이 sequential하게, executional flow가 하나이면 내 프로그램은 하나의 코어만을 쓸 수 있다.

단, 다른 프로그램이랑 같이 돌고 있었다면 스케줄링에 있어서 CPU를 더 자주 할당 받을 수 있었을 것이다

Thread란 무엇인가?

정의

Execution Unit

프로세스 내의 실행 흐름

프로세스 보다 작은 단위(finer grain)

→ 더 작은 단위가 필요하다!

프로세스를 나누어서 여러개의 executional flow를 만드는 것이다.

프로세스와 같은 protection domain은 없음

→ 같은 프로세스 소속이면, Address space를 공유한다.

→ 같은 메모리를 공유하지만, 다른 control flow, 독립적인 execution을 한다.

동기

하나의 프로세스에는 하나의 control만이 존재하기 때문에 한 번에 하나의 일만을 처리

프로세스에서 할 작업을 여러 개로 나눈 후에 각각을 thread화 한다면 병렬적으로 작업을 완수할 수 있음

→muti core를 동시에 쓰려면 thread를 사용해야한다는 의미이다.

Cooperative process와의 차이점

→ shared memory와 다른 점이 무엇인가?

Cooperative process는 프로세스간 통신이 필요 – 비용이 많이 듦

→ explicit한 통신이 필요. 이는 유저가 표현해야함 → 많은 비용!

thread는 implicit하다.

프로세스간의 문맥 교환 비용

Process 내에서 cooperation 하는 thread로 만든다면

Process보다 적은 비용으로 cooperative process가 하는 일을 동일하게 수행 가능

여러 프로세스를 만드는 것보다 thread를 쓰는 것이 더 저렴한 가격으로 multi core를 쓸 수 있다

Thread와 CPU unitilzation

CPU = core

Thread를 늘릴수록 througput이 늘지는 않는다.

core의 수를 늘릴수록 피크점의 thread가 늘어난다.

그러나 어떤 피크가 되면 성능이 줄어든다.

Thread의 수가 증가할수록 CPU의 utilization이 증가

→ 임계값을 넘어가면 다시 감소 → thread switching 비용이 증가함

CPU의 수가 많은 시스템일수록 thread를 이용하는 게 유리

Multi-processor, 즉 CPU의 수가 많을수록 한 process의 여러 thread를 parallel하게 실행

→ 코어를 4개를 쓴다면

Process와 Thread

Process

하나의 thread(실행 흐름)을 가짐

Process간의 memory는 독립적이므로, 서로의 영역 접근 못함

Process간의 switching 비용이 큼 (상대적으로 heavyweight)

Thread

하나의 프로세스 안에 여러 개의 thread가 존재

프로세스의 code영역과 data영역은 thread간에 공유

thread들은 같은 memory영역을 공유하여

thread간 switching 비용이 적음 (process switching 보다 lightweight)

→ 같은 메모리주소에 있으니 비용이 적다.

Called “lightweight process” (lwp)

→ 그래서 이렇게 표현하기도 한다.

Thread의구성요소

process에서는 PCB가 있었다.

Thread도 실행에 관련된 자료 구조가 필요

Thread ID – thread 식별자

Program counter – 현재 실행중인 instruction의 주소

Register set – CPU의 레지스터 값들

Stack per thread

→ thread마다 Program counter과 Stack이 필요하다.

→ Stack은 호출하는 지역변수들을 가지고 있기에 별도로 필요

→ 이 Stack의 크기는 일반적으로 8kb이다.

동일한 프로세스 내에 있는 thread가 공유하는 것

Code – 프로그램의 code section → Text를 의미

Data – 프로세스의 data section

File – 프로세스에서 open한 file

→ 많은 것을 공유한다.

위의 이유로 thread switching 비용이 적음

단일 thread와 multithreaded process

multithreaded process의 thread가 Code, Data, File에 접근하려면 먼저 동기화가 필요하다.

thread가 동기화를 해야한다.

Multithread program의 장점

Responsiveness

프로그램의 어느 부분을 담당하는 thread가 block되거나 시간이 걸리는 작업을 하더라도, 다른 thread들은 실행되고 있기 때문에 user의 입장에서는 그 프로그램이 interactive하다고 볼 수 있음

→ read()같은 것을 했다고 했을 때 그 thread는 기다려야하지만, 나머지는 작동가능

Resource sharing

Thread들간에는 프로세스의 memory와 다른 자원들을 공유

→ 프로세스의 메모지 주소를 공유한다.

Economy

Thread들은 단일 프로세스 memory영역에서 실행되기 때문에 새로운 프로세스를 생성하는 것보다 thread를 생성하는 것이 비용이 적게 들어감

→ fork-exec을 통해 프로세스를 만드는데, 이는 무겁다.

→ thread는 가볍게 만들수 있다. stack만 만들어도 된다.

Scalability

여러 개의 thread가 각각 다른 processor에서 동시에 실행 가능(parallelism)

core가 늘어나면서 성능도 늘어난다.

Multicore Programming

최근 프로세스 설계 동향은 하나의 chip에 여러 개의 computing core를 탑재 □ multicore processor

Multithread programming은 multicore 시스템에서도 효율적

Multicore processor는 운영체제에서 각각의 core를 하나의 프로세서로 인식하고 스케줄링

각각의 thread를 core에 할당하여 실행 가능

multicore는 cache를 공유하기 때문에 data, code 등 프로세스의 자원을 공유하는 multithreaded programming에 보다 효과적임

→multicore는 cache를 공유하기에 효율적이다.

Q. Scalability에서 processor는 프로그램을 나타내는 의미의 processor가 아닌 cpu, 코어 수를 나타내는 것이지요?

A. yes

Q. thread를 이용해서 한 프로세스를 나눈다면, 다른프로세스로의 전환은 언제 일어나나요?

Q. 그려면 타임 퀀텀은 어떻게 되나? thread에게 시간을 어떻게 줄것인가?

Q. 다른 프로세스로 전환이 되면 thread는 어떻게 되나? (contex swicthing)

A. Thread들이 다 멈춰야 한다.

숙제 : thread의 타임퀀텀? 쪼개서주나? 그대로주나?

User and Kernel threads – User thread

Thread를 지원하는 주체에 따라 나뉨

User threads

처음에는 유저레벨에서만 지원

커널 영역 위에서 지원되며 일반적으로 user level의 라이브러리를 통해 구현됨

→ 유저 레벨에서 라이브러리를 만듦

라이브러리에서 thread를 생성, 스케줄링과 관련된 관리를 해줌

장점

동일한 메모리 영역에서 thread가 생성, 관리되므로 이에 대한 속도가 빠르다.

굉장히 많이 만들어 질 수 있다.

단점

여러 개의 user thread 중에서 하나의 thread가 system call 요청으로 block이 된다면 나머지 모든 thread 역시 block된다.

→ execution은 하나이다.

→ thread끼리도 스케줄링을 하는 것이다.

Kernel은 여러 개의 user thread들을 하나의 process로 간주하기 때문

→ 커널에서 수행되는 thread가 하나라는 의미이다.

교수님 질문

Q. 왜 커널이 한개의 thread만을 지원했을까?

Q. 커널에서 멀티 thread를 지원하기 어려운 이유가 무엇일까? 뭐가 어려울까?

A. interrupt? 수행을 하던 것을 멈추고 다른 걸 하다가 다시 돌아옴

그런데 커널에 interrupt가 발생하면 어찌하는가?

→ 커널이 하던 일을 마치고 그 다음에 interrupt 서비스를 하는 것이다

→ 원래 주로 던 커널은 monoithic 커널이었다. 왜?

→ 커널의 data structure가 있는데, 그런데 여기에 여러 thread가 있으면 어떻게 되는가?

→ 동기화를 해줘야 한다.

→ 여기에 마구 접근하다면 커널이 제대로 동작할 수 없다.

결론

멀티 쓰레드를 허용하려면

1. interrupt를 허용하는 point를 만들어 줘야한다.

→ 커널은 동기화를 시키지 않고 있었기에 멀티 쓰레드가 어려웠던 것이다.

→ 커널의 data structure를 보호해줘야하는 것이다.

2. 언제 interrupt를 허용할 것인지를 정해줘야 한다.

→

Q: user level thread에서 한번에 하나의 thread만 동작한다면 굳이 하나의 process로 실행하는 대신 thread를 나누어 사용하는 장점이 무엇인가요??

A: 그 당시에는 multiprogramming이 나왔다.

그런 시대였는데 하드웨어가 먼저 나와버린 것이다.

그 요구에 맞춰 커널에는 하나만 올라가더라도 유저레벨에는 여러개를 올리려고 하는 것이다.

→ 비용이 적게 든다.

User and Kernel threads – Kernel thread

Kernel Thread

- 운영체제에서 thread를 지원
- 커널 영역에서 thread의 생성, 스케줄링 등을 관리
- 장점
  - Thread가 system call을 호출하여 block이 되면, kernel은 다른 thread를 실행함으로써 전체적인 thread blocking이 없음
  - Multiprocessor 환경에서 커널이 여러 개의 thread를 다른 processor에 할당할 수 있음
  - → 여러개의 core를 가능
- 단점
  - User thread보다 생성 및 관리가 느림

Mapping of User & Kernel Thread : Many to One

User & Kernel Thread을 어떻게 맵핑시킬 것인가에 대한 문제

Many-to-One

- Thread 관리는 user level에서 이루어짐
- 여러 개의 user level thread들이 하나의 kernel thread*로 매핑됨
- Kernel thread를 지원하지 못하는 시스템에서 사용됨
- → 지원하더라도 제한을 거는 경우이다.

- 한계점
  - 한번에 하나의 thread만 커널에 접근 가능
    - 하나의 thread가 커널에 접근(calling system call)하면 나머지 thread들은 대기해야 함
    - 진정한 concurrency는 지원하지 못함
    - 동시에 여러 개의 thread가 system call 사용 불가

  - Kernel의 입장에서 여러 개의 thread는 하나의 process이기 때문에 multiprocessor 이더라도 여러 개의 processor에서 동시에 수행 불가능

다른 model과 비교하기 위해 kernel thread라 한 것이며, many-to-one model에서는 multithread 프로그램이 실행되면 thread library에서 여러 개의 thread들을 스케줄링 한다. 이렇게 하여, 사용자로 하여금 하나의 프로그램이 여러 개의 thread가 동작하고 있는 것처럼 착각하게 한다.

Many to one 예시

커널 thread가 하나만 있기에 커널 thread를 유저 thread가 multiplex해야한다는 것이다.

Mapping of User & Kernel Thread : One to One

One-to-One

각각의 user thread를 kernel thread로 매핑

User thread가 생성되면, 그에 따른 kernel thread가 생성

Many-to-One 방법에서 문제가 되었던, system call 호출 시 다른 thread들이 block되는 문제를 해결

여러 개의 thread를 multiprocessor에서 동시적으로 수행 할 수 있음

한계점

Kernel thread도 한정된 자원이기 때문에 무한정으로 생성 할 수 없음

→ 다만 문제는 사용자 스레드 갯수가 많아지면 어떻게 할 것인가?

→ 어떤 프로세스는 여러 코어를 쓰는데, 하나는 하나의 core만을 사용하는 것이다.

같은 프로세스라도 자원을 더 많이 쓰게 되는 것이다.

이는 스케줄링 관점에서 나쁜일이다.

Thread를 생성, 사용하려 할 때 그 개수에 대한 고려가 필요

사용자가 무한하게 커널 thread를 만들 수 있다면, 그것은 한 프로세스가 시스템 자원을 다 써버리는 것도 가능하다는 의미이다.

One to one 예시

사용자가 생성가능한 thread를 제한하는 것 또한 방법이기도 하다.

Mapping of User & Kernel Thread : Many to Many

Many-to-Many

여러 개의 user-thread를 여러 개의 kernel thread로 매핑

Many-to-One과 One-to-One model의 문제점을 해결

Kernel thread는 생성된 user thread와 같거나 적은 수 만큼만 생성이 되어 적절히 스케줄링함

One-to-One처럼 사용할 thread의 수에 대해 고민할 필요 없음

Many-to-One처럼 thread가 system call을 사용할 경우, 다른 thread들이 block되는 현상에 대해 걱정할 필요 없음

커널은 적절히 user thread와 kernel thread 매핑을 조절하여 위와 같은 장점을 보장할 수 있음

유져 thread가 많아져도 커널 thread의 갯수는 한정됨.

Many to many 예시

Thread로 인한 운영체제의 변화

프로세스 기반의 운영체제 시스템 콜

프로세스 관련 시스템 콜 semantics는 프로세스 기준으로 작성됨

Thread의 개념에 대한 고려가 필요

fork(), exec()

Thread 지원을 위해서는 변화가 필요

→ Thread를 얼마나 복사해야하나?

Thread 종료 문제

Multi-thread일 경우 함께 일하는 thread의 종료는 프로세스보다 복잡해짐

- Thread cancellation issue

→ 그 address space를 어떻게 삭제하는가?

→ 전부 semantics문제이다.

Multi-threaded programming에 대한 지원

Thread 스케줄링

Thread 간 통신 방법

Thread가 사용할 메모리 공간의 할당

Stack, thread specific data

→ 다 문제가 된다.

Threading Issues: Creation

Multithreaded program에서의 fork와 exec의 의미는 달라져야 함

fork

- 하나의 프로그램 내의 thread가 fork를 호출하면 모든 thread를 가지 고 있는 프로세스를 만들 것인지 아니면 fork를 요청한 thread만을 복사한 프로세스를 만들 것인가의 문제
  - Linux에서는 2가지 버전의 fork를 만들어 각각의 경우를 처리하도록 함

exec

- fork를 하여 모든 thread를 복사 했을 경우 exec을 수행하면 모든 thread 들은 새로운 program으로 교체가 됨
  - 교체될 thread들의 복사는 불필요한 작업
- fork를 하고 exec을 수행할 경우 fork를 요청한 thread만이 복사 되 는 것이 더 바람직함
- 그러나 fork를 하고 exec을 수행하지 않는 경우에는 모든 thread의 복사가 필요하기도 함

1:23:49