1️⃣ OS_이론

▼ 2022.08.31 (수)

컴퓨터 시스템을 위한 큰 그림

- CPU(계산을 위한 register), Memory(program code) - Memory에 들어있는 CPU를 가져와서 저장
- 명령어들은 Memory안에 들어있다.
- process라고 불리는 CPU만의 register를 이용하여 계산하고 결과를 Memory에 저장한다.

Key Concepts in OS

[컴퓨터 구조]

폰 노이만 컴퓨터

- Stored Program - 내가 하고 싶은 일을 컴퓨터가 할 수 있도록 구조를 만들어 놓은 것
- Sequential execution - 순차적으로 앞에서 하나씩 실행하는 것 - 내가 어디서 실행하였는지 알아야 한다. (Program counter, Instruction pointer - CPU안에 들어있음)
- Memory(단위: B, KB, MB, GB, TB)는 CPU와 다른 장치, 다른 hardware로 연결된다.
- register는 CPU안에 들어있는 장치
- Address(위치를 나타냄-숫자 부여), Contents(bit 단위로 저장 - 8bit(1byte)), Code(instructor가 읽어옴)

OS (Multiple Programs execution)

- 원래는 software 분야이지만 hardware 관련 issue를 다룬다.
- <하나의 program>

1. Process - CPU(=processor) execution(하나의 program 실행하는 것)을 추상화 한 것
2. Address Space - Memory에 대응되는 것
3. I/O(Dual mode execution)
    - Input/Output으로 buffer로부터 data를 넣고 빼는 것을 의미한다.
    - 사용자가 실제 실행을 하려면 입출력을 받아야한다. - 모든 I/O는 OS에게 요청한다.
- user program 이 직접 실행할 수 없다. 다른 여러 program 실행 중 - OS에게 실행 요청하여 자신의 범위안에서 hardware를 쓰도록 허락한다.
- user mode(일반 program 실행) - user program 실행하는 것 - Memory 연산 (+,-,*,/), Memory에 있는 값을 CPU에 가져오기, CPU에 있는 값을 Memory에 저장하기, jump
- kernel mode(privileged mode) in CPU : 나머지 일은 OS에 요청한다.
    - <도움 요청방법>
        - (Operating) System Call , Interrupt 발생할 때 하던 일 멈추고 봐주고 다시 하던 일로 돌아온다.
        - Exeception(예외적으로 CPU가 할 수 없는 일을 하라고 할 때 ex)0을 나누어라.)
        - 유효하지 않은 Memory에 excess하라할 때 NULL pointer 참조한다.
    - entry - user mode하다가 kernal mode로 들어오는 입구
4. protection - 각각의 program들이 서로 침범하지 못하게 OS가 중재한다.

Q. 여러개의 program을 하나의 computer로 어떻게 실행하는가?

A. Scheduling - 어떤 program을 선택해 실행할 것인지 정하는 것

- CPU Scheduling (평가 방법: Waiting time, Fearness)
- FIFO (처음 들어가는 것이 제일 처음 시행되는 것)
    - 앞에서부터 실행하여 앞의 것이 오래 실행되면 뒤의 것이 기다려야된다. 기다리는 시간이 오래걸림(오래 기다린 시간+앞에 것이 끝나는 시간), 순서에 따라 운이 없게 오래걸리는 시간이 앞에 걸리면 전체 시간이 너무 오래걸리게 된다.
- Shortest Job(가장 짧게 걸리는 것) - 알고 있는 것 가장 앞에 배치하는 것이 최선
- RR(Round Roading) - 실행이 안 끝나고 중간에 끊어서 중간에 조금씩 시간을 나누어 쓰는 것 - 여러명이 거의 동시에 실행되는 것처럼 보이게 한다.
    - Waiting time으로 보면 RR은 최적이 아님 - 가장 느린 Sceduling 방법
    - [장점] - 비교적 공평하게 CPU 사용 가능
    - [RR scheduling 방법]
        - 컴퓨터가 얼마나 시간이 지났는지 알아야함 - Timer
        - Time interrupt - CPU, OS가 time이 얼마나 지났는지 알 수 있음 - CPU에게 timer 직접 들어옴
        - Time slice(quantam) - 시간을 나눠서 쓰고 누적시켜 다 찬다면 다음 process로 넘어감
        - eventually 공평하게 scheduling 되는 방법
            - Context Switching(process switching) - 실행되는 program 바꿔줌.
- SJF(최소 작업 우선 스케줄링) - 각 processor의 실행 시간을 이용하여 processor가 사용 가능할 때 실행 시간이 가장 짧은 작업에 할당
- MLQ
- Fairness : 공정성이란 모든 스레드가 자신의 작업을 수행할 기회를 공평하게 갖는 것
- RR scheduling 방법
    - Timer : 컴퓨터가 얼마나 시간이 지났는지 알아야함
    - Timer interrupt : CPU, OS가 time이 얼마나 지났는지 알 수 있음 → CPU에게 timer 직접 들어옴
    - Time slice(quantam) : 시간을 나눠서 쓰고 누적시켜 다 차면 다음 process로 넘어감
    - eventually(공평하게) scheduling 되는 방법
    - Context Switching(process switching) : 실행되는 program 바꿔줌

- Concurrency & Synchronization (동기화, 병렬성)
    - RR schedulung 방법 : Context Switching(process switching)
    - [문제 발생] (race condition, deadlock, starvation)
        - 어디에서 context switching 발생하는지 알 수 없음 → 여러 프로그램이 공유되는 메모리가 있는 경우 발생
        - concurrency 병렬성/동시성(동시 여러 개 실행) - 멀티 코어, 멀티 CPU
    - [문제 해결] : sychronization (동기화 기법 사용)

      1. Software Solution-Peterson’s algorithm (2 Threads solution)

      ex) withdraw, deposit : 둘 중에 하나만 실행하도록 하고 싶음 → flag를 두고 0이면 deposit, 1이면 withdraw 실행

      → 내 turn 지정한 다음 while문은 만약 상대가 먼저 turn을 지정하였다면 기다림

      → 내 turn을 flag를 두어 내 turn이라고 선언을 한다 → 상대의 turn이고 상대가 원하는 flag를 들었으면 나는 들어가지 않고 기다림

      (turn 변수 저장 직전에 scheduled out 되면 문제가 생김 → 해결:상대의 turn 지정(내 turn 선언 x)

      2. Hardware Solution (2개 이상의 thread)
        - test-and-set CPU instruction (읽기쓰기 동시에 하게 함 → 최종 결과가 업데이트되기 직전에 context switching되는 상황 없애버림)

→ 메모리에 있는 값을 register에 가져옴 → 메모리에 값을 씀

→ test(read memory) : 원래 있던 값 가져옴(=Road)

→ set(store memory) : 이전에 무슨 값이든 상관없이 1로 설정 → test&set 두가지 한꺼번에 실행

→ 초기에 메모리는 0값 갖음 → 나머지 모든 사람들 1가짐

→ mutual execlusion 첫번째 들어오는 사람만 1 나머지 0 : critical section에 하나만 나머지는 들어오지 못하게 가능

3. Critical section : 한번에 한명만 critical section에 접근하도록 하는
4. Spin-lock : CPU spinning in loop → 비싸지만 빨리 해결 가능
5. Mutex : 다른 애가 critical section이 끝나야 들어갈 수 있기 때문에 CPU에게 다른 작업을 하도록 하고 (CPU sleep)한 상태 → 나는 기다림(waiting 상태로 만듦) : waiting in sleep sate → 기다리는 시간이 더 길어질 수 있음

Virtual Memory

- 프로그램이 돌아가는 메모리 주소와 실제 메모리 주소를 구분함
- CPU 물리 메모리 상에서 전혀 동작 X → MMU키면 CPU가 보낸 주소 무조건 가상 주소
- B&B : Base주소 제한하고 Bound주소 설정(CPU안에 두개의 Register(시작하는 끝내는))
    - 목적 : Logical(Virtual)한 메모리, 내가 access할 수 있는 메모리 제공, process마다 독립적인 공간인 메모리 가질 수 있음.
    - 물리적인 메모리 x, 논리적인 virtual 메모리 사용
    - base bound 지정 불편한 점 : fragmentation(단편화)
    - mapping하는 이유
        - 메모리를 효과적으로 쓰려고, 아무리 손해가 커도 4KB내에 손해밖에 없음 (자신의 logical한 address space안에서)
        - sharing에도 쓰임 → 어떤 process가 다른 process와 데이터를 공유해야 할 때(통째로 mapping하여 넘김) → 동기화 문제 발생 → 동기화 메커니즘 필요
- Virtualization
    - Multiplexing : 물리적으론 하나지만 논리적으로는 메모리를 펼쳐놔 process마다 내가 사용하는 메모리가 다 있는 것처럼 하는 것
    - 입력의 여러 개 중 그 중 하나 골라서 씀, mapping을 이를 이용하여 사용
    - 메모리 주소가 같은 주소를 사용하더라도 물리적으로는 다른 주소를 나타내야함(mapping 바뀜 → 1번 process, 2번 process)
    - 메모리 주소 mapping하는 overhead가 대부분의 overhead 차지

OS가 하는 역할

- 전부 가상메모리에서 동작해도 문제가 없음
- 물리 메모리와 가상 메모리의 mapping
- Paging
    - 논리적으로 정의된 주소공간과 실제공간을 mapping하는 것 : 주소들이 모여있는 공간 → 전체 주소(address space)를 4KB(2^12)(mapping단위)로 쪼갬
    - 조각조각에 대해 번호 붙임 → page number, 실제 DRAM의 조각조각을 4KB로 쪼갬 (page frame)
    - page table(mapping table) → 실제 mapping하는 table
        - process마다 여기저기 메모리안에 들어있음
        - 프로그램 시작위치를 CPU안에서 지정할 수 있음
        - TTBR register(현재 시작하는 process의 mapping table 가리킴) : 멈췄던 곳에서 재개하도록함 (돌아가는 process에 mapping하도록 access함)
        - Mapping table memory(물리 memory)에 page table 내용 들어가있음 → 시작 주소를 CPU가 가리키고 있음
        - 사이즈 : entry 하나가 4KB(32bit) CPU의 경우 : 전체 address space 32bit → 32bit/4KB=4GB(2^32) → 4GB/4KB=1M(1만개 있음)

== 1M*4bytes=4MB개의 table size → 너무 많아~~~~~~~~~

- page number에 대한 index가 없더라도 pfn알 수 있음
- physical memory에 바로 접근 못함 → 실제로 메모리 접근 2번 일어남
- observation(solution) : 4KB mapping하려면 어떻게 해야하나? → 더 크게 4GB로 mapping하려면 entry 1개만 필요

→ 1MB로 mapping하게 되면 전체 주소가 4GB이므로 4GB/1M=4K(4000개)만 있으면 됨 (백만개 —> 사천개로 줄어듬)

→ 쓸데없는 공간, kernal-library 영역 큰 단위로 mapping할 때 사용

- Two-level paging
  - page table size때문에 사용
  - 큰 단위에 mapping(4MB로 가정 → entry 1000개 필요)하고 작은 단위 mapping(first level→second level)

    →둘다 하고 싶으면? 큰 단위 찾고 더 작은거 하고 싶다면 더 작은 단위에 entry 설정 (더 작은 단위에 table의 시작주소를 갖도록 함)
  - 큰 mapping(4MB)을 하는 entry안에 작은 mapping(4KB)을 하는 entry pointer 들어있음
  - translation : 주소를 세 부분으로 나눔
- Disk Swapping(바꾸다)
  1. disk와 memory에 들어있는 내용 교체 → memory에서 부족한 용량을 disk에서 빌려쓸 수 있도록 해줌
  - process를 실행하려면 execute함수 실행시켜야함 → 할 수 있는 방법 : Loading 파일의 정보를 memory에 올리는 방법
  - address space 만들기 시작 : 원하는 데이터 올릴 수 있음
  - 모든 process를 감당하기에는 부족하므로 파일에 있어도 되는 코드와 데이터 영역을 일부러 가져오지 않음 → 메모리 실제 사용량 줄임
  - 중간에 page table있음 → page table바꾸며 mapping도 바꿈
  - 지금 CPU가 access하려면 일단 memory에 꼭 있어야함 → 사용자의 주소공간에서 사용하려면 page table있어야함
  2. disk에 데이터 저장하여 필요할 때 가져다 쓰는 방법
  - 어떤 process의 어떤 page에 가면 쓸 수 있는지 (OS가 담당) → page table과 비슷
  - 소프트웨어로 자신의 메모리 영역을 어디서 찾을 수 있는지
  - 현재 memory면 바로 access 아니라면? 표시해놓은 CPU가 바로 알 수 있어 OS에게 알려줌

    → disk에서 가져옴, 가져와서 저장할 수 있는 page memory 생성 후 이곳에 저장하여 mapping
  - valid한 mapping이 없다면? page fault해줌
  - evict(쫓아냄) : 최근에 제일 적게 참조된 것 (오래전에) 내보냄
  - LRU, working set(같이 access되는 page있음) → 통째로 regression, swapping
  - Dirty bit : 원래 있는 값 저장해야함
- Thrashing
  - 컴퓨터 느려지는 상황
  - swap 계속 일어나는, 메모리를 full로 사용하여
  - disk에서 가져와야함 빈공간 필요, 빈공간 만들고 저장함 → 성능 느려짐
  - 실제 동기화 전혀 진행되지 않음
  - 미리 위험수위 정해놓고 미리 swap out하도록 관리
- TLB : SRAM 메모리 → 테이블 내용을 한꺼번에 메모리까지 오지않고 변환할 수 있도록 되어있음

▼ 2022.09.01 (목)

Cache

- Caching (빠르게 access할 수 있음)
  - 메모리에서 가져오면 6나노세컨 걸림
  - CPU안에도 데이터 저장기능있음 → 자주 access되는 것들
  - 자주 access되는 것들을 가져오는것
  - CPU뿐만 아니라 여러 level에서 가능하다 → OS에서 하고있음
  - 성능을 올리려면 속도 차이가 나야함 → Caching 효과 good
  - processing, CPU근처에서 데이터를 빠르게 access하고 싶음
- Cache hit/miss
  - 데이터를 cash해서 찾을 수 있으면 hit, 못 찾으면 miss
  - hit rate가 높다면 hit 시키기 쉽다 (hit가 잘 나게 만들기 쉽다) → cache쓰는 이유
  - locality : 한 지역에 모여있는 것
    - 메모리 주소 근방에 있는 것을 access할 가능성이 높음
    - 메모리 주소를 펼쳐놓고 봤을 때 근방에 있는 것을 access할 가능성이 먼 것들보다 높음
  - 원하는 데이터가 cache에 있는지 찾아봄 → hit하면 바로 가져오고 → 없다면 memory에서 가져옴
  - [걸리는 시간] : Average Memory Access Latecy (MAT)= Cache Access Latency * (Hit P+miss P)+Memory Access Latency*(Miss P)
    - Hit P와 Miss P는 0아니면 1, Hit P + Miss P=1이므로 곱하면 없어짐
    - Cache Access Latency:1ns, Memory Access Latency: 70ns
    - Miss P :50% → MAT=36, Hit P :20% → MAT=57, Miss P :80% → MAT=57
- Cache
  - 데이터를 CPU안에 들고있는 것
  - translation의 결과(물리주소를 가지고 데이터 access, 직접 가지않고 cache에 있는지 먼저 가도록)를 가지고 있는 것
  - DMA(direct memory access) : 보통 CPU가 하지만 네트워크나 하드디스크같은 장치들이 CPU를 거치지 않고 바로 들어가는 것
  - cache가 들어가는 경우 : CPY가 가지고 있는 것이 cache인데 데이터가 cache에 있는데 메모리 주소와 맞지 안으면 메모리 주소가 우선 동기화를 해야함(OS가 맞춰줌)
  - bit field : entry에 cache에 가능한 영역인지 구분

Concepts around Cache and File system

** Memory access with Cache (Cash, $)

- Locality(spatial, temporal) : 근처에 있는 애들이 access될 확률이 높다
- Cacheline, hit/miss : miss가 되면 메모리까지 갔다와야함 → access와 hit 많을수록 gain 커진다
- Memory Access Time : 수십 나노세컨, cache access time : 수백 picosec
- SSD time : 수십 ns, HDD : 십 ms

→ 속도 차이, hit율에 따라 효과가 있는지 결정됨

- Page table translation 결과를 caching
- CPU에서 memory access
  - VA → TLB → PA(hit) 0 → page table lookup(table walk)
  - PA → cache → memory

** File System

- directory 구조, file 구조 (meta-data)

- 파일 이름, 파일 위치 : meta-data(access를 하기 위해 추가적으로 필요한 데이터 설명하는 데이터)
- 실제 파일 access하는 방법
  - 파일이름을 가지고 file number를 찾는 → directory structure
  - file number를 가지고 storage block을 찾는 → file structure
- pathname → directory structure → file number
  - 사용자를 위해 제공되는 structure
- file number → index structure → storage location
  - OS가 인식하는 파일, 저장소 내의 위치 정보 기억
- FAT (file allocation table)
  - Linked List(연결리스트, 자료구조) 기반의 index structure
  - block과 1:1로 매치되는 FAT 테이블이 file system disk 제일 앞에 위치
  - [단점]
    - sequential read와 write해도 sequential하게 일어나지 않는다 → 완전히 쪼개져있음
    - free list 떼오는 건 쉬움, 한번에 쭉 연결된 데이터를 쓰고싶을 때 쓰기 불편
    - fitting argorithm을 돌리지만 효율이 좋지 않음 → 작은 file에 대해 두번의 access필요
    - index와 똑같은 number쓰므로 실제 access number 망가지는 문제가 있음
- UNIX file system (ufs)
  - 작은 크기 파일 빠르게 접근 가능하도록 함 (direct pointers)
  - 큰 파일 지원 (indirect, double/trip-indirect block)

** Directory

- directory안에 file이 들어있음 (file 목록)
- directory가 파일이기 때문에 directory안에 directory있을 수 있음, 그 안에 파일이름들 나옴
- directory entry : 파일과 directory로 구성가능 (directory 파일인 경우에 하위 계층의 directory 생성 가능)

**File

- 저장소의 이름을 붙인 것
- 데이터와 메타데이터 들어있음 (파일이름이 일종의 meta-data)
- FAT에서 free는 안쓰는 영역 → 새로운 것 추가해서 한블럭이상이 넘어간다면 free에서 할당받아 새로운 block생성
- format한다면? FAT 내용만 지우면 파일이 어디있는지 알기 어렵다 → 데이터가 있기 때문에 못찾는건 아님
- Link File
  - 이름은 다르지만 똑같은 내용의 파일 만듦 (=copy)
  - 어디에 저장되어있는지 똑같이 기억하는
  - index structure를 똑같이 만드는 것
  - 이름은 다르지만 번호가 똑같은 = 아예 똑같은 index structure
- FFS
  - [장점] : 큰 파일 작은 파일 모두 지원, 빠르게 찾을 수 있음
  - [단점] : 굉장히 작은 파일을 만들기 위해 아이노드 데이터파일 만드는 것은 overhead가 큼
  - prespace영역으로 10%~20% 영역을 잡고 있음
- NTFS : extents(block point가 여러 개의 pointer를 나타내게 되어있는)
- Read file_1

    - find root file number(in superblock) → 2
    - root file read → disk datablock 0,1,2,3
    - fine file_1 named file's Number → 3
    - file_1's inode read → disk datablock → 4
    - read the datablock → 0X3000

- Copy-on-Write
    - update한 데이터 빠르게 사용 가능
    - 예전 데이터 사용해도됨
    - kernel이 사용하는 영역은 P1,P2,Process 3 똑같은 주소공간 사용
- MMap : 내가 읽을 file의 data와 메모리 주소공간을 mapping해줌
- page cache : 전에 썼던 데이터를 지우지 않고 그대로 사용하도록 놔두는 것 → 성능 더 좋음
- buffer cache : 사용자가 직접 대하는 데이터와 meta-data구분
- I/O장치
    - 보통 CPU와 비동기적으로 동작
    - 네트워크카드와 CPU는 비동기적으로 동작
    - I/O와 관련된 CPU있음 / 입출력하는 instruction있음 / 고성능 과정에는 쓸 수 없음 → CPU가 돌아가는 시간 빠르기 때문에 중간중간 CPU가 들어가서

      ex) GPU : Mmap 관련된 장치 → 비동기적으로 연산 날릴 수 있음 → 매번 말하지 않고 장치가 일할 수 있을 때 묶어서 보냄 → 장치가 알아서 해야할 때 동작하도록

▼ 2022.09.02 (금)

- I/O and System performance
    - Queing model : performance 분석할 때 엔지니어들이 많이 쓰는 모델
- Internet & Distributed Systems
    - addressing : address 주는 방식 중요
    - routing : 경로 만듦 → 알고리즘
- Congestion Control of TCP
    - 원래 보내는 양의 절반으로 줄임
    - congestion이 발생하는 일을 미리 피하도록 → congestion avoidance
    - 조금씩 보내는 양을 늘린다
- Cloud Computing service model : As-a-service
    - 컴퓨터를 쓸 수 있는
    - (하드웨어 설정, 소웨 설정 개발해야함) 아무것도 하지 않은 → virtual → IAAS
    - 조금 더 추가하여 (돈을 더 내어) 쓸 수 있는 → PAAS
    - 더 돈을 내면 software 얹어줌 → SAAS

공정성이란 모든 스레드가 자신의 작업을 수행할 기회를 공평하게 갖는 것

CPU scheduling

- 현재 실행상태에 있던 process가 다른 상태로 천이될 때 다음 실행시킬 process의 선정이 필요한 경우 scheduling발생
- 성능평가 기준

    - throughput(처리율) : 단위 시간당 완료되는 process 수
    - turnaround time (반환 시간) : process가 생성되어 작업을 마치고 종료될 때까지 걸리는 시간
    - waiting time(대기 시간) : process가 생성되고 작업을 마치고 종료될 때까지 que에서 기다리는 시간
    - response time(반응 시간) : 대화형 system에서 system이 반응을 시작하는 데까지 걸리는 시간
- 비선점 스케줄링
    - FCFS
        - process의 도착순으로 CPU를 배정
        - 장기 scheduler에서 사용 가능
        - 시분할 system에서는 time slice를 적용하지 않는 process에 대해 사용 가능
        - 수행 중인 긴 작업을 여러 개의 짧은 작업들이 기다리게 되는 호위 효과의 문제 발생
    - SJF
        - 대기하는 작업 중 CPU burst time이 가장 작은 작업에 cpu를 할당하는 방법
        - 평균 waiting time에 있어 가장 최적의 알고리즘
        - cpu 요구시간을 미리 알기 어렵다. 이전 burst time의 지수적 평균으로 간주하여 예측
    - 우선순위 스케줄링 (priority scheduling)
        - 우선순위가 제일 높은 process에게 CPU 할당하되 우선순위가 같은 경우 FCFS 적용
        - 연산 위주 process보다 입출력 위주 process에게 더 높은 우선순위를 부여하여 대화성 증진
        - 우선순위가 높은 작업이 계속 들어오면 낮은 작업은 준비상태에 머물러 있음 (starvation) → system에 머무는 시간이 증가할수록 우선순위 높여주는 Aging으로 해결

- 선점 scheduling
    - SRT scheduling
        - 최단 잔여시간을 우선으로 하는
        - 진행 중인 process있어도 sleep 시키고 짧은 process 먼저 할당
        - 선점형 SJF scheduling
    - RR(round robin) scheduling
        - 시분할 시스템을 위해 설정
        - 준비 que를 원형 que로 간주하고 순환식으로 각 process에게 작은 단위의 시간량 (time quantam)만큼씩 CPU를 할당하는 방식(time quantam=time slice)
        - 알고리즘 성능 → 시간 할당량의 크기에 좌우됨

        → 시간 할당량 크면 FCFS와 같고 매우 작으면 process 공유와 같아짐
        - 일반적으로 평균 waiting time이 SJF보다 크지만 process가 공정하게 기회를 얻게 되어 (fairness) 기아 상태 (starvation)은 발생하지 않음
        - time quantam의 크기가 작으면 잦은 문맥 교환 overhead의 증가로 처리율(throughput)이 감소할 수 있다.
- MLQ(다단계 큐 스케줄링)
    - 우선순위마다 준비 que 생성
    - 항상 가장 높은 우선순위 que의 process에 CPU할당(선점형 scheduling)
    - 각 que는 rr이나 FCFS등 독자적 scheduling 사용 가능
    - starvation발생
- MFQ(다단계 피드백 큐 스케줄링)
    - 우선순위 변화 적용

    - process 생성시 가장 높은 우선 순위 준비 que에 등록되며 등록된 process는 FCFS 순서로 CPU를 할당받아 실행 해당 que의 cpu 시간 할당량이 끝나면 한 단계 아래 que가 준비한다
    - 단계가 내려갈수록 시간 할당량 증가
- MLQ & MFQ
    - MLQ는 que와 que 사이에 process들이 이동할 수 없는 반면 MFQ는 que 사이에 process들 이동 가능

Concurrency & Synchronization

- Spin lock
    - 다른 thread가 lock을 소유한다면 그 lock이 반환될때까지 무한 loop를 돌면서 최대한 다른 thread에게 CPU를 양보하지 않는것
    - 특정한 자원을 획득(lock)또는 해제(unlock)하여 공유자원에 대한 접근 권한을 관리하는 방식

→ 하나의 CPU나 코어에는 유리하지 않다, busy waiting을 하여 CPU를 쓸데없이 낭비한다. → overhead

→ 권한 획득전까지 CPU는 무의미한 코드를 수행하는 busy waiting 중

→ context switching 줄여 효율을 높일 수 있다.

- critical section
    - user 객체로 커널에서는 제공하지 않는 객체이다. 가볍고 빠르다. 그러나 한 process내의 thread사이에서만 동기화가 가능하여 가볍고 쉽게 쓸 수 있는 동기화 객체이다.
    - process간의 공유 자원을 접근하는데 있어서 문제가 발생하지 않도록 한번에 하나의 process만 이용하여 다른 process들의 접근을 제한하는 영역

    1. 상호배제 : 하나의 process 들어가있다면 다른 process들어갈 수 없음
    2. 진행 : 들어가려는 process가 여러개라면 어느 것이 들어갈 것인지 결정
    3. 한정된 대기 : 다른 process의 starvation을 막기 위해 critical section에 들어간 process는 다음 process section 접근 제한이 생겨야 한다.

- mutex
    - 커널 객체로 critical section보다 무겁다. critical section 이 한 process내의 thread 사이에서만 동기화가 가능한 반면, mutex는 여러 process의 thread 사이에서 동기화 가능하다. 흔히 사용하는 예가 process 다중 실행을 막을 때이다. cs를 가진 thread끼리 실행이 겹치지 않게 단독으로 실행하는 기술이다.
    - lock 또는 unlock 상태가 있으며 spin lock과 같이 접근권한을 획득할 때까지 busy waiting 상태에 머무르지 않고 sleep 상태로 들어가며 wakeup 상태가 되면 권한 획득을 시도
    - 오직 하나의 thread만이 동일 시점에 mutex를 얻어 critical section에 접근 가능

- semiphore
    - 커널 객체로 critical section과 mutex는 동기화함에 있어 동시에 하나의 thread만 실행되게 하지만 semaphore는 지정된 수만큼의 thread가 동시에 실행되도록 할 수 있다. 지정된 수보다 작거나, 같을 때까지 thread의 실행을 허용하고 지정된 수를 넘어서 thread가 실행하려고 하면 막는다.
    - 하나 이상의 thread가 공유자원에 접근하도록 할 수 있다. 특정 자원에 접근 할때 semWait가 먼저 호출되어 critical section에 들어갈 수 있는지 확인 후 빠져나와 critical section에 들어가게 되고 이 후 semsignal이 호출되어 빠져나온다.

- dead lock(교착 상태)
    - 각 process의 첫째줄이 동시에 실행될 경우 서로 wait상태로 빠져 다른 쪽에서 signal을 줄때까지 기다리게 되는 상태

- mutex & semaphore
    - 세마포어는 여러 개의 thread에 접근할 수 있지만 mutex는 하나의 thread에서만 접근이 가능

- 세마포어는 현재 수행 중인 thread가 아닌 다른 thread가 세마포어를 해제할 수 있지만 mutex는 획득하고 해제하는 주체가 동일해야 한다.

Process & Thread

- program
  - 파일이 저장 장치에 저장되어있지만 메모리에는 올라가 있지 않은 정적인 상태, 어떠한 작업을 위해 실행할 수 있는 파일
- process
  - 운영체제에서 프로세스는 일련의 작업 단위입니다. 프로그램은 파일이 저장장치에 저장되어 있지만 메모리에는 올라가 있지 않은 정적인 상태이며, 이러한 프로그램을 실행시켜 운영 체제로부터 CPU를 할당받고 실행되고 있는 상태를 바로 프로세스라고 합니다.
- 즉, 프로세스는 실행 중인 프로그램
  을 의미하며, 프로그램이 실행 중이라는 것은 디스크에 저장되어있던 프로그램을 메모리에 저장한 뒤 운영체제의 CPU 제어권을 받을 수 있는 상태가 된 것을 의미
  - OS로부터 시스템 자원을 할당받는 작업의 단위
  - 메모리에 올라와 실행되고 있는 program instance
  - 컴퓨터에서 연속으로 실행되고 있는 program
  - 각 process는 다른 process의 자원이 접근하려면 process 간 통신 필요
  - 할당받는 system 자원 → code, data, stack, heap구조로 되어있는 독립된 memory 영역

- thread
  - process내에서 실행되는 흐름의 단위로, process 하나에 자원을 공유하면서 일련의 과정을 여러 개 동시에 실행시킬 수 있는 것
  - 하나의 process 내에 주소 공간이나 자원들 공유
  - 하나의 process가 생성되면 하나의 thread가 생성됨
  - 하나의 process는 여러 개의 thread를 가질 수 있음
  - thread는 process내에 tack만 따로 할당받고 code, data, heap영역은 공유됨
- multi process & multi thread
  - multi process : 하나의 응용 프로그램을 여러 개의 process로 구성하여 각 process가 하나의 작업을 처리하도록 하는 것
    - 여러 개의 자식 process 중 하나에 문제가 발생해도 다른 자식 process에 영향이 안감
    - 각 process들이 독립적으로 동작하기 때문에 안정적
    - multi thread보다 더 많은 메모리 공간과 cpu시간 차지
  - multi thread : 하나의 응용 프로그램을 여러 개의 thread로 구성하여 각 thread가 하나의 작업을 처리하도록 하는 것
    - system자원의 처리 비용 감소
    - context switching 빠름 (thread는 stack영역만 처리)
    - thread간의 데이터 공유시 동기화 발생
    - 너무 많은 thread 사용은 overhead발생

    [multi process보다 multi thread 사용하는 이유]
    - process 생성 시 syscall 줄어들어 자원소모 적고 자원을 효율적으로 처리 가능
    - process의 경우 context switching시 메모리가 초기화되기 때문에 overhead가 큰 반면, thread는 stack 영역만 처리하면 되므로 문맥 교환 속도가 빠르다.
    - thread간의 자원 공유 간단하기 때문에 system자원소모 적다.

    but, 전역 변수를 통해 thread간의 자원을 공유하기 때문에 동기화 문제 잘 해결해야 함

▼ 운영체제 구성요소

커널(Kernel)

프로세스 관리, 메모리 관리, 저장장치 관리와 같은 운영체제의 핵심적인 기능을 모아놓은 부분입니다. 커널의 주요 기능은 컴퓨터에 속한 자원들에 대한 접근을 중재하는 것인데, 구체적으로는 입출력 관리, 자원이 필요한 프로세스에 메모리 할당, 프로세스, 메모리 제어, 프로세스 간의 통신, 파일 시스템 관리, 시스템 콜과 같은 역할을 운영체제 맨 하부에서 수행합니다.

커널 종류

- 단일형 구조(MS-DOS, VMS)
- 계층형 구조(Windows 등 오늘날의 대부분의 운영체제)
- 마이크로 구조(애플의 OS X, IOS)

인터페이스(Interface)

사용자의 명령을 컴퓨터에 전달하고 결과를 사용자에게 알려주는 소통의 역할을 합니다. 운영체제가 제공하는 대표적인 인터페이스로는 GUI, CLI가 있습니다.

시스템 콜(System call)

사용자나 프로그램이 직접적으로 컴퓨터 자원에 대한 접근하는 것을 막고 커널을 보호하기 위해 만든 인터페이스입니다. prinf(), write(), read()와 같이 사용자나 응용 프로그램이 컴퓨터 자원을 사용하기 위해서는 시스템 호출을 사용해야 합니다.

드라이버(Driver)

커널과 하드웨어의 인터페이스를 담당합니다. 운영체제가 각각의 하드웨어에 맞는 인터페이스를 개발하는 것을 비효율적이고 어렵기 때문에 드라이버를 통해 컴퓨터에서 하드웨어를 사용할 수 있게 하는 것이 바로 드라이버입니다. 마우스나 키보드와 같은 하드웨어는 꽂기만 해도 잘 작동이 되지만 복잡한 하드웨어의 경우 제작사가 만든 소프트웨어를 따로 설치해야 하는 경우가 있는데 이러한 소프트웨어를 디바이스 드라이버라고 합니다.

가상메모리

- 실제 메모리 크기와 관계없이 메모리를 사용할 수 있도록 가상 메모리 주소 사용
- process 일부만 메모리에 로드하고 나머지는 보조 기억 장치에 할당
- MMU를 통해 논리주소, 물리 주소로 나누어 사용
- 주기억장치의 실제적인 주소로 mapping하는 방법을 통해 구현

Paging

- process의 주소공간을 고정된 사이즈의 page 단위로 나누어 물리적 메모리에 불연속적으로 할당하는 방식
- memory는 frame이라는 고정 크기로 분할, process는 page
- page와 frame 대응시키는 page mapping 필요하며 paging table을 생성해야함
- page table에 각 page number와 해당 page가 할당된 frame의 시작 물리 주소 저장

Segmentation

- process를 서로 크기가 다른 논리적인 블록 단위인 segment로 분할하여 memory에 할당
- 각 segment는 연속적인 공간에 저장
- paging과 같게 mapping을 위한 segment table 필요