01 introduction to OS

Os는 왜 배워야 하는가 – 기술적, 비즈니스적 관점

어려운 이유 – 다양한 관점, 방대하다, 버그가 많다

트렌드 – 분산형 os, many core processors , connected computing (작은 디바이스에 os)

02 history

최초의 컴퓨터

주판 : 기계적 에너지

에니악 : 전기 에너지(전기적 컴퓨터의 효시)

*resident monitor : hw 와 sw를 구분하게 함. 여러 프로그램을 메모리에 올리고 실행순서 결정과 실행을 담당. 즉 sw가 sw를 관리 batch processing을 위해 등장함. OS의 효시였다는 점 중요

Batch system : 여러 개 작업을 묶어서 한번에 실행

Spooling : cpu작업과 io작업을 동시에 가능케 함

클라우드 컴퓨팅 : 대량의 컴퓨터, 분산형 os구조의 시작

Multicore computing : 멀티코어 cpu 등장, 이를 위한 os구조 바뀌는 계기

03 fundamental concpets

1. Abstraction : 복잡한 hw 추상화. hw쉽게 제어하도록 interface제공.

   System call , execution flow – *entry point : 응용 프로그램에는 보통 main하나, OS에는 다수의 entry

2. Protection

   예시 : base/limit register

   응용 프로그램으로부터 hw포함 os보호

3. Illusion : 시스템 물리적 제약 느끼지 못하도록 .

   Time sharing(cpu)

   Virtual memory(memory)

4. Coordination and optimization : 다수의 entitiy가 조화롭게 실행

   Multitasking(동기화) / multi-user / multithreading

04 Computer Hardware and OS

- 컴퓨터 구성 요소

  Processor, main memory, IO modules , system bus

  *폰노이만 모델 : stored program concept

- 프로그램 실행

- Interrupts

  Exception (synchronous 예측 가능) interrupt는 asynchronous, 예측 불가능

  Arm vector table

  IRQ , PIC

- dual mode execution

  Mode, context , space 개념 이해 중요

- System call vs Function call (address space 변화 유무)

05 Process

I. Process concepts

- process = address space(메모리) + current activity(CPU)

- PCB , process state

- VS program : 프로세스는 아니면서 명령을 실행하는 코드로 interrupt handler가 있고, 이는 process가 아님. 즉 스케줄링 대상이 아님

II. Process scheduling

- scheduling and dispatching

- context switching

System / memory / processor context 중 processor만 저장해주면 됨(레지스터 값들, PCB에 저장)

Multitasking – context switching 은 trade off 관계

- OS's queues : job, ready(ready state), device queue(waiting state)

Queue 안에선 PCB가 linked list 형태로 저장된다

III. Process creation and termination

- creation : scratch , cloning(fork)

- termination : normal/abnormal

IV. Cooperating process

- 장점
- bounded buffer 예제
- IPC (프로세스간 통신)

    구현 관점 : shared memory , message passing

    Link : logical, physical link

    Communication type : direct/indirect 동기/비동기적

- synchronization

    Blocking (synchronous) 전화

    Non-blocking (asynchronous) 문자

06 Threads

I. Thread 개념

- process와 비교
- 실행적 측면 성질

    Execution state(running, waiting, ready)

    Context switching한다

    Code and data영역은 공유

- thread api : pthread_create로 func을 concurrent하게 실행 가능
- thread private data(독자적 stack내부), thread local data(__thread)
- 이점

    응답성

    자원 공유, process보다 적은 자원

    경제성

    Multiprocessor 최대로 활용

    Concurrent programming model, 스레드로 동시작업(sequential X)

II. Multithreading models

- thread 구현 방법

    1) User thread 2) kernel thread

- many to one

- one to one

- many to many

III. Threading issues

- fork : 두 옵션 모두 제공

- other system call : data inconsistency

- stack overflow : 특히 user level thread

- thread pool

IV. Examples

- solaris 2 example (LWP)

07 CPU Scheduling

I. Preemptive / non-preemptive

II. Scheduling criteria

- cpu utilization

- throughput : 단위 시간당 처리 프로세스 개수

- turnaround time

- waiting time

- response time

III. Scheduling algorithm

A. Priority based scheduling

1) FCFS

2) SJF : optimality , feasibility

구현 불가능 하지만 임베디드의 경우 미리 실행시간 아는 경우, moving average로 실행시간 예측

-> starvation 문제와 aging기법

B. Round robin(RR)

(n-1)q 최대 기다리는 시간

SJF보다 turnaround는 크고, response time은 적게

Weighted rr

C. Multilevel queue

Priority와 rr을 혼합해서 사용

Multilevel feedback queue

08 synchronization

I. Why synchronization?

Bounded buffer 예제 : count 변수 추가. interleaving으로 인해 non deterministic하게 됨

II. Critical section problem

- Race condition
- 임계 영역의 세가지 조건

1) mutual exclusion

2) progress

3) bounded waiting

III. Synchronization algorithms

1) Flag

progress X(2개다) , mutual O , deadlock발생

2) Turn

progress X(1개만) , mutual O , better than 1)

3) Peterson's algorithm (flag + turn)

1) + 2) 조건 모두 충족

4) Bakery algorithm by Lamport

Bounded waiting X -> choosing으로 해결

조건 다 충족

IV. Synchronization with HW support

1) Interrupt disabling

커널 코드에서만, 즉 응용 프로그램에서 못씀

멀티 cpu에서 불가능

2) Test and set (intel)

Cpu가 직접 실행하는 hw 명령어

Load/store을 atomic하게 실행하도록

Bounded wating X -> bakery로 충족하도록 수정

3) Swap (arm)

어떤 cpu는 test-and-set대신 swap 제공

V. Synchronization with OS support

A. Semaphore

Wait , signal

atomicity , bounded waiting 보장 (나머지도)

구현 측면

1) Busy waiting semaphore(spin lock)

2) Blocking semaphore

진입 불가일 때 , running -> waiting으로 하고 , LL에 추가

진입 가능해지면, FIFO로 하나 뽑아 waiting->(ready)->running

사용 측면

1) Binary semaphore

2) Counting semaphore : 동일한 유형의 resource의 instance가 여러 개, 프로세스도 여러 개 일 때

General synchronization tool로도 사용

임계 영역 국한 된 게 아니라 concurrent 한 process의 실행도 제어. 폭 넓은 동기화. 예로는 프로세스 순서 강제할 때

유의할 점

Deadlock, starvation : 두 개 이상의 세마포어를 사용할 때 발생)

B. Mutex (binary semaphore)

Semaphore와 다른점

1) 소유권 개념

2) Thread간 동기화

Lock , unlock, trylock

VI. Classical synchronization problems

1) Readers-writers problem

a. First readers-writers problem

b. Second readers-writers problem

c. Third readers-writers problem

2) Dining-philosophers problem

a. 젓가락 개수보다 적은 사람 -> 비둘기집 원리 (낭비)

b. 첫번째 젓가락 lock, 두번째 젓가락 trylock (두개다 취할 수 있을 때만 취하도록)

c. Asymmetry solution

09 Deadlock

I. Reallocation graph

II. Conditions for deadlock(필요 조건)

1) Mutual exclusion : 오직 하나의 프로세스만 들어갈 수

2) Hold and wait

3) No preemption

4) Circular wait : 2)의 강화된 조건, 그 모습이 사이클 이룸

III. Handling Deadlock

1) Prevention

4가지 조건 중 하나라도 발생하지 않도록 -> 현실적 어려움

2) Avoidance : 사전에 p가 필요로 하는 resource유형, instance개수 알아야

Safe state , safe sequence

Safe state O : no deadlock

Safe state X : possibility

3) Detection and recovery

- detection

1. Single instance

Cycle 있으면 deadlock

Wait-for-graph 그려서 cycle 유무 판단

2. Several instances

cycle 있고, safe state아니면 deadlock

*감지 알고리즘을 언제 돌리는가

1) p가 진입 요청 할 때마다 : 오버헤드 크다. 비용 너무 큼,

2) 특정 조건 충족 시 , Ex > CPU utilization이 일정 수준 이하 , 상관관계는 있지만 인과관계는 적다

3) 주기적으로 : 그 주기 동안은 데드락 방치, 주기 짧게 하면 비용 커짐

- recovery

1. process termination(하나만 or 다)

2. resource preemption

4) Ignore : OS는 무시한다. 처리는 프로그래머의 책임임

10_1 Memory Management

I. Memory management

1) 프로세스 별로 하나의 단일, 연속적, 보호가 되는 공간 배정

2) 물리적으로 주어진 메모리보다 더 많은 양 메모리 사용가능 하도록 지원

- trick by OS

1) noncontiguous allocation with address translation

- Paging, segmentation

2) demand paging

II. Address binding : 프로세스에게 메모리 배정->P가 실행할 inst와 data 메모리 위치 정하는 것

1) Compile time : 컴파일 할 때

가상 메모리 없는 임베디드

2) Load time : 프로그램을 메모리에 로딩하는 시점

범용 컴퓨터

3) Execution time : 실행할 때

Dynamic library

III. Memory allocation

1) Contiguous allocation (external f 발생)

a. First-fit : 속도 가장 빠름

b. Best-fit : 공간 효율 좋음

c. Worst-fit : 속도 느리고, 효율 안 좋음

2) Non-contiguous allocation with address translation

- 구현 관점

a. Segmentation

b. Paging

- 이점

메모리 배정 유연성

메모리 보호, 공유 가능

- MMU : 주소 변환 HW (논리 -> 물리주소)

- Relocation register : 구현 어렵. 너무 많이 필요

IV. Fragmentation

1) External

2) Internal

10_2 Memory Management

I. Paging

- page(VAS) – frame mapping(PAS)
- page table : PTBR(시작 주소) PTLR(크기)
    1) hierarchical page table : two-level page table
    2) hashed page table : linked list 구현, 지금은 거의 사용 X
    3) inverted page table : #frame -> #page
        - 시스템에 하나만 존재. (1,2는 프로세스별)
        - 프로세스 구분 할 수 있는 정보 기록 (pid)
- TLB : 특수 HW 캐시, 한번의 메모리 접근만
    effective memory access time(EAT) 계산 . 성능 상 이득
    page number도 기록해야 함
- memory protection : valid/invalid bit
- memory sharing

II. Segmentation
- segmentation table : <segment-number , offset>
- limit : segment크기 표현
- base : #segment를 base로 교체

III. Examples (두개 혼합)
과거 segmentation -> 최근 paging
MULTICS
Intel

11_Demand_Paging

I. 초기 메모리 공간 활용 ? 현재는 ?

II. Optimization for better memory space utilization
1) Dynamic loading : OS가 필요할 때 로딩해줌
2) Overlays : 프로그램 자체가 동적 로딩
3) Dynamic linking : library , linking미룬다 . shared library에 구현
4) Swapping : HDD로 swap in/out

III. Demand paging

1) Swapping 과 비교(프로세스 통째로 , page단위로)

2) Page fault & handling (valid bit)

3) Performance(EAT) = (1-p)m + pf

P가 작을수록 성능 좋음(fault 발생 확률)

12 Page_replacement

I. Victim , reference string(page number sequence)

II. Page replacement algorithms

1) FIFO

Belady's Anomaly (number of frames – page faults 관계)

2) Optimal algorithm

Fault 수 최소화. 실제 구현은 못하고 다른 알고리즘 비교대상

3) LRU : optimal과 시간적 대칭

- Locality

a. Temporal : LRU는 여기 기반

b. Spatial

- 구현

a. Counter : 참조된 시각을 기억

b. Stack : 참조된 순서

-> 두 가지 다 실제 구현 어렵. 너무 빈번해서 속도 저하

4) LRU 근사 알고리즘

a. Reference bit

b. Additional-reference bits

주기 동안은 오차 발생

Bit 수 늘리거나 주기 줄이는 건 overhead큼

c. Second chance : FIFO + reference bit

다 1인 경우 FIFO와 동일해짐

5) Counting algorithms

a. LFU

b. MFU

13_Memory Allocation and thrashing

I. Memory Allocation

- frame 배정

1) fixed allocation

Proportional allocation(size에 비례해서)

2) priority allocation

작아도 신속하게 실행돼야 할 수도

- replacement

1) global replacement

모든 p들이 사용하던 frame중에서 선택

2) local replacement

Fault 발생시킨 p가 사용 중이던 frame중 victim 선택

II. Thrashing : demand paging 부작용

- 감지/처리 방법

1) working-set model

Working set window = 10

D > S -> thrashing

적절한 수의 p 중지

Working set 유지&파악

- reference bit(1) + in_memory bits(2) + interval timer

- 그 시점에 in_memory bit중 하나라도 1이면 포함

- 주기 동안은 오차

2) page-fault frequency scheme

p마다 page-fault 상한, 하한 지정

III. Optimization

1. 최적화 기법

- copy-on-write

Fork시, child에게 별도의 page제공X , 별도의 w행위하면 해당 page만 배정

예외) data영역은 0으로 초기화하여 별도로 제공

- memory-mapped files

HDD의 데이터 메모리에 올려서 메모리 상에서 R/W

HDD에 변경 내용 반영 전에 전원 꺼지면 손실. 중간에 sync작업 필요

(일반적으로는 close할 때 반영)

- modify(dirty) bit

Swap out할 때 변경 됐으면 HDD에 write

Page table에 추가

Swap out, HDD write 비용 감소

- prepaging

2. 고려할 이슈

1) page size

1. smaller

Page 당 I/O time 감소

Internal fragmentation 감소

Larger page table size (entry 개수 늘어서)

2. larger

Page당 I/O time 증가, 전체 I/O time은 감소

Internal fragmentation 증가

Smaller page table size

2) TLB reach

커질수록 hit rate 증가

키울수록 HW비용 증가

3) Belady's anomaly

OPT나 LRU같이 stack구현 가능한 알고리즘에선 발생 안함

FIFO에서 발생

14_1 IO Devices and operations

I. CPU execution and I/O

CPU burst time, I/O burst

Main job is I/O -> 성능 좌우

II. Character Devices : byte(char)단위로

1) Mouse

2) Terminal

Keyboard + display

III. Block devices : block(byte의 묶음) 단위

1) Disk drive

Seek + rotate + data latency

2) Flash drive

IV. Network devices : packet(byte의 stream)단위

가장 빠름

- NIC(network interface card)

V. Clocks and timers

- UNIX

Calendar time

process time(user + system)

Alarm() , ioctl

- Linux

Jiffies(부팅하면 0 , 주기적으로 ++)

14_2 IO Devices and Operations

I. I/O interfaces and operations

- I/O bus and connection

    1) address lines : 주소 값 연결

    2) data lines : transfer

    3) control lines : 제어

- I/O registers(IO controller / IO interface) – 전자적 동작

    1) Status : ready or busy

    2) control : cpu의 명령어 저장

    3) data-in (CPU 기준) : cpu로 가는 데이터

    4) data-out : cpu가 I/O에 쓰는 데이터

- I/O register access(CPU가 I/O register에 접근 방법 두가지)

    1) Memory-Mapped IO (MMIO)

    - memory address space상에 I/O 레지스터 매핑

    - same instruction

    - share data, address, control bus

    - 별도의 주소 decoding logic 필요

    - CPU가 쓰는 영역 reserve 필요

    2) Port-Mapped IO (PMIO / isolated I/O)

    - I/O address space따로

    - M / IO 구분하는 별도의 bus

    - different instructions(lw/sw , in/out)

    - control bus는 공유 안함

    - I/O 주소, memory 주소 동일한 경우, CPU명령어로 구분

II. Device drivers : 디바이스 별로 커널에서 5가지 함수 제공

- 접근 방식

    1) device file : character, block device의 경우

    2) socket API : network device

- device file : interface to a device driver

- block device driver 와 char device driver의 차이점

    1) cache : block은 대량 data를 R/W하므로 메모리에 캐싱 해놓음

    2) request queue : 시간 절약 위해 요청된 device 작업순서 재구성

- I/O data transfer(device 와 device driver이 정보 주고받는 방법)

    1) Polling (programmed I/O)

    2) Interrupts (Interrupt driven I/O)

    Handling (Ex : network interrupt example)

        -Top half : 시급하게 처리, disabled한 상태로

        Ex : 들어온 패킷 메모리에 시급하게 옮김

        - bottom half : 여유 있을 때 처리, enable한 상태로

        Ex : TCP/IP 코드는 나중에 처리

- Device driver 구현 방식

    1) blocking : ready 될 때까지 기다림

    2) non-blocking : R/W못하면 즉시 return

- read/write processing : (Polling/Interrupts) + (blocking/non-blocking) 조합 예시

- DMA

    Dma controller

    CPU 작업 부담 덜어주는 별도의 HW, detail한 I/O작업 처리 cpu작업 효율증가

    Cycle stealing문제점 : Mem-I/O bus를 cpu가 못쓰는 현상.

    -> '캐시'로 성능 저하 방지 bus거치는 경우 적다

15_1 Files and Directories(user관점)

I. File concept

A. Attribute

B. File Control Block : 실제 데이터 block위치 정보까지 담음

C. File types

D. File operations

Open , close

Open file table : open/close로 메모리에 파일의 meta data 가 저장되는 자료구조

1) Per-process 2) system-wide

E. Unix File : lseek함수

F. Access Methods

1) Sequential : file 내부

2) Random : 메모리 접근

G. Views on files and disk allocation

Logical record unit : file , 사용자 관점

Physical record unit : disk는 block단위로(low-level) , internal fragmentation

II. Directory structure

- partition : disk를 여러 개의 분할된 disk로 보이게 함

- 구조 변화

1) single-level dir

2) two-level dir : MFD , UFD

3) tree-structured dir

4) Acyclic-graph dir : 바로가기와 유사한 개념

Soft link / hard link

Aliasing problem

Dangling pointer problem

5) General graph dir

사이클 허용.

dir구조 탐색 시 loop에 빠지지 않도록 유의

- file system mounting : 제 3의 파일 시스템을 가져와서 mounting

16 File systems

I. File system structure : 4개의 계층 , 2개의 논리, 2개의 물리 계층

a. 계층 구조

- logical file system : FCB로 dir구조 관리, 물리적 HDD와 독립적 작동

- file-organization module : 논리 블록 주소 물리로 , free space도 관리

- basic file system : mounting , device driver함수 호출

- device drivers : 실제 R/W행위, 인터럽트 처리

b. 정보 저장 방법

- on-disk structure : disk상에 저장되는 정보구조, disk는 여러 partition으로

  1) boot control block

  2) partition control block

  3) directory structure

  4) FCBs

  5) data blocks

- in-memory structure : os 커널 메모리상, on-disk구조의 일부 정보 가져옴

  1) partition table

  2) directory structure

  3) system-wide open file table : copy of FCB

  4) per-process open file table : pointer to FCB of 3)

II. Allocation Methods 실제 DISK 저장 방식

A. Contiguous Allocation

+ : seek time 짧아서 성능 좋다

첫번째 블록 위치, 개수만 알면 모든 블록 찾음

Sequential, random access

- : external fragmentation

File 커지기 어려움

B. Linked allocation(non-contiguous)

각 블록은 다른 블록 가리키는 pointer가짐

+ : no external fragmentation

File size 커지기 가능

- : sequential access only, 오래 걸림

  공간 낭비 (pointer저장)

  Low reliability : 연결 구조 끊어지면 block유실

- 유사한 방법인 FAT(file allocation table)

  link정보를 중앙화된 영역인 FAT에 보관. Block 탐색 시간 감소. 유실 문제

C. Indexed allocation

   file마다 index정보를 담은 index block제공

   multi level index도 가능

III. UNIX on-disk structure (index allocation)

Index block -> i-node

I node : FCB와 같은 개념. Meta data 와 data block 위치 저장.

dir구조 정보가 필요 필요 없음. Dir을 special file 로 본다. (vi . 하면 curdir)

IV. More topics

A. Free space management

   1) Bit vector

      단순, 별도 공간 필요 , 메인 메모리에 저장돼야 OS가 빨리 찾음

   2) Counting

      Block을 연속배정 -> free space도 연속적이므로 1번째 block과 개수만 기억

   3) Grouping

      Free block 주소를 하나의 block에 저장

   4) Linked list

      Free block마다 pointer로 가리킴. 탐색 시간 증가

B. Buffer(block) cache

   - file system에서 담당

   - file data를 메모리에 캐싱.

   Cf) page cache

   - virtual memory 관리 영역

- paging 시 swap out하는 page를 바로 out이 아니라 한동안 메모리상에 유지

*doubling cache problem : buffer cache와 page cache가 동일한 data block저장, 공간 낭비. (without unified buffer cache)

-> unified buffer cache로 두개 통합해서 관리.

C. Journaling file system

DISK에 file R/W을 하는 것은 atomic 하지 않음. 중간에 전원 나가거나. file system consistency가 깨짐

Unix의 파일 지우는 과정

1) Dir entry 지움
2) Inode 반납
3) i-node가 가리키던 data 반납

-> 중간에 crash 돼 버리면 문제

해결 방법

1) fsck(file system checker) : dir 구조, data block 상태, i-node일일히 비교해서 확인. 시간 오래 걸림
2) journaling file system : file을 지우기 전에 'HDD의 journal file'에 지운다는 정보 기입. Journal을 확인하여 삭제 작업 종료 여부 확인함. Journal을 쓰다가 crash나는건 상관없음 . inconsistency없음.

18 Parallelism in HW

I. Advent of Multicore HW

Moore's law

End of frequency scaling

II. Multicore Processors

GPU : 신호 처리 기능만 제공하는 ALU 수백 개 , 캐시나 control부분 적다, 동일한 면적이면 GPU에 더 많은 core

Heterogeneity : 서로 다른 코어를 적재, 에너지 효율 향상. 처리 속도 낮을 때는 저 성능 코어, 높은 경우 고 성능 core활용

Manycore : 범용 core대량 적재 , core간 통신 비용 비대칭적(멀수록 큼)

+ : 성능 높이고 에너지 소비 최적화 , isolation

III. Amdahl's law

core수와 실제 속도 향상 관계

사실상 p(병렬 실행 가능 비율)이 성능을 크게 좌우함

Speedup = 1 / (1-p) + (p/n)

IV. Concurrency and Parallelism

논리적 병렬성(SW) , 물리적 병렬성(HW)

V. Parallelism in HW

A. Pipelining (ILP) : 각 fu가 다른 instruction처리

B. Superscalar , VLIW (ILP)

Superscalar : pipeline을 여러 개

VLIW : 하나의 instruction에 여러 instruction, pipeline여러개 인 것처럼 작동

C. SIMD processing / single instruction multiple data (DLP)

주로 멀티미디어 처리

Vector 형태의 data

Operand가 vector, 하나의 명령어

D. HW multithreading (TLP)

SMT(simultaneous multithreading) : Pipeline stall(bubble)에 다른 스레드를 실행 가능. CPU를 busy하게

Register set이 여러 개라서 context switching overhead감소

19 parallelism in SW

I. Parallelism in SW

A. Data ~ HW의 DLP : data set을 여러 개로 나눠서 task가 각자 처리

B. Task ~ HW의 TLP : 하나의 큰 작업을 여러 개로 (유튜브)

II. Creating a concurrent system

Algorithm design -> SW 구현

III. Parallel Design Patterns

Problem Decomposition : task분할, node와 edge로 표현

Parallel Algorithm design : 기존의 패턴 사용

Parallel Design patterns : 자주 발생 문제에 대해 재사용가능한 일반화된 sol

-Data decomposition : 행렬 계산, 식 의존관계 없어서 병렬 진행

- Recursive decomposition : 초기에 한번만 data set 분할, 진행하면서 동적으로 분할 반복 . divide and conquer

-Hybrid decomposition : data + recursive 최솟값 찾는 예제

-Direct task decomposition

1) 독립적인 task : 너무 많으면 grouping (knapsack과유사) , grouping 후 사이즈들이 유사하도록
2) 의존적인 task : 문제의 처리 과정 분석 , 의존 관계 task graph로 표현 producer-consumer 관계 , precedence constraints(선후관계)

-Exploratory : solution space모두 탐색, 공간 작으면 효과적

-Fork and join : 병렬(workers), 순차(master) 진행이 반복.

-Loop parallelism : loop의 병렬성 사용. scale낮춤. Loop의 iteration간 의존성은 없어야 한다.

-Pipeline : CPU의 pipeline과 비슷함. 동시에 여러 stage

-Divide and conquer : recursive + fork_join 결합

IV. Examples

A. Database query processing

Data decomposition + recursion

B. Monte carlo

task병렬

20 Concurrent programming approaches

I. Automatic Parallelization

컴파일러가 직접 병렬성 찾아서 스레드로 만드는 것 어렵다 loop에 집중

Loop에 함수 호출이 있으면 병렬화 실패

Loop에 pointer가 있으면 data 의존성 판단 어려워서 병렬화 실패

II. OpenMP : 개발자가 컴파일러에게 힌트를 준다

#pragma omp parallel for : 이 다음 for을 병렬화 해도 된다는 의미

#pragma omp parallel : loop가 아니어도 가능

-스레드 개수 지정

1) $export OMP_NUM_THREADS = 2 : 환경변수로 스레드 개수 지정 가능

2) Support lib 함수 호출로도 스레드 개수 지정 가능

-OpenMP에서 참조하는 변수 유형

1) Shared : threads가 공유 ex) 병렬 영역 바깥 변수

-> 컴파일러가 mutex같은 걸로 race condition안나도록 조치 취함

2) Private : 스레드마다 각각 사본 유지하는 변수 ex) loop induction 변수, 영역내부 변수 default로 병렬 영역 변수는 private

-explicit variable scoping(직접 지정 가능)

private(i) shared(~)

-reductions : private, shared 두 성질을 다 가지는 변수, 각 스레드 실행시엔 사본처럼 취급, 각자 계산 마치고 다 더할 때는 shared

reduction(+ : total)

-openMP scheduling

1) Static scheduling : loop 동일한 크기로 잘라서 동일한 횟수로 처리 , 어떤 스레드 작업이 먼저 끝날 수도? ->dynamic

2) Dynamic scheduling : 작업 양을 스레드 개수보다 많이 자름. chunk단위. chunk수행하고 다른 chunk수행 반복. 나중엔 스레드들이 비슷하게 종료

Schedule(dynamic)

III. GPGPU

-graphic처리 외에 일반적인 연산에도 적용한다. 이를 위해 필요한 frame work

1) CUDA

2) OpenCL

-Cpu와 달라 고려할 점

1) GPU는 CPU와 다른 자신만의 instruction set 가짐

2) 자신만 참조하는 별도의 address space 가짐

-Master-slave 방식 : 작업을 주면 GPU가 계산해서 결과 전달

-cudaMemcpy로 GPU에 data copy

21 Multicore operating system

I. Considerations for multiprocessors

Shared memory vs distributed memory

Bus-based(단순) vs network-based(복잡)

Homogeneous vs heterogeneous(core가 다른 유형)

II. Kernels for *small* multicore

1) master-slave organization

master은 OS , slave는 application process 실행

코어가 많으면 slave들이 요청하는 커널 서비스가 많아져서 문제

메모리는 core 수 만큼 partition

2) independent kernels

core마다 다른 OS

메모리는 코어 수 만큼 구분

장점 : 하나의 chip에 다른 OS이므로 별도의 컴퓨터 network로 연결하는 것 보다 효율적, 한 코어가 crash돼도 괜찮음

단점 : 각 코어 스레드 간 sharing안됨, 예를 들어 network장치를 하나의 OS가 쓸 땐 다른 os가 못씀. 동기화 필요

변형 : core마다 동일한 OS돌리지만 독립적인 system. Kernel code는 하나만, kernel data는 따로

3) SMP(symmetric multiprocessing kernels)

Kernel data까지 한 벌로 해서 공유

Data races : kernel data 공유해서 생기는 문제점. Deadlock도 발생할수있음

Giant lock : 커널 전체를 critical section으로 취급하여 하나의 코어만 접근할 수 있도록 하는 방법. Concurrency 관점에서 바틀넥. Not scalable

Corse- & fine-grained lock : 작은 size의 lock. 커널을 조금 더 세분화하여 여러 core가 동시 진입 가능 하도록. More scalable

장점 : OS 서비스나 HW 자원 공유 가능. 자원의 효율성 증가.

단점 : 대칭적 HW구조가 필요함.(모든 cpu가 동일한 inst set제공 , shared memory 구조 필요함) 여전히 scale관점에서 제한적임

III. Kernels for *large* multicore

1) Distributed kernels

core마다 독립적 OS, 분산 컴퓨터처럼 프로그래밍. core마다 공통적인 state가 있는데 message passing으로 state 조율. Scalability에 초점, 좋다. Core 간 interaction은 한계

2) Clustered organization

Core를 grouping. 인접한 core끼리는 m-s 구조나 smp쓰고, cluster간에는 distributed kernel 성격

3) Factored kernels : FOS

Manycore 형태. Core 그룹 마다 기능이 다름. Space multiplexing. 즉 공간적으로 배정. 코어 수백 개