|  |  |
| --- | --- |
| **2021년 2학기 운영체제** | **CSE30100** |
| **중간고사 대체리포트** | |

|  |  |
| --- | --- |
| **학번** | **2018110659** |
| **학과** | **소프트웨어융합학과** |
| **이름** | **이인석** |

Ch1) Introduction

1. operating system란 무엇일까?  
   - 운영체제는 사용자가 컴퓨터 하드웨어를 효율적으로 관리해서 사용자가 컴퓨터 시스템을 편리하게 사용할 수 있도록 해 준다.  
   - main management 구성요소에는 Process Management, Main memory management, I/O system management, secondary storage management, file management등이 있다.  
   - Operating system = kernel + system program(운영체제와 함께 깔림 ex)윈도우 탐색기)
2. Memory hierarchy(메모리 계층 구조)  
   magnetic tape->optical disk->hard-disk drives->nonvolatile memory->main memory->cache->registers  
   위로 갈수록 CPU와 가까이 있어서 속도가 빠르고, 아래로 갈수록 거쳐야 하는 계층이 많기 때문에 느려진다. 경제성 때문에 생긴 구조.
3. I/O control progress(수행 루틴)  
   디바이스 인터럽트 -> CPU에서 현재 수행중인 것 중지, 보존-> 인터럽트 루틴 수행 -> 상태를 복구, 중단된 프로세스 재개
4. Interrupts and Exceptions 의 차이
5. Interrupts -> I/O장치가 CPU에게 이벤트가 발생했다는 사실을 전달함
6. Polling: 지속적으로 CPU에 물어봄(장치 작업 다 끝남?)
7. Hardware interrupt: 장치에서 알려줌(주로 이 방식 사용)
8. Exceptions -> CPU가 자기 자신에게 interrupt를 건다.
9. trap(expected)-> 의도적으로 발생시킴(어플리케이션이 운영체제에 서비스를 요청, ex) system call )
10. Fault(unexpected)->예상치 못한 상황 (ex)참조할 수 없는 메모리)
11. Hardware Protection

* I/O Protection, Memory Protection(자신에 할당되지 않은 메모리 참조 방지), CPU Protection(하나의 프로그램이 CPU를 독차지하지 못하도록 방지(Timer interrupt을 통해))

ch2) System structures

1. User mode와 kernel mode은 무엇일까?  
   - 우선 위 두가지 모드는 dual-mode operation에서 나온 개념이다. User mode는 사용자가 일반적으로 컴퓨터를 사용할 때의 모드이고 Kernel mode는 운영체제가 컴퓨터 제어를 하는 모드이다. User mode에서 kernel mode로 접근하기 위해서는 system call이 필요하다. Read()와 같은 system call 함수를 통해 kernel mode로 진입 후 운영체제에서 이를 처리해준다. 위와 같은 방식으로 user mode와 kernel mode는 system call interface를 통해 통신한다.
2. System call  
   -운영체제에 접근하기 위한 함수  
   - system call interface를 통해 커널모드로 진입, 시스템 제어
3. Monolithic Kernel 의 장단점은? (Unix, Linux)  
   - 장점: 하나로 통합되어 있어서 각 component간의 통신이 효율적  
   - 단점: 특정한 드라이버를 추가하거나 삭제하려면 커널을 재빌드 해야함, 기능을 추가할수록 무거워져서 활용도가 떨어짐.(embedded system에서 사용하기 힘듬)?? , 복잡하게 얽혀있기 때문에 관리하기 힘듬(software engineering issue가 있다)
4. MicroKernel 의 장단점은?(Mac OS)  
   - 중요한 서비스들은 커널의 밑단에 넣고 덜 중요한 서비스들은 user mode에서 각각 server형태로 관리한다. Monolithic kernel의 문제점을 해결하기 위해 탄생.  
   - 장점: 각 서버를 추가하는 방식이기 때문에 기능을 추가하기 쉽다. 시스템이 견고하고, Real-time성이 높다.  
   - 단점: 각 서버의 communication에 오버헤드가 있다는 것이 단점.

**Ch3) Process Concept**

1. Program과 Process의 차이  
   - Program 은 실행’파일’(XX.exe)  
   - Process는 program의 인스턴스
2. Process address space를 구성하는 4가지 영역  
   - Stack(dynamic allocated memory)- ex)지역변수, 매개변수(함수 호출)  
   - Head(dynamic allocated memory)- ex)동적 메모리 할당  
   - Static data(data segment)- ex)전역변수(변화하지 않는 데이터, 프로그램이 종료되어야 없어짐  
   - Code data(text segment)-text
3. Process state의 5가지 상태  
   - New: 프로세스(process)가 생성됨  
   - Running: 명령어(instruction)가 수행됨  
   - Waiting: 이벤트(I/O interrupt)가 발생하여 프로세스가 기다리고 있는 상태  
   - Ready: Running상태가 되기전에 준비가 된 상태  
   - Terminated: 프로세스가 종료됨
4. Process Creation – fork()함수의 실행 과정  
   a. Creates and initializes a new PCB  
   b. Creates and initializes a new address space  
   c. Initializes the address space with a copy of the entire contents of the address(부모 프로세스의 모든 정보를 새로 할당된 공간에 카피)  
   d. initializes the kernel resources to point to the resources used by parent(부모프로세스의 커널 리소스를 자식프로세스가 사용할 수 있게 해줌)  
   e. Places the PCB on the ready queue  
   f. Returns the child’s PID to the parent, and zero to the child
5. Inter-Process Communication(IPC) – 실제 프로세스가 통신하는 방식  
   - Process A가 ProcessB의 영역에 직접 접근해서 통신하는 방식은 불가능하다. 왜그럴까?  
   -> Memory protection에 다른 Process 메모리 영역에 접근 불가함.  
   a. Message passing  
   - 장점: Process A와 Process B가 수행될 때 동기화를 맞춰주는 작업이 필요한데 이것을 커널이 알아서 해줌  
   - 단점: A에서 B로 데이터를 옮길 때 데이터의 크기가 크다면 운영체제를 거쳐서 가야하기 때문에 오버헤드가 발생  
   b. Shared memory  
   - 장점: Shared memory 생성만 운영체제가 하고 두 프로세스가 Shared memory를 통해 통신하기 때문에 오버헤드 발생X  
   - 단점: 두 Process간의 데이터 동기화 작업을 kernel에서 해주는 것이 아닌 직접 프로그래밍 하면서 동기화를 해줘야한다.(번거로움, 동기화 작업을 신경써줘야함)

**Ch4) Multithreaded Programming**

1. Thread와 Process의 차이는?  
   - Process는 독립된 공간에서 다른 process의 영역을 확인하지 못하기 때문에 IPC(Inter-process communication)필요  
   - Thread는 code와 data는 서로 공유하고 register와 stack은 초기화해서 영역을 따로 할당받는다.  
   - Process는 은행 지점, thread는 은행의 창구  
   - Thread의 장점: thread간의 통신, 소통 가능  
   - Thread의 단점: 한 thread가 다른 thread에 영향
2. Threading Issues  
   - UNIX가 만들어진 10년 후 thread개념이 등장. 이슈가 많음  
   a. Semantics of fork() and exec() system calls - two versions of fork()  
   - Ex) thread를 5개 생성했다. 그 중 1개를 fork()를 통해 생성한다면, thread가 6개가 되어야 할까? 아니면 전체 복사를 통해 process 2개, thread 10개가 생성되어야 할까?  
   - 답이 없다. 운영체제마다 다름, thread에서는 fork()를 하지 않도록 만드는 것이 일반적임.  
   b. Thread cancellation  
   - Asynchronous cancellation(비동기식 취소): 즉시 취소해야 하는 thread 강제 종료  
   - Deferred cancellation(지연 취소): thread가 주기적으로 자기가 강제 종료되어야 할지를 확인한 후 강제 종료  
   - Thread 취소를 어렵게 만드는 것? – 취소하는 thread에 할당된 자원문제. 갑자기 thread를 강제 종료하면 thread의 자원, 공유, 자료구조에 관한 내용의 수정이 일어나야 함.  
   c. Signal handling – signal은 event발생으로 생겨나서 프로세스에 전달되어 시그널을 핸들하게 된다. 동기식, 비동기식으로 나뉨.  
   d. Thread pools - 프로세스를 시작 할 때 미리 일정한 수의 스레드들을 풀로 만들어 두는 것  
   e. Thread specific data - 스레드의 장점으로 한 프로세스에 속한 스레드들은 그 프로세스의 자료를 모두 공유한다. 그런 각 상황에 따라서 스레드들은 자기만 접근할 수 있는 자료를 가질 필요가 있다. 이러한 자료를 스레드별 데이터라고 한다.

Ch5) Process Scheduling

1. CPU burst와 I/O burst의 차이  
   - CPU burst: CPU가 명령어가 수행되는 구간 (Process가 running상태)  
   - I/O burst: wait상태에서 I/O가 끝날 때 까지 기다리는 구간(Process가 wait상태)  
   - CPU-bound process: CPU burst가 큰 프로세스(연산 위주)  
   - I/O-bound process: I/O burst가 큰 프로세스(I/O작업 위주)  
   - 우리가 쓰는 I/O-bound process에는 어떤 것들이 있을까? – 문서편집기(아래한글), 동영상 재생 프로그램 등등 대부분의 프로그램  
   - CPU-bound process에는 어떤 것들이 있을까? – 수퍼 컴퓨터의 기상예측 프로그램(계산이 많이 필요함) 등등 과학용 계산 프로그램. 그나마 우리가 쓰는 것에 가까운 것은 압축 프로그램(알집)
2. CPU Scheduler  
   - CPU scheduling이란 short-term scheduling을 말한다.  
   - Long-term scheduling: 옛날에 작은 메모리 환경일 때 어떤 프로세스를 메모리에 올려놓을지 결정하는 것(현재는 없어짐)  
   - Short-term scheduling: 그 메모리에 올라가 있는 동안, 프로세스 중에 어떤 것을 먼저 CPU에 올려놓고 처리할 것인지 결정하는 것(현재의 CPU scheduling)
3. Preemptive(선점) VS Non-preemptive(비선점)  
   - Preemptive – 중요도가 높은 프로세스가 등장하면 기존에 수행되던 프로세스를 중단하고 그 프로세스를 먼저 수행  
   - Non-preemptive – 더 중요한 프로세스가 들어오더라도 하던 것을 계속 수행  
   - Preemptive 스케줄링이 훨씬 효율적인 방법임. 대부분의 운영체제가 이 방식을 지원
4. Scheduling Criteria  
   a. CPU utilization- CPU사용률  
   b. Throughput – 단위 시간당 처리량  
   c. Turnaround time – 총 수행 시간  
   d. Waiting time – Ready Queue에서 대기하는 시간(Device Queue에서 대기하는 시간은 포함되지 않음(wait Queue, I/O수행 시간))  
   e. Response time – 사용자에게 응답 시간  
     
   - CPU-bound process는 a, b에 초점, I/O-bound process는 c, d, e에 초점을 맞추는 것이 좋다.  
   - CPU-bound process는 context switching을 적게 하는 것이 좋고, I/O-bound process는 많이 하는 것이 좋다. 현재는 두가지를 모두 만족하는 방법은 없다.  
   - Trade-off의 관계 – 하나가 좋아지면 하나가 나빠짐
5. Scheduling Algorithms

* FCFS Scheduling, SJF Scheduling, Priority Scheduling, Round Robin(RR)

1. First-Come, First-Served(FCFS) Scheduling  
   - 먼저 들어온 것을 먼저 수행, Queue의 자료구조(FIFO)  
   - 일상생활에서 많이 사용(음식점 줄, 은행, 마트 등)  
   - Preemptive  
   - 가장 공평한 알고리즘(Fairness)  
   - No starvation  
   - 평균 Waiting타임 측면에서 아주 좋지 않은 알고리즘  
   - Convoy effect – 수행시간이 짧은 프로세스가 뒤로 밀려서 효율성이 매우 떨어지는 현상  
   - 어짜피 전체 수행시간에는 변함이 없는데 왜 waiting time을 줄이려고 할까?

* 사용자와 소통해야 하는 interactive 시스템에서는 waiting time을 줄여주는 것이 사용자의 만족도를 높일 수 있기 때문이다.

1. Shortest-Job-First(SJF) Scheduling  
   - Preemptive(SRTF) VS Non-Preemptive  
   - SJF is optimal(waiting 타임 측면)  
   - Fair 하지 않음
2. Non-Preemptive SJF
3. Preemptive SJF (=SRTF, Shortest-remaining-time-first)  
   - Non-preemtive SJF보다 preemptive SJF이 더 waiting 타임을 줄일 수 있는 방법이다.  
   - 하지만 이러한 SJF는 optimal이지만, 실제 컴퓨터에 적용하기엔 무리가 있다. Process를 실제로 실행해 보기 전에는 CPU burst time을 정확히 알 수가 없기 때문.
4. Priority Scheduling  
   - 각 process에 우선순위를 줘서 높은 것 먼저 처리  
   - Fair하지 않음 -> Problem=starvation현상 -> Solution=Aging  
   - Priority가 같은 process들 끼리 줄 세우고 높은 순으로 처리  
   - process에서 어떤 기준으로 priority를 줄 것인가?, 같은 priority안에 있는 process는 어떻게 처리할 것인가? 의 문제가 남아있음  
   🡪 Priority4는 SCFS, priority3는 SJF 이러한 방법으로 설정 가능, 각 Priority는 Round Robin(RR)로 설정하는 것이 가장 합당
5. Round Robin(RR)  
   - Time quantum을 두고 돌아가면서 처리하는 방식  
   - Robin은 새 이름(새끼들에게 밥을 골고루 나눠서 준다고 함)  
   - Time Quantum 과Context Switch Time에 따라 성능의 차이를 보인다.  
   -> Trade-off 관계에 있기 때문에 적절하게 time quantum을 설정하는 것이 중요  
   -> CPU burst time(평균?)보다 약간 더 크게 설정하는 것이 일반적  
   -> 현재 리눅스에서는 60ms의 값을 취하고 있다. 즉 우리가 수행하는 process의 CPU burst time이 60ms보다 조금 작다

**Ch6) Synchronization**

1. Threads cooperate in multithreaded programs  
   - 쓰레드는 멀티쓰레드 환경에서 서로의 자원을 공유한다.(process내에서 각각 register와 stack을 가지되 code와 data는 공유한다) 따라서 동기화가 필요하다.
2. Synchronization Problem  
   - 두 개의 concurrent threads(or process)가 synchronization없이 공유 자원을 서로 접근하려 했기 때문에 문제 발생  
   - 이러한 상태를 Race Condition 이라고 한다. Race condition일 때의 결과는 non - deterministic 하다고 한다.(deterministic: 항상 같은 결과, non-deterministic: 항상 같은 결과가 아님, 때에 따라 달라짐)  
   - Synchronization은 shared data structure에서 필수이다.
3. Critical-Section problem  
   - Race condition을 유발하는 코드 level에서의 구간, 영역을 말한다.  
   - 각각의 process는 shared data에 접근하는 critical section의 code segment를 가지고 있다.  
   - 이후에 한 개의 process가 critical section에 있을 때는 다른 process가 critical section에 접근하지 못하도록 보장하는 방법으로 이 문제를 해결한다.
4. Solution to Critical-Section Problem
5. Mutual Exclusion  
   - 상호 배제, 나만 사용  
   - P1 process가 critical section을 실행하고 있을 때, 다른 어떤 process도 critical section에 진입할 수 없다.
6. Progress  
   - Critical section을 아무도 안 쓰고 있으면 내가 사용  
   - 만약 어떤 process도 critical section을 수행하고 있지 않고 critical section에 진입하고자 하는 process가 있으면, 그 process를 수행해야한다.
7. Bounded waiting  
   - Critical section에 들어가야할 process는 무한히 대기하고 있으면 안된다. Bounded time이 있어야 한다.

* 위 3가지 조건을 만족해야 critical section problem을 해결할 수 있다.

1. Mechanisms for critical sections
2. Locks(잠금)  
   - 가장 원초적이고 간단하고 쉬운 방법임  
   - 하지만 근본적으로 critical section problem을 해결하지 못한다.(while부분에서 timer interrupt가 일어난다면? 또 다른 무한 루프 발생)  
   - 아래 2, 3, 4는 lock에 대한 solution임
3. Software-only algorithm  
   - Peterson algorithms for two processes – 2개의 process를 위한 알고리즘(flag와 turn 변수)  
   - Bakery algorithms for n processes – n개의 process를 위한 알고리즘(각 process에 번호표 부여)  
   - 위 두가지 알고리즘은 3가지 조건(mutual exclusion, process, bounded waiting)을 모두 만족시킨다. 하지만 오버헤드가 크게 발생, 현재 운영체제에서 사용하지 않는다.
4. Hardware atomic instruction  
   - Test-and-set (bounded waiting을 만족시키지 못함)  
   - Swap  
   - 하나의 명령어(hardware atomic instruction)로 Lock, unlock을 구현
5. Disable interrupt  
   - 문제의 원인인 Interrupt를 disable시킴. 언뜻 보면 가장 합당한 항법이라고 생각할 수 있지만 오버헤드가 매우 크기 때문에 비효율적인 방법
6. Semaphores(higher-level synchronization -> user-mode)  
   - Semaphore 변수(0이하면 실행x, 1이상이면 실행o)를 통해 process 제어  
   - 이를 통해 해결할 수 있는 3가지 classical problems(bounded-buffer problem, readers and writers problem, dining philosophers problem)
7. Monitors(higher-level synchronization) - Signal, waiting을 통한 방법