4,5주차 Process

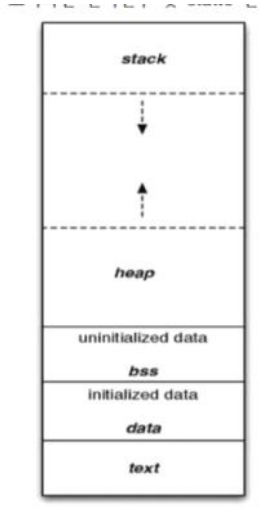
2022년 4월 11일 월요일 오후 3:39

Process 프로세스

실행중인 혹은 실행을 위해 주기억장치 상에 적재된 프로그램

- 메모리에 load된 순간 부터 프로그램은 프로세스가 된다

프로세서가 포함하는 정보들



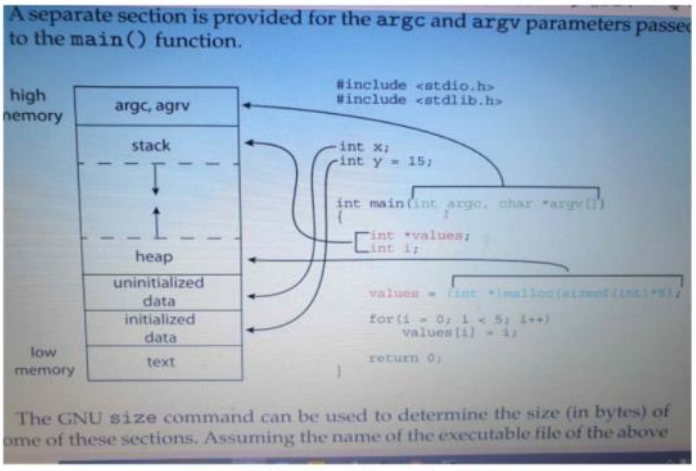
[프로세스의 주기억장치 상의 구조]

화면 캡처: 2022-04-16 오후 9:48

- **Stack section**
  - 지역변수, 매개변수,분기 후 돌아올 명령어 주수등이 저장되고 ,함수 반환값 반환후 ,종료시 top부터 제거 되는 구조 (stack)
  - Function call 생각 하면 됨
- **Heap section**
  - 동적으로 메모리 할당/해제가 이뤄지는 영역
  - Malloc, free,new ...
- **Text section**
  - 메크로 상수 및 함수의 선언 및 정의등 **소스 코드 => 기계어로 저장된 공간**
- **Data section**
  - 소스 코드 내에 선언된 전역 변수, static 변수 등이 저장된 공간
  - **Bss** : 초기화 되지 않은 전역변수 및 static 변수 ,PCB 이곳에 저장 된다
  - **Data**: 초기화 된 전역변수 및 static 변수

완전히 동일한 프로그램을 여러 개 실행을 시키더라도 각각이 별도의 데이터영역, 스택영역,힙 영역이 부여된다 즉 별도의 process로 간주되는것이다

예) 같은 크롬을 4개 뛰우더라도 별도의 프로세서로 인식한다 사실 크롬은 멀티 프로세스 기반이라고 한다



[C로 작성된 프로그램의 프로세스 구조]

화면 캡처: 2022-04-16 오후 10:01

- OS must:
  - Create, delete, suspend, resume, and schedule processes
  - Support inter-process communication and synchronization, handle deadlock

화면 캡처: 2022-04-14 오전 10:40

어느 프로그램이던지 탑재되어있는 API

- Create
- Open
- Read
- Write
- Close

-> socket도 같은 메커니즘 사용

File systme management

Disk는 많은양의 저장 공간을 제공하지만 직접 사용하지는 않음

- File을 제공하고 파일의 다양한 기능을 제공한다
- Directories 제공

- OS must:
  - Create and delete files and directories
  - Manipulate files and directories
    - Read, write, extend, rename, copy, protect
  - Provide general higher-level services
    - Backups, accounting, quotas

3 Fall 1998, Lecture 04

화면 캡처: 2022-04-14 오전 10:49

Os는 directory 관리 해야하고 파일과 directoires를 다뤄야한다

Memory Management

Memory에는 상태들을 저장 시킨다 가장 효율적으로 메모리를 쓴다는것은
Locality of Reference의 효과가 잘 나타나는 것을 의미한다
-> OOP로 만들수록 locality of reference 효과를 이끌어낼수있다
->module화 많이 하면 더 효과적인 locality of reference를 볼수있다

- OS must:
  - Mechanics
    - Keep track of memory in use
    - Keep track of unused ("free") memory
    - Protect memory space
    - Allocate, deallocate space for processes
    - Swap processes: memory <–> disk
  - Policies
    - Decide when to load each process into memory
    - Decide how much memory space to allocate each process
    - Decide when a process should be removed from memory

화면 캡처: 2022-04-14 오전 10:55

Os는 policies에 따라 mechanics적으로 메모리 관리 한다

polices내용
- 프로세스를 메모리 어디에 load할것인지
- 얼만큼의 space로 메모리 할당 할것인지
- 언제 remove할것인지에

정해진 polices내용에 따라 메모리 관리를 mechanics적으로 이루어진다

프로세스에 할당,해제 하고 메모리 공간을 분리하고 등등이 이루어진다

Disk Management

Formatting 방법에 관한 이야기?

- File system 아래에 있는 실제적인 HW이다

■ OS must:
- Manage disk space at low level:
  - Keep track of used spaces
  - Keep track of unused (free) space
  - Keep track of "bad blocks"
- Handle low-level disk functions, such as:
  - Scheduling of disk operations
  - Head movement
- Note fine line between disk management and file system management

화면 캡처: 2022-04-14 오전 10:59

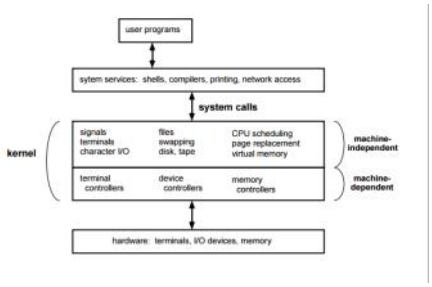
OS는 low - level에서 disk관리를 한다

System call

Process control한다

- Program End/abort(강제 중단)
- Load /excute another program
- Create/terminate a process
  - Process create,terminate 단계에서 system call이 관여한다?

OS structure : Large Kernel

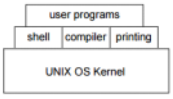


화면 캡처: 2022-04-14 오전 11:05

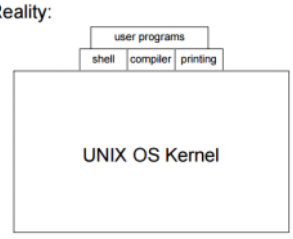
OS의 한 부분인 kernel은 user programs와 구분 되어있다 보호되고 있음

특별한 명령어를 통해 접근 가능 하다

■ Ideal:



■ Reality:



화면 캡처: 2022-04-14 오전 11:07

Kernel process안에서 Active 하게 돌아가는 프로세스

- Schedule
- Swapped
- Poenes?

나머지는 passive

Passive 프로세스는 active가 있어야 돌아갈수있다 passive 독립적으로 run 하지 않는다

OS Design issue

Os를 layer 구조로 만들면 ?

- Module화 되고 simplicity 된다

그러나
- Performance 문제가 심하다

OS는 simple한것보다 성능 performace를 더 중요하게 생각해야한다

Process Model

Process는 왜 필요한가?

- 하나의 프로그램을 여러 번 돌리기 위해서
- Process schduling필요하다 : IO 시간이 CPU 시간보다 느리기 때문에 scheduling필요하다
- 최악은 순차적으로 같은일 하는 경우이다
- 입력 - 계산 - 출력
    - 입/출은 시간이 길기 때문에 새끼 cpu가 처리
    - 계산을 엄마 cpu가 하니까 동시적으로 동작하는것이다

Passive 와 Active를 구분해야한다
프로그램은 passive고 프로세서는 active이다
Active는 passive가 있어야 돌아갈수있다.

여러명의 사람들이 같은 프로그램 실행 할수있다
이때 각 사감들에게 running하는 프로그램은 각각 다른 process로 구분 된다

Process creation /Terminate / Execution

Reason for process creation

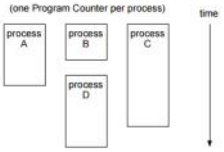
- User logs on
- User start a program
- OS creates process to provide a service
    - Printer daemon to manage printer
- Programes starts another process

Reason for process Terminate

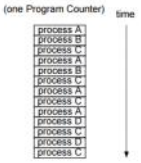
- Normal completion -> 정상 종료시
- Arithmetic error, data misuse.. .. -> error났을때
- Invalid instruction execution -> 부적절한 명령어 실행 되었을때
- 메모리 부족할때
- 다른 process 침범시
- I/O failure

Process Execution

■ Conceptual model of 4 processes executing:



■ Actual interleaved execution of the 4 processes:

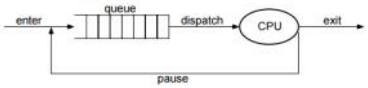


3 Fall 1998, Lecture 05

화면 캡처: 2022-04-14 오전 11:38

Two state process model

■ Queuing diagram:



■ CPU scheduling (round-robin)

- Queue is first-in, first-out (FIFO) list
- CPU scheduler takes process at head of queue, runs it on CPU for one time slice, then puts it back at tail of queue

화면 캡처: 2022-04-14 오전 11:39

공평성을 보장한다 : 공평하게 정해진 시간동안 cpu쓴다 다 안끝난 process는 ready queue에 들어가게 됨

Time tick 발생 구간 동안 process는 돌수있다

Running 상태에서 wating 상태로 넘어가는 이유
-> 후반에 process state관련에 한번 더 언급 될 예정

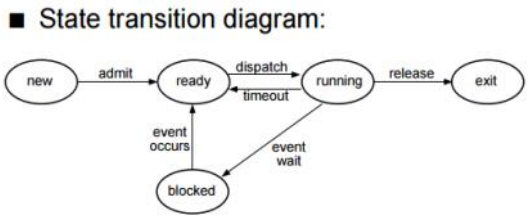
- I/O 때문이다
- User의 key 입력을 받아야하는 상황일때
- Output screen에 출력해야하는 경우
- OS는 ready상태와 waitting상태를 구분해야한다
- I/O 이외에도 Hardware Event , OS operation(우선순위문제등)에 의해 watting으로 가기도 한다

- **Wait for user to type the next key (다음 키 칠때까지 기달)**
- **Wait for outptut to appear on the screen (display 시킬 때까지 기달)**
- **Program tried to read a file (OS가 어떤 블록을 읽고 실제로 메모리에서 요청된 정보를 읽을 때까지)**
- **Netscape tries to follw a link (URL) – 웹사이트에서 주는 정보를 다 받을 때까지**

Five State process Model

New,ready,running,waitting,terminate 단계가 있다.

- **New**
  - 해당 프로세스의 PCB가 커널 내부에 생성
  - Process가 처음 만들어질때
- **Ready**
  - 준비상태 running할 차례를 기다리고 있을때
- **Waiting**
  - 프로세스가 실행도중 어떤일이 일어나기를 기다리는 상태
  - 입출력 대기 : I/O 의 속도가 cpu 연산속도보다 느리다는것을 알아라
  - 시간이 제일 길다 i/o 디바이스마다 linked list처럼 처리해야할 프로세스 컨트롤 블록이 매달려 있다 프로세스가 입출력 기다리면서 새끼 cpu가 입출력 실행
  - Critical Section을 기다릴때도 여기 watting상태를 사용한다
- **Running**
  - 명령어가 주기억 장치로 부터 읽혀지면서 실행되는 상태
- **Terminated**
  - 프로세스가 완료되고 종료
  - Pcb도 제거 된다



화면 캡처: 2022-04-14 오전 11:46

상태 변환

New->ready

- Cpu scheduler가 new로 만들어진 프로세스를 ready queue에 넣을때

Ready->running

- Sheduling algorithm에 따라 cpu scheduler가 실행할 프로세스를 결정한다
- Preemption이라고 한다

Running -> ready

- Process가 cpu에 정해진 시간을 다 썼을 경우 (안끝나더라도 공평성에 의해 시간이 끝나면 들어간다)

Running -> watting

- Process가 I/O 라던가 event를 기다려야할때

Watting -> ready

- 기다려야했던 상황이 끝났을때 언제든지 다시 ready로 간다

Terminate

pcb정보만 남아있는 상태 scheduler가 관여함 cpu 시간을 할당 해주어야 하기 때문이다

상태 전환 이게 왜 필요한가?

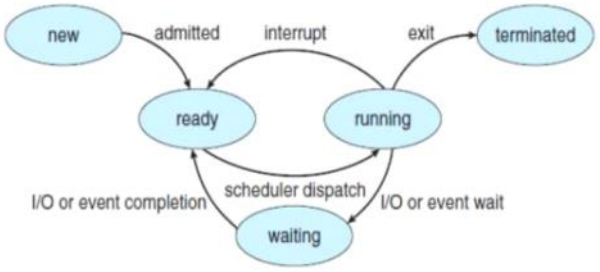
엄마 cpu가 computing과 I/O 처리 까지 한다면 자원의 활용성이 떨어진다
왜냐하면 computing시간보다 i/o 처리 시간이 매우 길기 때문이다.
-> process scheduling이 필요한 이유이다.

엄마 cpu는 새끼 cpu에게 보내서 동시 병렬적으로 돌아가게 한다

-> process scheduler필요한 이유이기도하다

프로세스 상태 전이

- **Scheduler dispatch**
  - Cpu 스케줄러가 준비 상태의 프로세스 중 어느 하나를 골라 cpu할당해줌
- **Interrupt**
  - 인터럽트 걸리면 실행 상태의 프로세스는 준비 상태가 되고 우선 순위가 높은 프로세스를 cpu스케줄러가 실행
- **I/O or Event Wait**
  - Cpu가 실행중인 프로세스가 io 처리 및 이베트 발생 해야한다면 wati상태로 전환 되고
  - 이뤄지기 전까지 wait에 대기 그와 동시에 우선순위가 높은 프로세스를 cpu스케줄러가 실행 (wait 상태에 있는 프로세스는 우선순위와 스케줄러 선택을 고려하지 않음)
- **IO or Event 완료**
  - Wait 상태에서 io 끝나는대로 ready로 전이 된다



[프로세스의 상태 및 상태 전이도]

화면 캡처: 2022-04-16 오후 10:35

Process state consists of

- Code for the program
- Program's static and dynamic data
- Program's procedure call stack
- Contents of general purpose registers
- Contents of Program Counter (PC) —address of next instruction to be executed
- Contents of Stack Pointer (SP)
- Contents of Program Status Word (PSW) — interrupt status, condition codes, etc.
- OS resources in use (e.g., memory, open files, connections to other programs)
- Accounting information

Everything necessary to resume the process' execution if it is somehow put aside temporarily

Fall 1998, Lecture 05

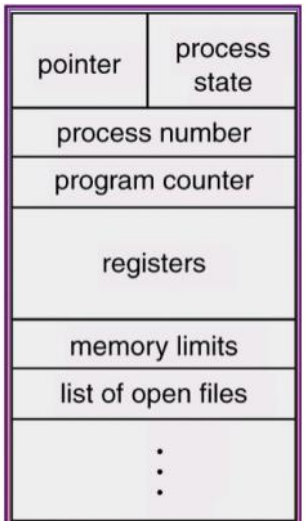
화면 캡처: 2022-04-14 오후 12:05

다시 시작 하기 위해 필요한 모든것 프로세스가 일시적으로 보류된 경우 프로세스 실행

Process Control Block (PCB)

Cpu는 여러개의 프로세스를 여러번 바꿔 가면서 실행 프로세스 정보와 상태를 저장해 놓은 자료구조

커널 안에 bss 영역에 저장이 된다



화면 캡처: 2022-04-16 오후 10:28

- **PID (process id number)**
- **Userid of owner**
  - 계정정보 (자원 사용시간등을 관리)
- **Memory space**
  - 메모리 관리 정보(page table,segment table)
- **Pc,sp, register 정보들**
  - Cpu 레지스터에 있던 그 당시 상태 값
- **Process State**
  - new,ready,watting....
- **Cpu scheduling information**
  - 우선순위등과 같은 스케줄링 관련 정보들 -> 우선순위 때문에 스케줄러가 관여한다
- **I/O state**
  - 입출력 상태 정보 (할당 받은 입출력 장치, 파일등에 대한 정보등)
- **context saving 영역 저장**

Interrupt 종류

- IO interrupt -> keyboard 입력,
- Clock interrupt
- Console interrupt
- Program check interrupt
- Machine check interrupt
- Inter process interrupt

- System call interrupt

System call vs interrupt 차이

System call : 응용 프로그램의 요청에 따라 커널에 접근하기 위한 interface
-> 운영체제 서비스를 접근하기 위한 유일한 수단. 프로그램이 컴퓨터 자원을 사용하기 위해서는 sytstem call을 통해 커널 자원 사용을 요청한다

프로세스 제어 : 프로세스를 생성 ,중지(fork,exec..)등은 system call에서 이루어진다

interrupt : 입출력 연산 혹은 예외 상황이 발생하여 처리가 필요할때 처리 할수있도록 하는 것

비 동기식 인터럽트 : 하드웨어에 의해
입출력 장치 ,타이밍 장치 ,전원등 외부적인 요인에 의해 발생
동기식 인터럽트 : 소프트웨어에 의해 걸린다
소프트웨어가 os서비스를 요청하거나 에러를 일으켰을때

Extern interrupt :

- **Cpu 외부에서 interrupt 요구 신호에 의해 발생하는 인터럽트**
- **하드웨어 흐름에 의해 생기는 인터럽트 이므로 비 동기적 특성을 가지고 있다**

- **IO interrupt**
  - 입출력 작업이 종료되어 결과를 반환하거나 오류에 의해 정지 되었을때 발생
- **Power fail interrupt**
  - 전원 공급이 중단 되었을때
- **Machin chcek(기계착오) interrupt**
  - Cpu 기능이 잘못 되었을때 기계 착오 있을때 발생
- **External interrupt**
  - 외부 장치로부터 인터럽트 오거나 crt C 키 발생 할때

Inner interrupt == 소프트웨어 인터럽트 Trap

- **Cpu 내부에서 발생하는 인터럽트**
  잘못된 명령 혹은 데이터 사용할때 발생
  프로그램 내부의 명령어에 의하여 고정적인 위치에서 발생하는 인터럽트로 동기적 특성을 가진다

- **프로그램적으로 발생하는 오류에 의하여 발생**
  - **0으로 나누기 over/under flow , 예외 명령어등 ?**
  - **시스템 콜에 의하여 발생**
  - **Trap의 한 종류가 system call**

- **Trap의 한 종류가 system call 이다 즉 system call은 trap에 의해 처리 됨**

IO 기준이다

프로세스 A가 디스크로부터 파일을 읽어오는 명령을 실행한다고 했을 때 내부적으로 일어나는 과정은 다음과 같다.

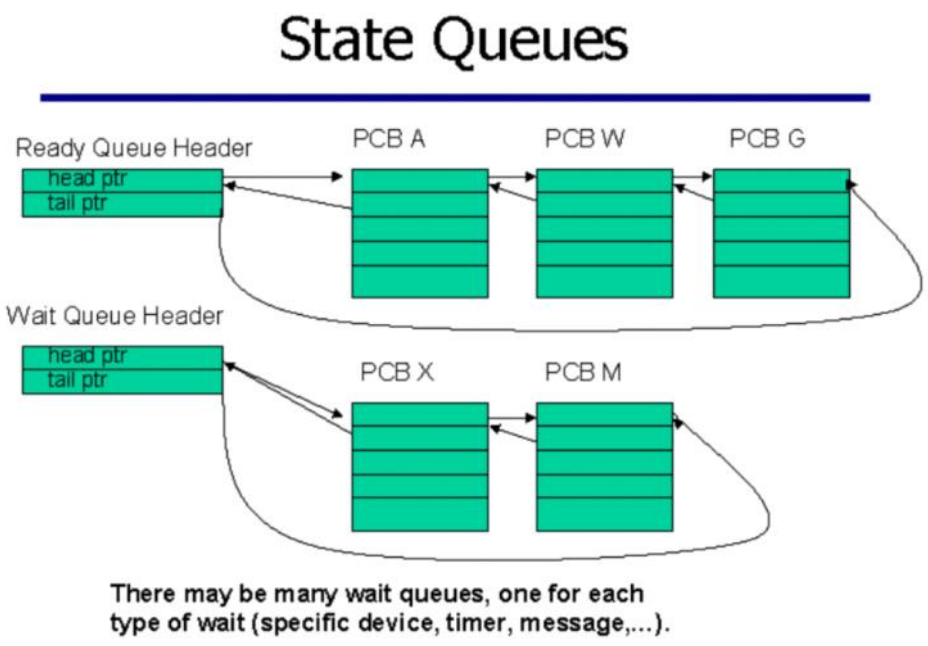
1. 프로세스 A가 시스템 콜을 요청하면서 CPU 내에 인터럽트 라인을 세팅한다.
2. CPU는 실행 중이던 명령어를 마치고 인터럽트 라인을 통해 인터럽트가 걸렸음을 인지한다.
3. mode bit를 0으로 바꾸고 OS에게 제어권을 넘긴다.
4. 현재 실행 중이던 프로세서의 상태 및 정보를 PCB(process control block)에 저장한다. 그리고 PC(program counter)에는 다음에 실행할 명령어의 주소를 저장한다.
5. 시스템 콜 루틴에 해당하는 곳으로 점프하고, 시스템 콜 테이블을 참조하여 파일 읽기에 해당하는 시스템 콜을 실행한다.
6. 해당 루틴을 끝내면, mode bit를 1로 바꾸고 PCB에 저장했던 상태들과 PC를 복원시킨다.
7. PC에 저장된 주소(=마지막으로 실행했던 명령어의 다음)로 점프하여 계속 실행한다.

출처: <https://velog.io/@chowisely/Operating-Systems-Interrupt-System-Call>

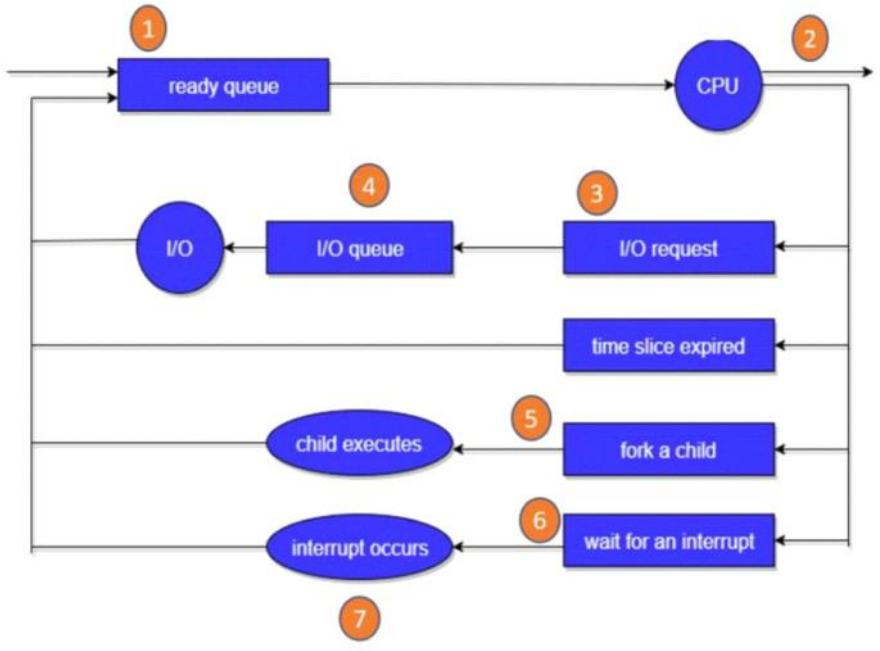
Schedulers

scheduling Queue

- **Ready Queue**
  - **Cpu에 할당되기르 ㄹ기다리는 프로세스들**
  - **Linked list형태로 커널 내 저장**
    - **굳이 linked list 아니더라도 우선순위 큐 ,등으로 구현 가능**
    - **어떤 알고리즘 사용하느냐에 따라 다르다**
  - **Linked list 각 노드는 각 프로세스의 정보를 담은 pcb**
  - **Ready queue는 유일하다**
- **Wait Queue**
  - **IO 완료와 같은 특정 이벤트가 일어나기 까지 대기 상태인 프로세스들이 양방향 연결리스트의 형태로 저장**
  - **Wait queue는 여러개 존재 가능 [하나의 컴퓨터에 다양한 io 장치들의 개수 만큼 존재]**



화면 캡처: 2022-04-17 오전 11:55



[스케쥴링 큐를 활용하여 나타낸 프로세스 스케쥴링]

화면 캡처: 2022-04-17 오전 11:57

이 그림을 이해 하는 것이 중요하다 전체적인 흐름:

사각형은 cpu의 제어권이 넘어가는 이유
원형은 프로세스가 요청한 이벤트가 발생했음을 의미한다

Cpu에서 fork 발생하면 child exec 하고 ready queue에 들어간다
Interrupt발생하면 interrupt 처리하고 ready queue로 들어간다

Ready Queue에 있는 process들을 어떠한 순서로 처리할 것 인가?를 결정하는게 scheduler

선택한걸 실행 시켜주는건 dispatcher

//process state설명할때 scheduler가 관여한다고 한 scheduler가 STS이다

Short term scheduler

- **Milies 단위로 실행된다**
- **Process is created or terminated**
  - **Created : Fork될때 마다 우선순위 확인 위해 STS 실행된다**
  - **Terminated : 다른 process 돌게 하기 위해서, 남은 cpu 잔여 시간동안 다른 process돌게 해줄려고 STS 실행된다**

- **Running state --> Wait State**
- **Interrrupt occure 인터럽트 발생시**
  - **IO 출력의 interrupt 시 IO process가 돌림**

STS 목표

- **첫 반응 시간을 최소화 해주는것 즉 응답 시간 최소화**
- **평균 응답시간 최소화 (예측이 중요)**
- **처리량 최대화 (오버헤드 최소화 , 효율적인 자원사용)**
- **Fairness - 공평한 time 동안 cpu 사용**

Scheduler는 언제 작동할까?

- **Procesee created of terminated**
  - Created : fork()될때마다 우선순위 확인하기 위해
  - Terminated : 다른 process돌아가게 하기위해 CPU 잔여 시간 있다면 다른 프로세스에게 할당 해주어야한다
-
- **Interrupt 발생시**
- **Wait 상태로 갔던걸 다시 running할때**

Context switching

Context -> 프로세스 관련된 정보들의 집합
Context switching : 프로세스 관련된 정보들의 집합이 바뀌는것

-> 현재 프로세스의 상태를 저장하고 다른 프로세스의 상태를 읽어오는 과정

커널에 의해 context switching 일어난다 그래서 overhead가 발생하는것이다.

Time sharing system의해 정해진 시간동안만 cpu사용가능한데 사용하다가 process바뀌면
그전에 수행된 레지스터 정보등을 저장을하고 ready에서 다시 오면 그때 restoring해주어서 처리해주는 개념

- Interrupt, time sharing system time 만료 등과 같은 이유로 context switching 발생할때 마다 kernel은 이전에 실행 중이던 프로세스를 PCB에 저장하고 실행할 프로세스의 PCB읽어온다

- Overhead는 내용이 저장 되어야 하는 레지스터 수, 주기억장치 속도 등에 따라 달라진다

System call과 context switching

System call과 context switching은 별개의 개념이다
즉 system call 발생한다고 해서 반드시 context switching되는것은 아니다

=> system call이 해당 system call을 일으킨 프로세스와 독립적으로 실행되는 것이 아니라(별개의 프로세스가 아니라는 의미) 해당 프로세스의 일부로 실행되기 때문이다.

System call은 kernel service 사용하기 위한 interface라는 것을 생각하면 조금 이해할수있다?.

근데 system call은 trap을 통해 이뤄지는데 이때는 문맥 전환 필요하다

왜 사용하냐?

- Overhead 이지만 전체적으로보면 이득
- 시분할 시스템 때문에 자주 발생 즉 레지스터값 저장하고 가지고오는 과정이 자주 발생한다는 것이다 -> overhead
- Process가 io를 기다리거나 다른 이벤트를 기다리는 동안 cpu가 아무것도 안하는것보다 process바꿔주면서 생기는 overhead가 오히려 낮다

그럼 왜 더 자주 안하냐

- 순수한 overhead가 발생하기 때문이다
- cpu더 쓰고 싶은데 switch자주하면 순수한 오버헤드가 되기때문이다
  - 실행 시간보다 스위칭에 더 많은 시간을 쓰게 되니까

프로세스간에 context switching할때는 메모리가 다르기 때문에 메모리 관련 작업을 추가로 해야한다
그래서 Thread간의 문맥전환이 더 가볍다고 하는것이다. -> 메모리 관련 처리 하지 않기 때문이다

Process Creation

부모 프로세스는 자식 프로세스를 만들고 트리를 만든다

일반적으로 부모 프로세스로 부터 생성된 자식 프로세스는 부모 프로세슨의 pcb의 복사본과 주소공간의 복사본을 가진다
하지만 PID는 다르다

Resource sharing **//thread의 resource sharing과 비교를 해보면 더 명확해질 수 있다**

- Parent and children share all resources 모든 자원 공유
- Children share subset of parent's resources 일부분 공유
- Parent and child share no resources 공유하지 않음

Execution

- Parent and children execute concurrently. 부모, 자식 프로세스는 동시에 실행됨
- Parent waits until children terminate. 부모 프로세스는 자식 프로세스가 죽을 때까지 기다림

2가지 방식

Address space

- Child duplicate of parent. – fork
- Child has a program loaded into it. – exec

fork()는 새로운 process(자식 process)를 만든다. -> 부모와 똑같이 복제
exec()는 fork() 이후에 쓰이며, process의 메모리 공간을 바꾼다. -> 자식만의 프로그램이 돌아감

⇒ Linux 상의 자식 프로세스의 생성 (C 언어)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
        pid t pid:

        pid = fork():

        if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed"):
            return 1:
        }
        else if (pid == 0) { /* child process */
            execlp("/bin/ls","ls",NULL):                    // exec () 계열의 함수
        }
        else {                                               // 부모 프로세스의 분기점
            wait(NULL):
            printf("Child Complete"):
        }
            return 0:
}
```

화면 캡처: 2022-04-17 오후 3:50

Fork()

- 새로운 프로세스를 만드는 시스템 콜
- 부모와 똑같이 카피함
- 부모의 fork() return 은 자식의 pid 값
- 자식의 fork() return 값은 0

- Fork()함수로 인해 시스템 호출이 발행하면 현재 부모 프로세스의 상태가 PCB의 형태로 저장/갱신되고 fork() 시스템 호출에 대한 ISR 실행 된다 해당 ISR은 커널 내부의 sys_fork()를 호출한다

- **Sys_fork()**
  - 자식 프로세스의 주소를 할당해주기 위함
  - 부모프로세스와 동일한 주소 공간을 복사
  - 동시에 부모의 PCB 복사하여 자시의 PCB 생성한다

- 단, PID는 부모와 자식 서로 다르다

- ISR이후 프로세스로 돌아오면 부모 프로세스의 return값은 자식의 pid값 자식 프로세스의 return 값은 0
  => 이 때 자식 프로세스는 PCB의 형태로 Ready Queue에 들어가서 ready 상태가 된다

4) else 문으로 돌아온 부모 프로세는 wait system call 들어간다 부모 프로세스는 wait 상태가 됨
5) Cpu 스케줄러에 의해 ready상태 있는 자식 프로세스는 running상태로 들어간다
6) 자식 프로세스는 이전 fork 함수 실행중에 대한 fork 함수에 대한 return을 해야한다
   a. 부모 프로세스의 PCB를 그대로 복사 받았기 때문이다
7) 자식 프로세스는 Return 값으로 0을 반환한다 **else if** 문으로 들어감
   a. **Execlp함수를 통해 자식 프로세스의 주소공간의 모든 영역들은 전혀 새로운 프로세스의 내용들로 바뀌게 됨**
      i. **자식 process의 PCB는 program counter,이전 프로세스의 연산과 관련있는 항목들(레지스터 값, IO 관련 상태 정보등)만이 리셋됨**
      ii. **PID, PPID, CPID는 변하지 않는다**
      iii. **물론 execlp도 system call이기 때문에 먼저 cpu해체 되고 이후 다시 실행되면서 완전히 다른 프로세스로서 실행이 된다**

=> 서로 다른 프로세스가 실행되는 과정

8) 자식 프로세스가 종료가되면(정상적이든 /비 정상적이든) 종료 상태값을 커널에 전달
9) 이때 대기 상태에 있던 부모 프로세스는 ready queue에 들어감으로써 준비가 됨
10) 부모 프로세스가 다시 실행될때 wait함수의 대기 상태 전환 부분에서부터 실행되어 wait함수는 자식프로세스의 종료 상태 코드값을 커널로부터 받음
    a. 이때 자식 프로세스의 모든 메모리 상의 자원이 해제 즉 부모가 wait에서 깨어나서 실행이 되어야 자식 프로세스가 끝이 난다
    b. 부모 프로세스는 많은 자식 프로세스 중에 어느 프로세스를 종료할지 알아야 하기 때문에 wait()함수는 자식프로세스의 pid를 반환한다
11) 이후 부모 프로세스는 지 갈길 간다

Exec()

- **Fork() 이후에 가능 독립적으로 사용도 가능한데 한정적이다**
- **현재 실행되는 프로세스에서 다른 프로세스 일을 하게 하는 것이다. -> fork된 프로세스가 exec 때문에 다른 프로세스로 대체 될수 있는 이유이다.**
- **새로운 프로그램으로 프로세스 메모리 공간을 대체함**
- **Exec를 호출한 프로세스가 아닌 exec()에 의해 호출된 프로세스만 남게 된다**
  - **실행 결과로 생성되는 새로운 프로세는 없다**
  - **Exec로 호출되는 프로그램이 현재 메모리에 올라와있는 프로그램을 덮어서 로딩되기 때문에 현재 메모리에 상주하고있는 이후 프로그램은 무시되어 버린다**
  - **-> fork()가 필요한 이유**
    - **Fork는 별도의 공간을 할당 해주기 때문에 원래의 process는 자기 갈길을 갈수있다**

Fork & exec() 차이점

쓰레드와 프로세스의 예를 들어보고 fork와 exec에서 process control을 살펴 보도록 하자

프로세스와 쓰레드를 설명할 수 있는 쉬운 예는 곰플레이어와 같은 프로그램이다.

avi 파일을 클릭하면 이 프로그램은 자막이 있으면 자막을 보여주고, 사운드를 들려주고, 영상을 보여주며 전체화면으로 바꾸면 끊기지 않고 전체화면으로 바꿔준다.
이것은 단일 프로세스의 단일 쓰레드로는 구현이 불가능하다.(불가능하다기 보다 대단히 어려운 과정을 거쳐야 한다.)
사운드를 들려주는 쓰레드, 영상을 보여주는 쓰레드, 프레임을 조정하는 쓰레드, 자막을 관리하는 쓰레드가 각자 자기 할일을 하고 있으며 각각의 쓰레드가 CPU의 자원을 독점적으로 쓰지 않고 적절히 양보하면서 쓰기 때문에 가능하다.

-> 살펴 볼것 : fork와 exec는 process control 한다고 했으니

곰플레이어 같은 process에서 자막 관리 쓰레드 같은 여러 쓰레드 control 하는 것을 의미하는 것일까?

Exec() 실행 프로그램 예제)

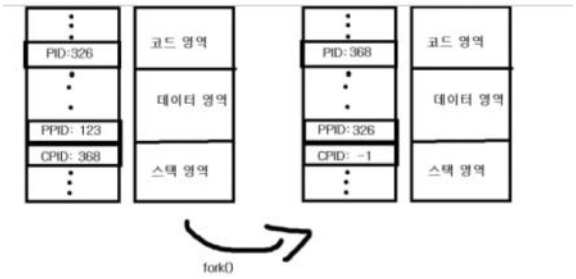
```
Main
{
        Printf("excuting\n")
        Execl("bin/ls","ls","-l",(char*)0);
        Perror("execl failed to run ls");     //execl 함수가 정상 작동이 되면 실행 되지 않음
        Exit(1)
}
```

Exec는 다른 프로세스 실행 시킬 때 현재 프로세스 메모리 공간에 덮어 써버린다
따라서 execl이 정상 작동 되면 perror는 실행 되지 않는다

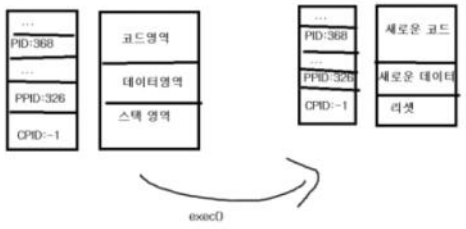
-> 기존 a.out은 새롭게 부른 프로세스 bin/ls에 의해 덮어지게 된다

Exec 호출할때 많은 경우에는 원래의 process가 사라지는 것을 원치 않을것이다.
이런 경우를 해결 해 줄 수 있는 것이 fork()이다.

유닉스 체계에서 inint 프로세스가 실행이 되면 inint을 fork를 하고 fork된 프로세스를 exec로 덮어 씌우도록 하여 프로세스를 계속 생성하는 방식이라고 볼수있다.



[fork() 함수 실행 시, 일어나는 주소 공간의 복사]



[exec() 함수 실행 시, 새로운 프로세스의, 기존 프로세스의 주소공간 대체]

화면 캡처: 2022-04-18 오전 1:14

고아 프로세스 좀비 프로세스

고아 프로세스

부모 프로세스가 자식 프로세스보다 먼저 종료 될 경우 해당 부모 프로세스의 자식 프로세스는 고아 프로세스가 된다

- 고아 프로세스는 이후 좀비 프로세스가 될 수 있다
  - 자신의 반환값을 받을 수 있는 부모 프로세스가 없기 때문이다
- **Linux에서 고아 프로세스 처리**
  - PID가 1인 systemd 프로세스가 고아 프로세스의 새로운 부모 프로세스가 되며
  - 주기적으로 wait()함수를 호출하여 고아 프로세스가 좀비 프로세스가 되지 않도록 한다 해당 고아 프로세스를 소멸 시킨다
- 어떤 OS경우 고아 프로세스 자체를 방지 하기 위해 부모 프로세스가 자식 프로세스 이전에 종료 되면 자식 프로세스도 강제로 종료 시켜 버림

좀비 프로세스

: 부모 프로세스가 wait 호출하기 전에, 자식 프로세스가 종료되는 등의 이유로, 실행을 모두 마친 자식 프로세스가 주기억장치 상에서 사라지지 않고 계속 남아있는 상태 (할 일을 다하여 죽어야하지만, 겉으로는 죽지않은 상태)

좀비 프로세스가 되는 과정

1. 자식 프로세스가 할 일을 다하고, exit 함수나 return 문을 이용하여 값을 반환함
2. 자식 프로세스가 반환한 값은 커널로 넘어감
   이와 동시에, 자식 프로세스는 여전히 주기억장치 상에 소멸되지 않고 존재
3. 커널이, 반환값을 부모 프로세스에게 전달
   a. 부모 프로세스는 먼저 소멸됐기에, 반환값을 받을 수 없음
4. 커널은 부모 프로세스가 반환값을 받을 때까지, 자식 프로세스를 소멸시키지 않음 (wait 함수등을 이용해, 부모 프로세스가 반환값 요청을 하지 않는 이상, 커널은 전달 x)
5. 할 일을 다 했음에도, 자식 프로세스는 소멸되지 않고 주기억장치 상에 계속 존재 = 자원 낭비

※ 자식 프로세스가, 부모 프로세스의 wait() 호출 시점 이전에 종료됐더라도, 커널은 자식프로세스의 반환값을 갖고있으며,이후, 부모 프로세스의 wait() 호출 시, 커널은 반환값을 부모 프로세스에 전달한다. (이후, 좀비 프로세스는 소멸됨)

Cooperating Processes

IPC(inter process communication)
-> 프로세스 간의 통신에 관한 내용이다

독립적인 프로세스는 다른 프로세스의 실행에 영향을 주지도 받지도 못한다
협력 프로세스는 다른 프로세스의 실행에 영향을 주거나 받을수 있다
-> 하나의 일을 끝내기 위해 여러 개의 프로세스가 협력한다

Pros

- **Imformation sharing**
  - 여러 개의 응용 프로그램에서 하나의 데이터 자원을 공유해 써야 하는 경우 있다
- **Computation speed up**
- **Modularity**
- **Convenience**

협력 프로세스로써 어플리케이션이 더 나은 프로그램 구조를 가지게 된다 각각의 단일화된 하나의 프로그램보다 작다

Modularity -> 소프트웨어를 분업화 하니까 caching up [Module화가 높으면 재사용성이 커지기 때문에 cahcing 효율을 높일수 있다]

IPC 통신 방법

2가지가 있다 message passing model , shared memory model

1. **Message Passing Model**
   a. 프로세스 간 메시지의 형태로 데이터를 주고 받는 형태 (직/간접 모두 가능)
   b. 분산 시스템(네트워크에서 client - server model에서 유용하다)
   c. 메시지 패싱 기법에는 통신을 위한 연산 최소 2개가 존재 [송신[send] 수신[recieve]]
   d. 메시지의 전달 매체가 있는가에 따른 분류
      1) **직접 통신**
         i) **송신하는 프로세스 입장에서는 해당 메시지를 수신할 프로세스의 식별자를 명시 수신하는 프로세스 입장에서는 해당 메시지를 송신하는 프로세스의 식별자를 명시**
         ii) 1:1 대응 관계

      2) **간접 통신　　// 메일박스로 통신?**
         a) **메일 박스(Maill Box): kernel 혹은 한 프로세스의 주소 강간 내에 위치한 버퍼 -> 해당 버퍼로 전달된 메시지를 기록하거나 수신 프로세스가 본인 앞으로 수신된 메시지 읽기를 수행하는 공간**

            i) **메일 박스 공간을 통해 프로세스 간 메시지를 주고 받는 방법**
            ii) 각각의 메일 박스가 가지는 고유의 식별자를 인자로 하는 송/수신 연산 진행

            1) 프로세스가 소유하는 주소 공간 내의 메일 박스

               프로세스의 주소 공간 내에 존재하는 고유한 메시지 보관 버퍼로 메시지를 수신할 수신 프로세서를 명확히 할수있다
               -> 프로세스 소멸시 주소강간 해제와 함께 소멸

            2) 운영체제가 소유하는 커널 내의 메일 박스

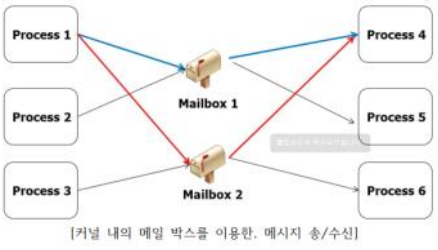
               커널 내의 버퍼로 해당 버퍼를 통해 프로세스들 끼리는 메시지를 주고 받을수있따

               1) 프로세스는 먼저 시스템 호출을 통해 커널 내에 통신을 위한 메일 박스를 생성
               2) 해당 커널의 메일박스를 생성한 프로세스는 자연스럽게 해당 메일 박스를 통해 데이터를 수신만 가능하게 됨
               3) 다른 프로세스들은 해당 커널내의 메일박스로 메시지를 전달

            추가로 알아야하는 사항

            각각의 메일박스는 고유 식별자를 가짐
            만약 공유되는 메일박스의 종류가 여럿이면 두 프로세스간의 통신 경로는 2개 이상이 될수있다

            두 프로세스 간에 통신을 위해서는 반드시 두 프로세스 간에 공유되는 메일 박스가 있어야한다



[커널 내의 메일 박스를 이용한, 메시지 송/수신]

화면 캡처: 2022-04-23 오후 1:57

      3) **동기화**
         a) **메시지 송신시(send 함수 호출시) 수신 프로세스로부터 수신확인을 할때까지 프로세스는 대기 상태(wait)가 됨 반대로 메시지 수신 요청시 메시지를 받을때까지 reveive 연산을 호출한 프로세스는 대기 상태(wait)가 됨**

      4) **비동기화**
         a) **메시지 송신 시 메시지를 보내자마자 수신 프로세스의 수신확인과는 상관없이 프로세스는 실행한다. 반대로 메시지 수신 요청시 현재 보내진 메시가 있든 없든 읽어 들임**

2. **Shared Momory Model 공유 모델**
   a. **여러 개의 프로세스들이 데이터를 공유하는 주기억장치 상의 공유 영역을 두어 협동 프로세스들 간에 해당 공유영역을 자유롭게 사용토록하는 방법**
      i. 메모리 공유전에 , 특정 프로세스에서 공유 메모리를 설정하는 시스템 호출을 발생 시켜야 한다

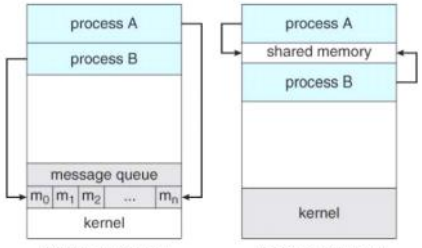
ii. 공유 메모리를 설정하는 시스템 호출을 발생시키는 프로세스의 주소공간의 일부를 공유 메모리로 사용
iii. 처음 공유 메모리를 설정하기 위한 시스템 호출을 제외하면 이후 공유 메모리 사용 접근에는 일절 커널의 개입이 없이 자유롭게 해당 공유 메모리 이용
iv. 보통 운영체제는 한 프로세스가 다른 프로세스의 메모리(주소공간)에 접근 /사용 하는것을 막음
v. 공유 메모리 접근 이전에 공유하는 프로세스들끼리의합의가 있어야한다
vi. 즉 합의하에 한 메모리를 다수의 프로세스가 공유할 경우 쓰기 /읽기 작업에 OS가 관여하지 않는다

b. **공유 메모리 기법은 생산자-소비자 문제에 대한 해법의 일종**
   i. **공유되는 메모리 영역 내에 임시 버퍼를 두어 생산된 데이터를 버퍼에 보관한다. 소비자는 버퍼의 데이터를 사용**
   ii. **즉 버퍼 내에는 생산이 완료된 데이터(=소비될 준비가 끝난 데이터)만 위치하게 됨**

c. **버퍼 2가지 구현 방법**
   i. **Bounded buffer (버퍼 크기 고정)**
      1) 소비자는 버퍼가 비워지면 데이터 사용 불가
      2) 반대로 버퍼가 가득차면 생산자는 버퍼가 조금이라도 비워질때까지 데이터 생산이 불가

   ii. **Unbounded buffer (버퍼 크기 무제한)**
      1) 소비자만 기다리면 됨
      2) 생산이 완료된 데이터를 보관할 버퍼의 크기에 제한이 없으므로 무작정 생산 가능

메시지 패싱 모델 우위 : 적은 양의 데이터를 옮기는데 유리 모델 구현이 쉽다
공유 메모리의 우위 : 편의성 및 속도의 우위 (메모리 상 공유영역에 접근하기 때문이다)



[2가지 프로세스 간 통신 방법]

화면 캡처: 2022-04-23 오후 2:32

Producer - consumer problem

-> cooperating process의 대표적인 사례이다

동기화를 어떻게 해야하느냐의 문제이다

- 생산자-소비자 문제 (Producer-Consumer Problem)
  : 생산자(데이터를 만들어내는 측)가 만들어낸 데이터를, 소비자가(데이터를 사용하는 측) 사용한다 할 때, 생산과 소비는 반드시 동기적으로 이루어져야 함(만들고 있는 데이터를 소비자가 사용할 수 없음).
  ⇒ 즉, 생산자-소비자 문제란, "어떻게 해야 생산-소비 가 동기적으로 이루어질까??" 의 문제
  ⇒ 생산자-소비자 관계의 예시

ex) - 컴파일러-어셈블러
⇒ 컴파일러가 어셈블리어로 번역한 코드를, 어셈블러가 기계어로 된 목적파일로 번역
- 어셈블러-링커/로더
⇒ 어셈블러가 생산한 목적파일들을, 링커/로더가 하나의 실행 파일로 만듦

화면 캡처: 2022-04-23 오후 2:33

네트워킹도 결국 IPC 두개의 프로세스가 통신 하는것이니까 그 프로세스들이 네트워크로 연결된 다른 컴퓨터에 존재하는것
-> ip 주소와 port number보내줘야 이때 port number가 그 컴퓨터의 프로세스를 나타낸다고 이해하면 될듯?
이것을 socket이라 한다
Socket은 tcp/ip 이용 sender/receiver와 동일 하게 create, connect, recevie,listen,send,close가능