Abstraction

Memory Virtualization

- 운영체제가 물리적 메모리를 가상화해서 프로세스별로 가상의 메모리공간을 만들고, 각각의 프로세스가 모든 메모리를 사용하는 것처럼 보이게 하는 것
- 프로그래밍에서 사용하기 쉬움
- 시간과 공간면에서 메모리 효율성이 우수함
- 각각의 프로세스는 물론 운영체제가 독립적임
- 다른 프로세스의 잘못된 메모리 접근을 막음

Multiprogramming and Time sharing

- 옛날의 운영체제는 하나의 프로세스만 메모리에 load했음 -> 효율성이 안좋음
- 현재는 메모리에 여러개의 프로세스를 load -> 효율성 상승
- 보호 문제를 불러일으킴(다른 프로세스로부터의 잘못된 접근)

Address Space

- 운영체제가 물리 공간을 추상화해서 만든 것
- 하나의 프로세스에 대한 모든 정보가 들어있음(Code, Heap, Stack 등)
- Code : 명령어집합, Heap : 동적할당을 위한 공간, Stack : 반환주소, 지역변수, 인수
- 프로그램에서의 모든 주소는 가상의 주소 -> 운영체제가 가상 주소를 물리주소로 변환해서 접근

Memory API

malloc()

- void* malloc(int size_t size)
- Heap에서 메모리 공간을 할당받음
- size_t size = size of memory block(unsigned integer)
- 리턴 값 = 할당받은 heap 메모리 공간의 주(void type pointer), 실패할경우에는 null pointer
- 공간을 할당안하고 접근을 시도 할 경우 Segmentation Fault 발동
- 충분한 공간을 할당하지 않고 값을 넣을경우 뒤쪽의 값은 짤림
- 공간을 할당해놓고 값을 넣지 않으면 Uninitialized Erorr 발생
- **brk** System Call을 사용함 -> 프로그램의 break를 확장(heap의 끝의 주소) -> 프로그래머가 직접사용하는 일은 없어야함

sizeof()

- 해당 변수의 실제 사이즈를 반환(byte)

free()

- void free(void* ptr)
- malloc으로 할당받은 메모리 공간을 free
- 인자로 메모리공간을 가리키는 포인터 값 넣음
- 이미 free한 공간을 다시 free하려고 하면 Undefined error 발생

Memory Leak

- 할당해놓고 사용하지 않는공간을 free() 해주지 않으면 Memory Leak 발생 -> 계속 할당하다보면 메모리 다 씀

Dangling Pointer

- 다 사용하기 전에 메모리 공간을 free() 할 경우 해당 공간을 참조하고 있는 다른 변수쪽에서 오류 발생

calloc()

- void* calloc(size_t num, size_t size)
- size만큼의 메모리블럭을 num개만큼 할당
- Heap에 메모리공간을 할당하고, 모든 바이트를 0으로 초기화함

realloc()

- void* realloc(void* ptr, size_t size)
- 해당 메모리 블럭의 사이즈를 변경

mmap()

- void* mmap(void* ptr, size_t length, int port, int flags, int fd, off_t offset)
- System Call로, 익명의 메모리공간을 할당

Address Translation

- LDE(Limited Direct Execution)과 비슷한 전략 사용, 하드웨어의 지원을 받아 효율성/컨트롤 득
- 하드웨어가 가상주소를 물리주소로 변환시킴
- 운영체제가 적당한 시간에 하드웨어에게 명령을 해줘야됨

Relocation

- 운영체제가 '0'주소가 아닌 다른 어딘가에 프로세스를 저장하기를 원함(Address Space는 0에서 시작)
- Base : 해당 프로세스가 물리적 메모리에서 위치한 곳(시작점)
- Bounds : Address Space의 길이/크기
- Dynamic Relocation : 프로그램이 실행 시작되면, 물리적 메모리 어디에 프로세스가 Load되어야 하는지 운영체제가 정함
  -> physical address = virtual address + base, 0 <= virtual address < bounds
- 물리적 주소의 끝자락 위치를 bounds로 볼 수도 있음

OS Issues for memory virtualizing

- Base and Bounds 접근법을 구현하려면 OS가 움직여야함
  프로세스가 실행 시작될때 : 물리적 공간에서 Address Space가 배치될만한 곳이 있는지 탐색
  프로세스가 종료 될 때 : 메모리를 사용가능하도록 reclaim
  Context Switch가 일어날 때 : Base/Bounds 쌍을 저장

프로세스 실행 시작 시

- 새로운 Address Space를 위한 물리적 공간을 찾아야 함

- Free List : 현재 사용하지 않고있는 물리적 메모리 공간의 리스트

프로세스 종료 시

- 운영체제가 사용한 메모리를 다시 Free List에 집어넣어야 함

Context Switch 시

- Base/Bounds 쌍을 저장/복원 해야 함(Process Structure 또는 Process Control Block에서)