1. What abstraction is provided by process?

Virtual CPU(by appropriate scheduling), independent PC, SP, context flow, address space(= role of memory), states(= PCB)

==> CPU, PC, SP, context(function), memory, states

==> an abstraction of a running program and its state

==> provide an abstraction of whole (virtual) machine

2. Scheduling algorithm design

3. Threading system 주어진 문제에 무엇이 적합한 시스템인가

도대체 kernel thread system 에서 kernel thread가 여러 개 존재한다는게 무슨 의미일까

4. Multithread 프로그램의 단점

서로 자원을 공유하므로 하나의 스레드에 생긴 문제가 다른 스레드에도 영향을 줄 수 있 다.

문맥이 서로 교환되므로 디버깅시에 추척하기가 까다롭다.

5. 4 conditions for deadlock to occur

1) mutual exclusive - 언제나 자원은 한 스레드만 접근 가능

2) hold and wait - 한 자원을 얻은 뒤 다른 자원을 얻기 위해 대기한다

3) circular waiting

4) no preemption - 스케줄링이 비선점방식이라 비자발적 yield가 일어날 수 없다

6. Compare\_and\_swap 으로 spin-lock 구현

acquire(lock){

while(!compare\_and\_swap(lock->value, 0, 1)){

// no-op

}

}

release(lock){

Compare\_and\_swap(lock->value, 1, 0);

}

7.Test\_and\_set 으로 spin-lock 구현

Lock->value가 0이었으면 1로 바꾸고 acquire 성공. lock->value가 1이었으면 계속 while.

acquire(lock){

while(test\_and\_set(lock->value)){

// no-op

}

}

release(lock){

lock->value = 0;

}

8. Spin-lock 은 lock을 얻지 못했을 떄 re-schedule하는 오버헤드보다 잠시 busy waiting 하는 것이 더 낫다고 예측 가능한 경우에는 써도 된다.

9. 또한, 리눅스 처럼 interrupt 가 interrupt context에서 처리 될 경우 block, sleep을 쓸 수 없으므로(핸들러가 자게되면 재앙) spin lock을 쓸 수 있다. 다만, 솔라리스에서는 interrupt handling을 위해 또 다른 thread를 사용한다. 각 interrupt마다 thread를 새로 만든다면 overhead가 매우 클 것이기 때문에 그렇게 하지 않고 interrupt threads pool을 관리하여 interrupt가 발생할 때마다 이 pool에서 thread를 하나 선택하여 그 thread가 해당 interrupt를 처리하게끔 한다. 따라서 interrupt 처리시에 일반적인 동기화 방법( semaphore, mutex 등등 blocking가능한 것들)을 사용할 수 있다.

10. Starvation vs Deadlock

Starvation - 잡으려는 자원을 무기한으로 잡지 못 할 때

Deadlock - 내가 잡은 자원을 상대가 원하고, 상대가 잡은 자원을 내가 원하면서 둘 다 자 원을 기다리는 상황. 더 일반적으로, 대기열에 있는 프로세스들이 서로 상대방의 작업이 끝나기만을 기다려 결과적으로 아무것도 완료되지 못하는 상태

11. OS의 목적과 기능

자원 분배(페이지, CPU 스케줄링, 등)

인터럽트 처리

프로세스 서로 간의 memory protection 제공

VM abstraction 제공

편리한 개발 및 유저프로그램 실행환경 제공

12. HW sync vs semaphore vs monitor

HW sync - atomic 함을 보장하므로 상위 단계의 sync primitive를 만들 때 time cost 감소

13. Critical section을 interrupt turn off/on으로 제어할 때 단점

1)멀티프로세스 환경에서 쓸 수 없다. 다른 프로세서에서 돌아가는 스레드가 CS 침범 가능

2)유저프로그램 레벨에서 제어할 수 없다.

3)real time application fails

14. Evaluation criteria on sync solution

Mutual exclusive

Progress

Bounded wait

15. Aging technique을 pintos에 적용하는 방법

Thread\_tick()에서 intr\_yield\_on\_return 다음에 ready\_list의 모든 스레드의 priority를 1씩 올려주는 코드를 작성한다. 매 thread\_tick마다 하는 것 보단 if(++thread\_ticks >= TIME\_SLICE) 조건문 안에서 intr\_yield\_on\_return과 함께 하는 것이 더 좋을 것 같다.

16. sema\_down() 코드 채우기

void sema\_down(struct semaphore\* sema){

enum intr\_level old\_level;

ASSERT (sema != NULL);

ASSERT (!intr\_context ());

old\_level = intr\_disable();

while(sema->value == 0){ // sema->value > 0 이라야 낮출 수 있다.

list\_push\_back(&sema->waiters, &thread\_current()->list\_elem);

thread\_block();

}

sema->value--;

Intr\_set\_level(old\_level);

}

17. Syscall handler 코드 채우기

(int \*)f->esp + 1, +2, +3 이 arg1,2,3이다. 당연하지만 이것은 argument의 주소이다.

\*(int \*)f->esp 는 syscall\_code 이다.

Syscall\_exit(status)는 printf(“%s: exit(%d)\n”, thread\_name(), status); 를 출력하고 thread\_exit() (내부에 process\_exit() 있음)를 부른다.

Syscall\_read에서는 fd가 0일때, input\_getc()가 0(또는 ‘/0’이라고도 씀)일 때 까지

(또는 size만큼 읽을 때까지) 문자를 인풋에서 받고 \*((uint8\_t\*)buffer+i)에 할당한다. 즉 buffer[i] = input\_getc(). 그리고 길이반환. 이때 길이는 ‘/0’까지 포함한 길이다.

Syscall\_write에서는 fd가 1이면 putbuf(buffer, size)를 이용한다.

18. Counting semaphore란

Semaphore를 얻어 CS에 접근하는 스레드의 수를 유한개로 조절할 수 있다. 초기값은 가능 한 자원의 수, 또는 접근가능한 스레드 수로 지정된다.

이진세마포어 - test\_and\_set등 하드웨어가 지원하는 동기화 기법을 이용하여 구현도 가능 (그게 lock이다)

19. User program 은 할 수 없고 kernel 이 해야만 하는 일들

1) 인터럽트 핸들링 - 인터럽트 on/off등을 유저프로그램이 할 수 있다면 악성 프로그램이 될 수 있다.

2) page allocation - 유저프로그램은 자신에게 할당된 address space 이외의 공간에 접근 불가(protection against other processes)

3) CPU scheduling

4) Creating a shared memory between multiple processes

20. Running, Ready, Blocked

Running -> Blocked - I/O(slow 하므로 이 시간에 다른 process를 돌린다) or event wait

Blocked -> Ready - I/O or event complete (Disk read operation complete)

Ready -> Running - scheduled

Running -> Ready - timer interrupt 에 의한 비자발적 CPU yield 등

21. Critical section problem 을 해결할 수 있는 방법으로 interrupt off/HW/SW/OS 각각의 장단점

22. User mode 에서 kernel mode 로 전환될 3가지 경우

23. User level thread system 이 kernel level보다 context switch의 오버헤드가 적은 이유

사용자 쓰레드 방식이 오버헤드가 적은 이유는 쓰레드 간 전환을 할 때마다 커널 스케줄러 를 호출할 필요가 없기 때문이다. 커널로 진입하려면 프로세서 모드를 사용자 모드에서 커 널 모드로 전환해야 하는데, 이때 사용자의 레지스터를 전부 저장하고, 커널의 레지스터를 전부 복구하고 기타 등등 많은 작업들이 밑에서 일어난다. 따라서 모드를 자주 바꿀 수록 성능은 떨어진다.

24. 사용자 스레드의 단점. Blocking system call

\*추가바람,

커널이 스레드의 존재를 알지 못하므로 한 스레드가 커널로 진입하는 순간 나머지 스레드 들도 전부 블록된다. 따라서 멀티프로세서 환경을 제대로 활용하지 못한다.

제대로 활용하지 못하는 이유 중 다른 하나는, 커널에서 스레드 단위로 스케줄이 안되므로 효율적으로 CPU 자원을 분배하기 힘들다.

25. 스레드 스위칭이 프로세스 스위칭보다 빠른 이유

스레드 스위칭 -> virtual address space 는 안 변함. 스레드는 유저 레벨에서 스위칭 가능. Needs only a procedure call

프로세스 스위칭 -> 캐시 메모리가 기억하고 있는 많은 address space가 새 것으로 대체된다. 따라서 locality가 파괴되어 캐시의 부스트를 받지 못한다.

26. Deadlock 시나리오

27. IPC에서 shared memory를 쓰는게 왜 message passing 보다 빠른가?

단순히 메모리에 접근하는 것과 커널 모드를 바꾸면서 정보를 전달하는 것을 비교하면 당 연히 전자가 빠르다. Kernel에 대한 의존성이 낮을수록 빠르다.

28. Shared memory vs Message passing

Communication 주체 - 자동적으로 알아서/ 프로그래머가

Synchronization - Producer-consumer problem/ 전달한 내용이 도착하기 전에 또 신호주는 문제