과제3: dining philosophers problem

과목명 : 운영체제(공통반)

교수명 : 양승민 교수

학과 : 산업정보시스템공학과

학번 : 20131847

이름 : 박호정

제출일자 : 2019.11.9

목차

1. 관련연구

1.1. 세마포어 동기화 기법

1.2. pthread API 함수 조사

1. 문제 해결 방법
2. 추가 및 개선할 부분과 결론

1.관련연구

1.1. 세마포어 및 동기화 기법

공유되는 자원이 존재하는 경우 process가 서로 자원을 사용하려는 경쟁을 펼친다. 이와 같은 상황으로 원하지 않는 결과가 나올 수 있다. 이를 race condition이라 한다. 이를 해결하기 위한 방법이 synchronization 즉, 동기화 기법이다.

동기화 기법에는 synchronization hardware, mutex locks, semaphores이 있다. synchronization hardware는 여러 개의 명령어를 automic하게 실행해주는 hardware를 말한다. 대표적인 명령어로는 test\_and\_set이 있는데 여러 줄의 연산을 하나의 트랜잭션으로 처리되도록 하드웨어에서 지원해준다 이런 hardware를 통해서 critical section을 순서대로 실행할 수 있다.

다음은 mutex locks이다. 이는 critical section 시작 전에 lock을 걸고 끝나면 unlock 해주는 방식이다. lock 권한을 획득할 때까지 busy waiting 하는 것이 특징이다. lock을 얻기 까지 기다리는 시간이 적을 때 사용하는 것이 유용하다.

마지막으로 semaphores이다. mutex lock은 오직 하나의 process가 critical section에 들어갈 수 있다. 하지만, mutex로는 동시에 2개 이상의 자원이 존재하여도 1개의 자원밖에 사용하는 process라면 자원 1개가 낭비된다. 이 같은 경우에는 semaphores를 사용하면 된다. semaphores는 사용할 수 있는 자원을 초기 값으로 가진다. 이를 통해, 사용 가능한 자원이 0개가 될 때까지 요청이 들어온 process에게 할당해준다.

1.2. pthread API 함수 조사

구현시 사용한 함수에 대해서 설명하도록 한다. 다음과 같은 순서로 설명한다.

pthread\_create

pthread\_cancel

pthread\_mutex\_init

pthread\_mutex\_lock

pthread\_mutex\_unlock

sem\_init

sem\_wait

sem\_post

int pthread\_create

(pthread\_t \* thread, const pthread\_attr\_t \* attr, void \*(\*start\_routine)(void\*), void \* arg);

-인자 설명

thread : 생성된 스레드 id가 저장될 주소

attr : 스레드 속성

start\_routine : 스레드 시작 함수

arg : 스레드 시작 함수의 인자

-리턴 값

성공 시 0, 실패 시 오류 값 반환

-오류 값

EAGAIN : 자원 부족으로 thread 생성 불가

EINVAL : thread 속성 객체가 유효하지 않음

EFAULT : thread id 반환 주소나 시작 함수가 NULL로 설정됨.

-설명

attr이 NULL이면 기본 thread 속성으로 생성한다. 성공적으로 생성되면 생성된 thread id는 thread에 저장된다.

생성된 thread는 arg인자를 사용하여 start\_routine을 실행한다. start\_routine이 리턴되면 내부적으로 pthread\_exit()을 호출되어 thread가 종료된다.

int pthread\_cancel (pthread\_t thread);

-인자 설명

thread: 취소할 thread

-리턴 값

성공 시 0, 실패 시 오류 값 반환

-오류 값

ESRCH : 인자 thread가 유효하지 않음

-설명

thread 실행을 취소하도록 요청한다. thread의 취소 속성과 종류에 따라 처리가 된다. 실행 취소가 결정되면 취소 처리 함수들이 수행된다. 취소 함수들이 다 처리되면 사용하면 thread를 위한 자원들을 회수한다.

int pthread\_mutex\_init (pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t attr);

-인자 설명

mutex: 뮤텍스 객체

attr : NULL 또는 뮤텍스 속성 객체의 주소

-리턴 값

성공 시 0, 실패 시 오류 값 반환

-오류 값

EINVAL : mutex가 NULL일 경우

EBUSY: mutex가 이미 초기화된 경우

EINVAL: attr이 초기화되지 않은 경우

-설명

mutex를 attr에 따라 초기화한다. attr이 NULL이면 시스템 기본 속성으로 초기화한다. 초기화된 mutex는 잠겨있지 않다.

int pthread\_mutex\_lock(pthread\_mutex\_t \* mutex);

-인자 설명

mutex: 뮤텍스 객체

-리턴 값

성공 시 0, 실패 시 오류 값 반환

-오류 값

EINVAL : mutex가 NULL이거나 유효하지 않은 mutex 객체인 경우

EDEADLK: 이미 잠금을 얻은 쓰레드가 다시 잠금을 요청하는 경우

EBUSY: 뮤텍스가 잠겨 있어서 잠금을 얻을 수 없는 경우

-설명

호출한 thread가 mutex를 잠그도록 한다. 이미 잠겨있는 mutex라면 호출한 thread는 잠금이 해지될 때까지 기다리다가 mutex를 획득한다.

int pthread\_mutex\_unlock(pthread\_mutex\_t \* mutex);

-인자 설명

mutex: 뮤텍스 객체

-리턴 값

성공 시 0, 실패 시 오류 값 반환

-오류 값

EINVAL: mutex가 NULL이거나 유효하지 않은 mutex 객체인 경우

EPREM: 호출한 thread가 해당 mutex를 잠그지 않았던 경우

-설명

muext잠금을 해지한다. mutex 객체의 타입에 따라 잠금 해지 방법이 다르다.

int sem\_init(sem\_t \* sem, int pshared, unsigned value);

-인자 설명

sem : 초기화할 세마포어

pshared: 프로세스간 공유 여부

value : 초기화된 세마포어에 할당하는 값

-리턴 값

성공 시 0, 실패 시 -1 반환하고 errono값 설정

-오류 값

EINVAL: value 인자 값이 {SEM\_VALUE\_MAX} 초과하는 경우

ENOSPC: 세마포어 초기화하기 위한 자원이 불충분할 경우 혹은 세마포어 개수가 {SEM\_NSEMS\_MAX}에 도달한 경우

EPERM: 세마포어 초기화 권한이 없는 경우

-설명

세마포어를 초기화 할 때 사용하는 함수이다. 세마포어의 값은 value에 저장한다. pshared가 0이 아니면 프로세스들 간에 공유할 수 있다. pshared가 0이라면 프로세스 내의 스레드들간에 세마포어를 공유할 수 있다.

int sem\_wait(sem\_t \*sem);

-인자 설명

sem: 잠궈야 할 세마포어

-리턴 값

성공 시 0, 실패 시 -1 반환하고 errno를 설정해 오류 표시

-오류 값

EAGAIN: 세마포어가 이미 잠겨 있어서 즉시 잠글 수 없는 경우

EDEADLK: 데드락 상태가 감지되는 경우

EINTR: 시그널이 함수수행을 중단하는 경우

EINVAL: sem이 유효하지 않은 semaphore인 경우

-설명

세마포어의 값이 양수이면 세마포어 하나를 획득하고 그렇지 않으면 기다린다.

int sem\_post(sem\_t \* sem);

-인자 설명

sem: 잠금 해제할 세마포어

-리턴 값

성공 시 0, 실패 시 -1 반환하고 errno를 설정해 오류 표시

-오류 값

EINVAL: sem이 유효하지 않은 세마포어인 경우

-설명

semaphore를 되돌려준다. semaphore의 값을 1증가시킨다.

2. 문제해결 방법

Dining philosophers은 각 철학자가 식사를 할 때2개의 포크를 집어야 한다. 따라서, 철학자 한 명당 thread 하나를 생성하고 각 포크는 mutex 배열을 사용하여 매칭하였다. 또한 2개의 리소스를 가지는 과정에서 deadlock이 발생할 수 있다. deadlock 해결 방법으로는 prevention방법 중 필요한 자원을 모두 잡아서 deadlock이 생기지 않도록 하였다. 또한, synchronization을 위해 mutex lock과 semaphores를 사용하도록 하였다.

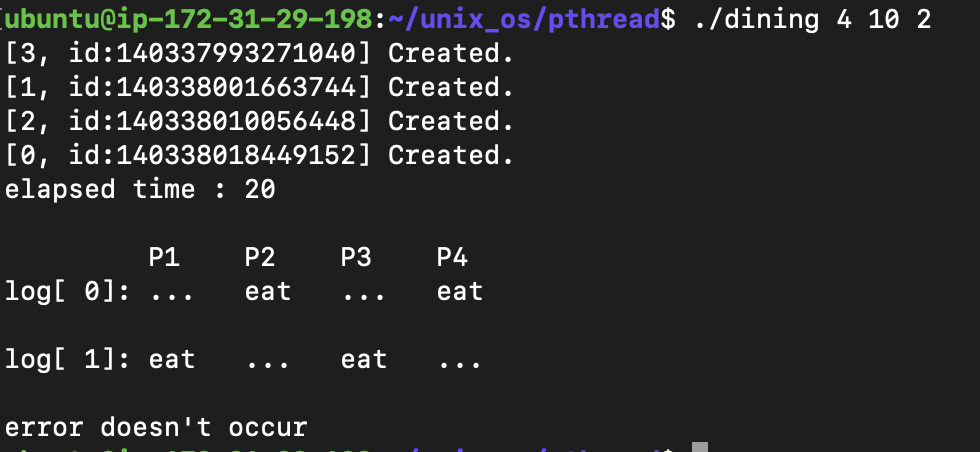
입력 값으로 주어지는 철학자의 수로 결정되는 것은 생성할 thread 수와 semaphores 값이다. 철학자의 수를 n이라고 하면 생성할 thread 수는 n이며 semaphores의 초기 값은 n/2(내림)이다. 즉, 철학자의 수가 5이면 5개의 thread를 생성하고 semaphores의 초기 값은 2로 할당한다. Semaphores의 초기 값을 n/2로 한 이유는 deadlock해결 기법인 prevention을 이용하였기 때문이다.

메인함수에서 입력 값에 대한 예외 처리와 mutex, semaphores 변수 값의 초기화, thread의 생성을 진행한다. 이후 for loop을 사용하여 number\_of\_cycle값만큼 sleep(number\_of\_msec)을 수행해 준다. 한 번의 반복 마다 시간대를 변경하는데 이 정보는 출력을 위한 정보이다. 전역 변수로 시간대에 대한 정보를 관리하고 이를 각 thread에서 접근하여 출력 정보를 저장하는 것이다. Number\_of\_cycle번의 sleep을 거치고 나서 출력 정보를 간단히 출력하고 난 후 동일 시간대에 포크를 집을 수 없었던 상황이 있었는지 체크하는 함수를 호출하고 종료한다.

thread가 실행되면 각 thread는 semaphore 값이 1 이상이어야 critical section으로 들어갈 수 있다. 즉 사용할 수 있는 포크의 개수가 2개 이상이 있어야 진행이 가능하다. 이후 2개의 포크를 집는 과정에서 race condition이 발생할 수 있다. 2개의 포크를 잡는 것이 atomic하게 이루어져야 한다. 따라서 2개의 포크를 잡는 과정을 critical section으로 보고 mutex lock을 사용하였다. 이 과정 까지 마치면 sleep(number\_of\_msec)을 통해 eating의 과정을 수행하고 semaphores와 사용한 mutex를 release한다. 이렇게 하여 각 thread간 deadlock이 없는 동기화를 수행할 수 있다. 이후 thinking을 위한 sleep을 하고 위 과정을 반복한다. 출력을 위하여Eating직전에 기록을 하였다. 현재 thread의 pid를 인자 값으로 받아 두었다. 이 값을 배열의 현재 시간대에 저장하여 결과 출력 시 사용되도록 하였다. 배열에 eating의 내용이 없으면 thinking하고 있는 것으로 판단한다. 시간대의 관한 정보는 main함수에서 관리한다.

다음은 ./dining 4 10 2의 출력 결과이다.

초기에 쓰레드를 생성하고 간단히 결과를 출력한다. 마지막 라인에는 error가 없는지 체크하고(바로 옆사람이 식사하는 경우를 체크) 종료한다.



3.추가 및 개선할 부분과 결론

추가로 고려해볼 부분은 쓰레드의 생성 과정에 따라서 eating-thinking pattern이 나타나는 경향이 있다. 이는 eating의 시간이 정해져 있고 os의 scheduling 방식의 영향을 받아서 나타난다. 따라서, 동일한 입력에 대해서 다양한 eating-thinking 결과를 만들어 내기 위해서 실행시마다 랜덤하게 thread 생성을 하는 방식을 채택하는 것도 고려해볼 부분이다. 본 과제에서 구현 방식은 홀수와 짝수 그룹으로 나누어 생성하였는데 최대한 많은 철학자가 식사할 수 있는 방식이다.

개선할 부분은 다른 deadlock 해결 기법과의 비교를 통해 더 오버헤드가 적으면서 더 효율적인 작동을 하는 기법을 확인할 필요성이 있다. 결과적인 관점에서는 의도한 결과가 나왔지만 이 문제에 대하여 최적의 성능을 낼 수 있는 기법을 적용할 필요가 있다. 또한, 시간 단위에 대한 기록 과정에서 elapsed time의 0.5% 이내의 추가 시간이 발생하였다. 2000ms인 경우 10ms 이내의 추가 시간이 발생하였다. 이를 없앨 수 있도록 디버깅할 필요가 있다.

이번 과제를 통해 mutex lock, semaphore를 이해하고 실제 문제를 통해 해결하는 시간이 되었다. multi-thread 환경에서 race condition이 발생할 수 있음을 인지함으로서 앞으로 multi-thread program을 구현할 때 많은 도움이 될 것이다. 또한, 다양한 deadlock 기법에 대한 인지와 과제2에서 수행했던 시뮬레이션을 토대로 효율적인 deadlock없는 synchronization을 구현할 수 있을 것이다.