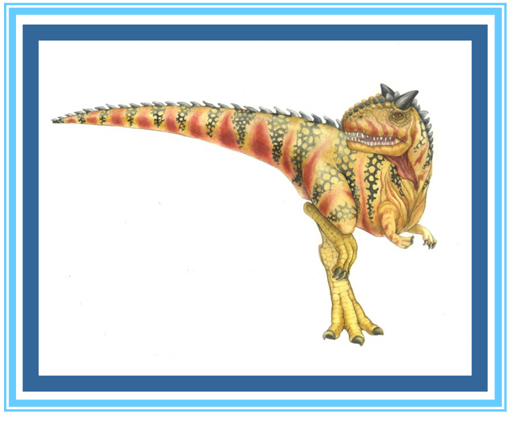
Chapter 7: Synchronization Examples



Chapter 7: Synchronization Examples

- Explain the bounded-buffer, readers-writers, and dining philosophers synchronization problems.
- Describe the tools used by Linux and Windows to solve synchronization problems.
- Illustrate how POSIX can be used to solve process synchronization problems.

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

counter 변수 없이 mutex만 사용해서 문제를 해결할 수 없다. Counting semaphore인 full, empty를 사용하는 것이 효율적이다.

세마포의 값 = 리소스의 개수

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
         ...
        /* add next produced to the buffer */
         ...
    signal(mutex);
    signal(full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full);
    wait(mutex);
        ...
        /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
        /* consume the item in next consumed */
        ...
}
```

생산자와 소비자가 별도의 뮤텍스락을 사용하는 것이 더 효율적이다.

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do ***not*** perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Reader와 writer의 상호배타용 세마포

Reader가 read_count를 업데이트하기 위해 사용하는 mutex 세마포

현재 진행 중인 reader의 수

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
        signal(rw_mutex);
}
```

오직 한 개의 writer만
들어갈 수 있어야 한다.

이 해는 multiple reader를 허용하지만
writer가 지속적으로 굶을 수 있다. 이것을
우리는 “reader 선호” 알고리즘이라 부른다.

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);

        ...
        /* reading is performed */
        ...
        wait(mutex);
        read count--;
        if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);
}
```

자신이 유일한 reader인가?
그렇다면 writer와 경쟁,
그렇지 않다면 다른 reader가
들어오는 것을 허용함.

reader가 중복해서
들어오는 것을 허용함.

더 이상 reader가 없으므로
writer을 허용함.

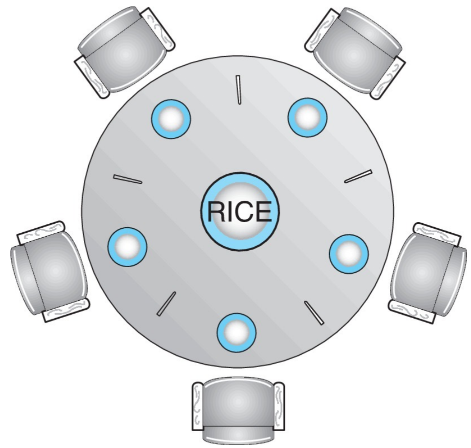
Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object (reader 선호)
- ***Second*** variation – once writer is ready, it performs the write ASAP (writer 선호)
- Both may have starvation leading to even more variations

Writer 선호: reader의 중복을 최대한 허용하되 기다리는 writer가 있으면 reader가 더 이상 writer를 앞지르지 못하게 하는 방식이다. 늦게 온 writer는 기다리고 있는 reader를 앞지를 수 있다.

공정한 reader-writer: 선착순으로 CS에 들어가면서 reader의 중복을 최대한 허용하는 방식이다. 이 방식에서는 늦게 온 reader/writer가 기다리고 있는 다른 reader/writer를 앞지르지 못한다.

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher *i*:

```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

     /* think for awhile */

}
```

Deadlock을 피하는 3가지 방법

① 테이블에 최대 4명만 앉는다.
② 젓가락 두 개를 다 집을 수 있을 때만 집는다.
③ 짝수번 철학자는 왼쪽, 오른쪽 순서로, 홀수번 철학자는 오른쪽, 왼쪽 순서로 집는다.

- What is the problem with this algorithm?

deadlock 가능

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
                   // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
   }
```

배고프지만 젓가락을 집을 수 없을 때 기다리기 위해 사용하는 조건 변수

좌우에 누가 먹기를 기다리고 있으면 깨워줌

Solution to Dining Philosophers (Cont.)

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
                 (state[i] == HUNGRY) &&
                 (state[(i + 1) % 5] != EATING)) {
                    state[i] = EATING;
                    self[i].signal();
            }
    }

    initialization_code() {
            for (int i = 0; i < 5; i++)
                    state[i] = THINKING;
    }
}
```

i의 왼쪽과 오른쪽 철학자
모두가 식사 중이 아니면 i는
식사가 가능 → i를 식사
중으로 설정 → i를 깨움

Solution to Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

    /** EAT **/

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Kernel Synchronization - Windows

Kernel level 동기화 메커니즘

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted

효율성 때문이다. 예를 들면 spinlock을 가진 스레드가 preempt되면 deadlock이 발생할 수 있다.

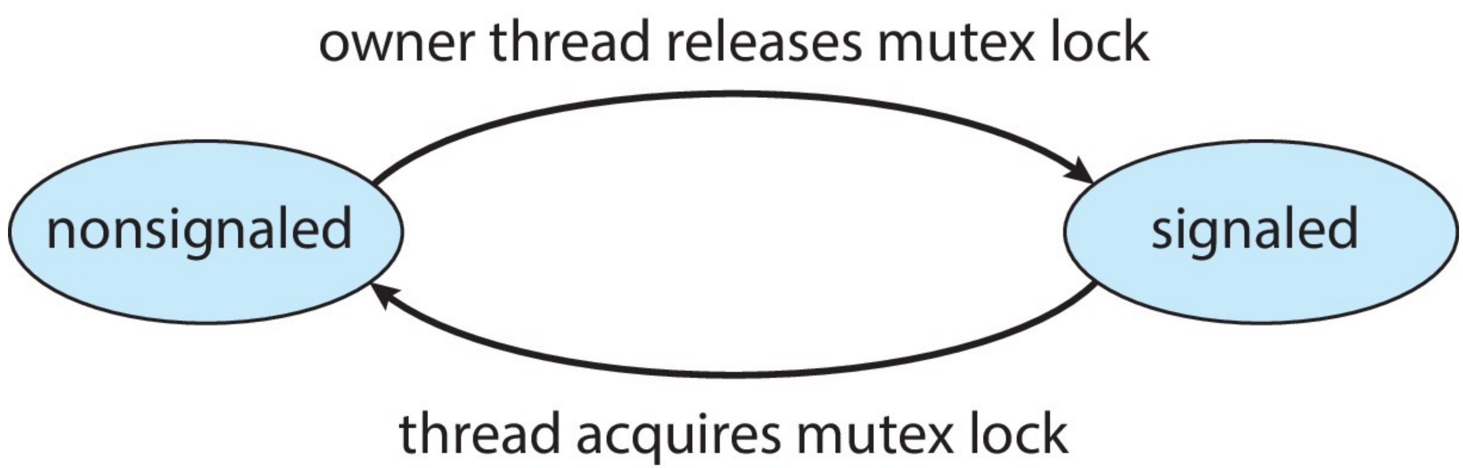
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events** → 어떤 조건을 만족하면 기다리던 스레드에게 notify()함
    - An event acts much like a condition variable
  - **Timers** notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

User level 동기화 메커니즘

Dispatcher object마다 waiting queue가 있다. Object가 signaled-state로 바뀌면 queue에 대기하던 모든 스레드 또는 일부를 깨운다.

Kernel Synchronization - Windows

- Mutex dispatcher object



Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - atomic integers
  - spinlocks
  - mutex lock

Linux의 spinlock과 mutex lock은 nonrecursive하다. 즉, lock을 가진 상태에서 다시 lock을 걸 수 없다. (cf, reentrant lock)

- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

그러나 SMP에서는 spinlock 사용

Kernel에 있는 task가 lock을 가지고 있으면 이 task는 nonpreemptive하다 (preemtive하면 교착상태가 발생할 수 있기 때문). 현재 가지고 있는 lock의 수를 preempt_count라는 변수에 저장한다. 이 값이 0이면 preemption이 가능.

Linux Synchronization

- Atomic variables

  `atomic_t` is the type for atomic integer

- Consider the variables

```
atomic_t counter;
int value;
```

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

지금까지 언급한 Linux 동기화 기법은 커널 개발자가 사용하는 것이고,
사용자 수준에서 사용할 수 있는 동기화 기법은 다음에 있다.

Atomic library functions (C11)

void **atomic_init**(_Atomic(T) *object, T value); /* non-atomically initialize */

T **atomic_load**(_Atomic(T) *object);

void **atomic_store**(_Atomic(T) *object, T desired);

T **atomic_exchange**(_Atomic(T) *object, T desired);

_Bool **atomic_compare_exchange_strong**(_Atomic(T) *object, T *expected, T desired);

_Bool **atomic_compare_exchange_weak**(_Atomic(T) *object, T *expected, T desired);

T **atomic_fetch_add**(_Atomic(T) *object, T operand);

T **atomic_fetch_and**(_Atomic(T) *object, T operand);

T **atomic_fetch_or**(_Atomic(T) *object, T operand);

T **atomic_fetch_sub**(_Atomic(T) *object, T operand);

T **atomic_fetch_xor**(_Atomic(T) *object, T operand);

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
```

POSIX Named Semaphores

- Creating an initializing the semaphore:

관련 없는 프로세스
사이에서 동기화할 때 사용

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

접근권한으로 rw 권한이
있어야 다른 프로세스가
사용할 수 있다

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

0: 같은 프로세스 내에 있는 스레드 사이에서 공유함. 변수 sem은 모든 스레드가 보이는 곳에 있어야 함.

Non-zero: 프로세스 사이에서 공유함. 변수 sem은 shared memory 공간에 있어야 함.

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX의 조건변수는 상호배타가 필요함

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

POSIX Condition Variables

- Thread waiting for the condition **a == b** to become true:

Pthread_cond_wait()는 mutex lock을 풀기 때문에 리턴되었을 때 a나 b의 값이 변경되어 있을 수 있다. 따라서 while 루프를 사용하여 조건을 다시 검사하는 것이 매우 중요함.

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

① release mutex lock
② wait on the condition variable
③ 깨어나면, acquire mutex lock, and return

- Thread signaling another thread waiting on the condition variable:

a=b를 수행하기 위해 걸었던 lock을 signal을 호출한 후 unlock을 하게 되면 깨어났지만 아직 대기상태에 있던 스레드가 lock을 획득한 후 다음을 진행할 수 있다.

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

조건변수에서 기다리고 있는 스레드가 있으면 깨우고 나온다. 이 때 깨어난 스레드는 mutex lock을 획득해야 하므로 이 예제의 경우 아직 진행이 정지된 상태이다.

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- Consider a function update() that must be called atomically. One option is to use mutex locks:

전통적인 방식

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding **`atomic{S}`** which ensure statements in **`S`** are executed atomically:

프로그래밍 언어에 추가된 기능으로 atomic{S}하면 S를 트랜잭션으로 실행하라는 뜻.

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

트랜잭션은 모든 연산이 올바르게 처리되어 commit(확정)되거나, 또는 취소되서 원점으로 롤백하는 두 가지만 가능한다.

OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
#pragma omp parallel
{
     void update(int value)
      {
           #pragma omp critical
           {
                count += value
           }
       }
 }
```

코어의 수만큼 스레드 생성 후, 병렬로 실행하라.

Atomically 실행하라.

The code contained within the **#pragma omp critical** directive is treated as a critical section and performed atomically.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

End of Chapter 7