Dining-Philosophers Problem – 세마포 사용

Semaphore chopStick[5] = {1,1,1,1,1} ; // initially all values are 1

- Philosopher **i**

```
while (true) {
    P(chopStick[i]);            // get left chopstick
    P(chopStick[(i+1) % 5]);    // get right chopstick
    … eat
    V(chopStick[i]);            // return left chopstick
    V(chopStick[(i+1) % 5]);    // return right chopstick
    … think
};
```

- 이 코드는 교착상태의 가능성이 있음(not deadlock-free)
  - 모든 철학자가 왼쪽 젓가락을 집으면 ? → 오른쪽 젓가락을 무한히 대기
  - 해결책: (1) 젓가락 두 개를 모두 집을 수 있을 때에 집게 함
    (2) 홀수철학자는 왼쪽 먼저, 짝수 철학자는 오른쪽 먼저 집는다.
    (3) 최대 4명의 철학자가 앉는다.

6.8 모니터(Monitors)

- Semaphore 사용의 문제점
  - 프로그래머가 잘못 사용하면 다양한 유형의 오류 발생 가능
- 잘못된 Semaphore 사용 예
  - P와 V 연산의 순서를 반대로 사용

```
V(mutex)
critical section   → 상호배제 위반
P(mutex)
```

  - V대신에 P를 사용

```
P(mutex)
critical section
P(mutex)           → deadlock
```

  - P 또는 V를 누락

```
deadlock ← P(mutex)
           critical section

critical section   → 상호배제 위반
V(mutex)
```

  - 이러한 실수가 아니어도 동작의 정확성을 증명하기 어렵고 식사하는 철학자 문제와 같이 교착상태를 유발할 수 있다.

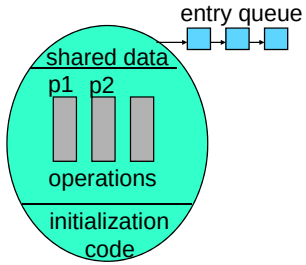
모니터(Monitor)

- 병행 프로그래밍(Concurrent Programming)을 위한 지원
  - 병행 프로그래밍이 순차 프로그래밍보다 어려움
  - 세마포와 같은 기능을 직접 사용하지 않고 병행 프로그래밍을 위해 **고수준의 상호배제를 지원하는 기능**이 개발됨
  - 잘못 사용할 위험성을 줄임
- 모니터(Monitor)
  - 프로세스 동기화를 위한 편리하고, 효율적이고, 안전한 방법을 제공하는 고수준의 동기화 추상화 데이터 형(ADT) – data와 procedure를 포함
  - monitor에서 정의된 procedure만에 monitor 내의 local data를 접근가능
  - monitor의 procedure들은 한 번에 하나의 프로세스만 진입하여 실행할 수 있음 → 상호배제 보장

Monitor 구문

- Monitor 구문

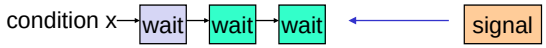
```
monitor monitor_name
{
   // shared variable declarations
   procedure p1(…) {
            …
   }
   procedure p2(…) {
            …
   }
            …
   initialization code (...) {
            …
   }
}
```



- monitor를 지원하는 프로그래밍 언어
  - Concurrent Pascal, Ada, Mesa, Concurrent Euclid, Modula-3, D, C#
  - Java, Python …

조건 변수(Condition Variables)

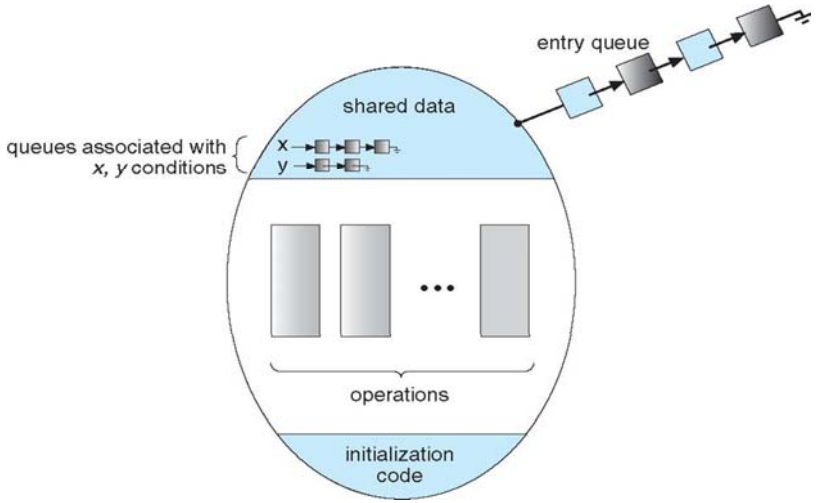
- Condition: 모니터에서 **동기화**에 사용되는 특별한 자료형
  condition x, y;
- Condition 변수를 사용하는 연산 - wait, signal
  wait(), signal()
  - **wait(x)** 를 호출한 연산은 프로세스는 다른 프로세스가 signal(x)을 호출할 때까지 일시 중지(suspend)됨
  - **signal(x)** 연산은 정확히 하나의 프로세스만 재개시킴. 일시 중지된 프로세스가 없으면, 아무런 영향이 없음



(cf) 세마포의 V 연산은 세마포 값에 영향을 줌

- 모니터 내에서의 동기화
  - 조건변수에 대한 두 연산(wait, signal)을 사용하여 수행함

condition 변수를 갖는 모니터



Dining Philosophers 문제 – monitor 사용

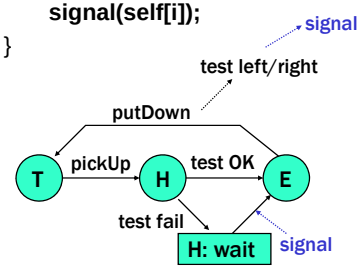
```
monitor diningPhilosophers {          // pseudo-code
    int state[5];                     // THINKING(0), HUNGRY(1), EATING(2)
    condition self[5];                // condition variables
     diningPhilosophers {    // initialization code
          for (int i = 0; i < 5; i++)
             state[i] = THINKING;
    }
    procedure pickUp(int i) { /* see next slides */ }
    procedure putDown(int i) { /* see next slides */ }
    private test(int i) { /* see next slides */ }
}
```

- 양쪽 젓가락 모두 사용 가능할 때에만 양쪽 젓가락을 집고, 그렇지 않으면 wait 상태가 됨
- 젓가락을 내려놓을 때에 옆의 철학자의 상태를 확인하여 철학자가 HUNGRY상태에 있고, 양쪽 젓가락 사용이 가능하면 signal을 보내 wait 상태를 해제함

Dining Philosophers 문제 – monitor 사용 (계속)

```
procedure pickUp(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        wait(self[i]);
}
procedure putDown(int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
}
```

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



- 이 해결안은 deadlock-free이지만 starvation-free는 아님

Dining Philosophers 문제 – monitor 사용 (계속)

```
// initially,
monitor dp = new diningPhilosophers();
```

- philosopher i

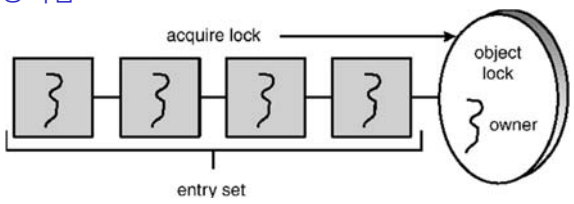
```
while (true) {
    dp.pickUp(i);
    eat();
    dp.putDown(i);
    think();
}
```


Java Monitors - Java Synchronization

- Synchronized method
- wait(), notify() statements
- Multiple Notifications: notifyAll()

synchronized Statement

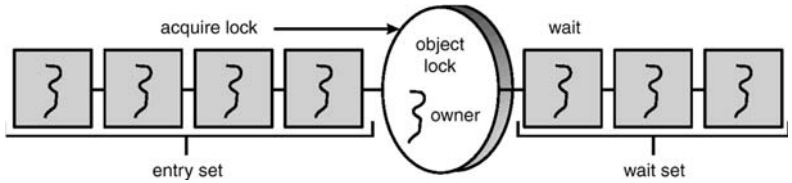
- Synchronized method
  - Java의 모든 object는 자신과 연관된 lock을 가짐
  - **synchronized method** 호출은 lock의 소유를 필요로 함
  - object에 대한 lock을 소유하지 않은 thread가 object의 synchronized method를 호출하면 object lock에 대한 entry set에 놓임
  - synchronized method를 사용하는 thread가 method 사용을 종료하면 lock을 해제함
- 공유 데이터에 대한 병행 접근을 직렬화(serialize)하여 경쟁조건 발생을 방지함



wait()와 notify() Method

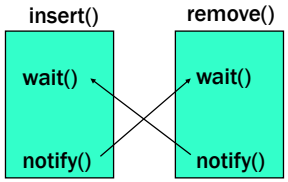
- wait() method
  - object lock을 해제하고 thread를 Block상태로 설정하고 wait set에 놓임
- notify() method
  - wait set에서 임의의 thread T를 선택하여 entry set으로 이동하고 thread를 Runnable상태로 설정

    ➔ thread *T* 는 다시 object lock을 얻으려고 경쟁할 수 있음
- Java의 wait(), notify()는 모니터의 wait(), signal()과 유사하지만 condition variable은 사용하지 않음 (object 당 1개의 조건변수를 사용하는 것과 같음)



wait/notify methods를 사용한 insert()와 remove()

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE)            // full
        try { wait(); } catch (InterruptedException e) { }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    notify();       // notify threads waiting for non-empty buffer
}
public synchronized Object remove() {
    Object item;
    while (count == 0)        // empty
        try { wait(); }
        catch (InterruptedException e) { }
    - - count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();       // notify threads waiting for non-full buffer
    return item;
}
```

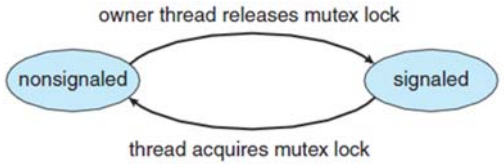


6.9 동기화(Synchronization) 사례

- Windows Synchronization
- Linux Synchronization
- Solaris Synchronization
- pthread library

Windows Synchronization

- Window kernel의 global resources에 대한 접근 보호
  - 단일 프로세서 시스템 – 일시적으로 interrupt mask
  - 다중 프로세서 시스템 – spinlock 사용
    - spinlock을 가지고 있는 동안 선점되지 않음
- Kernel 외부의 thread 동기화
  - **dispatcher objects** 제공 - mutex, semaphores, events(조건변수)



  - **critical-section object** – user-mode mutex, kernel 개입 없음
    - 다중 프로세서 시스템에서, 처음에 spinlock 사용.
      오랫동안 기다리면 kernel mutex를 할당하고 block됨
    - 경쟁이 있을 때만 kernel mutex가 할당되어 효율적

Linux Synchronization

- Linux kernel에서의 동기화
  - atomic 정수 연산 제공

```
atomic_t counter;
int value;

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 */
```

  - mutex locks, semaphore, reader-writer locks
  - spinlocks (다중 프로세서 시스템에서) } 짧은 시간만 유지될 때 사용
  - disable/enable kernel preemption

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Solaris Synchronization

- Solaris Synchronization
  - **적응적(adaptive) mutex** : spinlock 또는 sleeping lock
    - lock을 다른 CPU에서 실행 중인 thread가 소유 → spinlock 사용
    - lock을 소유한 thread가 수행 중이 아니면 → lock이 방출될 때까지 sleep (sleeping lock)
    - 짧은 코드 세그먼트(수 백개 이하의 명령어 사용)에서 접근되는 데이터를 보호하기 위해서 사용
  - condition variables, semaphores
    - 긴 코드 세그먼트 용으로 사용
  - reader-writer locks
    - 빈번히 사용되지만 주로 read-only로 접근되는 데이터 보호에 사용
  - turnstile – lock을 대기하는 상태에 있는 thread들의 큐
    - 객체가 아닌 커널 thread 마다 한 개의 turnstile을 가지게 함

Pthread Synchronizations

- Pthreads 동기화
  - Pthreads API는 사용자 수준에서 프로그래머에게 제공.
  - 운영체제에 독립적 API
- Pthread 동기화 기능
  - mutex locks
  - condition variables
  - non-portable extensions – semaphore, read-write lock, spin locks

```
#include <pthread.h>

pthread_mutex_t mutex;

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

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

/* critical section */

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


Pthread mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t   *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- pthread_mutex_t
  - mutex (**mut**ual **ex**clusive lock) type
- pthread_mutex_init
  - initialize mutex (**mut**ual **ex**clusive lock)
  - mutexattr: attributes (usually NULL)
- alternative intialization code

```
pthread_mutex_init(&mutex, NULL);
mutex = PTHREAD_MUTEX_INITIALIZER;      /* same code */
```


Example: job-queue2.c

```
#include <malloc.h>
#include <pthread.h>

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

initialize mutex variable

```
/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Now it's safe to check if the queue is empty.  */
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* Get the next available job.  */
      next_job = job_queue;
      /* Remove this job from the list.  */
      job_queue = job_queue->next;
    }
    /* Unlock the mutex on the job queue because we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);
```

lock()

unlock()


```
    /* Was the queue empty?  If so, end the thread.  */
    if (next_job == NULL)
      break;

    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```


Pthread semaphore

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

- sem_t
  - semaphore type
- sem_init
  - intialized a semaphore object
    - sem: pointer to the semaphore
    - pshared: a sharing option (0: local)
    - value: initial value to the semaphore
- sem_post: V operation (atomically increment)
- sem_wait: P operation (atomically decrement)

job-queue3.c

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```c
/* Perform one-time initialization of the job queue.  */

void initialize_job_queue ()
{
  /* The queue is initially empty.  */
  job_queue = NULL;
  /* Initialize the semaphore which counts jobs in the queue.  Its
     initial value should be zero.  */
  sem_init (&job_queue_count, 0, 0);
}
```



```c
/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Wait on the job queue semaphore.  If its value is positive,
       indicating that the queue is not empty, decrement the count by
       1.  If the queue is empty, block until a new job is enqueued.  */
    sem_wait (&job_queue_count);

    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty.  Get
       the next available job.  */
    next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Unlock the mutex on the job queue because we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);
```



```c
      /* Carry out the work.  */
      process_job (next_job);
      /* Clean up.  */
      free (next_job);
    }
    return NULL;
  }

  /* Add a new job to the front of the job queue.  */

  void enqueue_job (/* Pass job-specific data here...  */)
  {
    struct job* new_job;
```



```c
    /* Allocate a new job object.  */
    new_job = (struct job*) malloc (sizeof (struct job));
    /* Set the other fields of the job struct here...  */

    /* Lock the mutex on the job queue before accessing it.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Place the new job at the head of the queue.  */
    new_job->next = job_queue;
    job_queue = new_job;

    /* Post to the semaphore to indicate that another job is available.  I
       threads are blocked, waiting on the semaphore, one will become
       unblocked so it can process the job.  */
    sem_post (&job_queue_count);

    /* Unlock the job queue mutex.  */
    pthread_mutex_unlock (&job_queue_mutex);
  }
```

Pthread condition variables

```
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int  pthread_cond_init(pthread_cond_t  *cond,
                                        pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t  *cond, pthread_mutex_t *mutex);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- pthread_cond_init
  - initialize a condition variable
  - cond_attr: usually NULL
  - a condition variable must always be associated with a mutex


```
pthread_cond_t cv;
pthread_mutex_t mutex;
...
pthread_mutex_init (&mutex, NULL);
pthread_cond_init(&cv, NULL);
...
pthread_mutex_lock(&mutex)
pthread_cond_wait( &cv, &mutex);
pthread_mutex_unlock(&mutex)
...
pthread_mutex_lock(&mutex)
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mutex)
```