Synchronization

경희대학교 컴퓨터공학과

조 진 성

Synchronization

- Threads cooperate in multithreaded programs
  - ✓ To share resources, access shared data structures
  - ✓ Also, to coordinate their execution

- For correctness, we have to control this cooperation
  - ✓ Must assume threads interleave executions arbitrarily and at different rates
    - Scheduling is not under application writers' control
  - ✓ We control cooperation using synchronization
    - Enables us to restrict the interleaving of execution
  - ✓ (Note) This also applies to processes, not just threads
    - And it also applies across machines in a distributed system

An Example

- Withdraw money from a bank account
  - ✓ Suppose you and your girl(boy) friend share a bank account with a balance of 1,000,000won
  - ✓ What happens if both go to separate ATM machines, and simultaneously withdraw 100,000won from the account?

```
int withdraw (account, amount)
{
   balance = get_balance (account);
   balance = balance - amount;
   put_balance (account, balance);
   return balance;
}
```

An Example (Cont'd)

- Interleaved schedules
  - ✓ Represent the situation by creating a separate thread for each person to do the withdrawals
  - ✓ The execution of the two threads can be interleaved, assuming preemptive scheduling:

Execution sequence as seen by CPU

```
balance = get_balance (account);
balance = balance - amount;
```

Context switch

```
balance = get_balance (account);
balance = balance - amount;
put_balance (account, balance);
```

Context switch

```
put_balance (account, balance);
```

Synchronization Problem

- Problem
  - ✓ Two concurrent threads (or processes) access a shared resource without any synchronization
  - ✓ Creates a race condition
    - The situation where several processes access and manipulate shared data concurrently
    - The result is non-deterministic and depends on timing
  - ✓ Critical section
    - Code segment in which the shared data is accessed
  - ✓ We need mechanisms for controlling access to critical sections in the face of concurrency
    - So that we can reason about the operation of programs
  - ✓ Synchronization is necessary for any shared data structure
    - buffers, queues, lists, etc.

Another Example: Bounded Buffer

- No synchronization

Producer

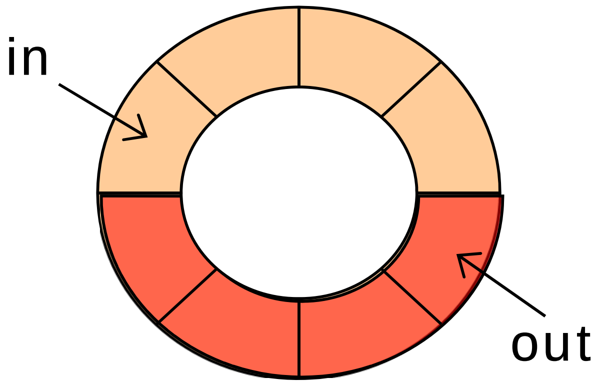
```
void producer(data)
{

 while (count==N) ;
 buffer[in] = data;
 in = (in+1) % N;
 count++;

}
```

```
int count;
```

```
struct item buffer[N];
int in, out;
```



Consumer

```
void consumer(data)
{

 while (count==0) ;
 data = buffer[out];
 out = (out+1) % N;
 count--;

}
```

Another Example: Bounded Buffer

- No synchronization: synchronization problem
  - ✓ The statement "**count++**" may be implemented in machine language as:

```
register1 = count
register1 = register + 1
count     = register1
```

  - ✓ The statement "**count--**" may be implemented as:

```
register2 = count
register2 = register - 1
count     = register2
```

  - ✓ Assume **count** is initially 5. One interleaving of statements is:

```
producer: register1 = count          (register1 = 5)
producer: register1 = register1 + 1 (register1 = 6)
consumer: register2 = count          (register2 = 5)
consumer: register2 = register2 – 1 (register2 = 4)
producer: count     = register1      (counter   = 6)
consumer: count     = register2      (counter   = 4)
```

  - ✓ The value of **count** may be either 4 or 6, where the correct result should be 5

Exercise

- Producer & Consumer sharing bounded buffer (multi-threads version)

```
$ gcc –o prodcons_t prodcons_t.c -lpthread (or make prodcons_t)
$ ./prodcons_t
```

- Producer & Consumer sharing bounded buffer (multi-processes version)

```
$ gcc –o producer producer.c (or make producer)
$ gcc –o consumer consumer.c (or make consumer)
$ ./consumer

$ ./producer
```

- But,... what about the result?

Synchronization Mechanisms

- Disabling interrupts
- Spinlocks
  - ✓ Very primitive, minimal semantics, used to build others
  - ✓ Busy waiting
- Semaphores
  - ✓ Basic, easy to get the hang of, hard to program with
  - ✓ Binary semaphore = mutex (≅ lock)
  - ✓ Counting semaphore
- Monitors
  - ✓ High-level, requires language support, implicit operations
  - ✓ Easy to program with: Java “synchronized”
- Mutex + Condition variables
  - ✓ Pthreads

Semaphores

- Semaphore
  - ✓ A counter used to provide access to a shared data object for multiple processes or threads
  - ✓ Two operations
    - wait or P
    - signal or V

- Synchronization procedure using semaphores
  - ✓ Test the semaphore that controls the resource
  - ✓ If the value of the semaphore is positive, the process can use the resource
    - The process decrements the semaphore value by 1, indicating that it has used on unit of the resource
  - ✓ If the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0
    - When the process wakes up, it returns to above step

Semaphore Implementations

- System V semaphore
  - ✓ Named semaphore → between processes
  - ✓ Shared key (number) between processes
  - ✓ Serviced by kernel

- POSIX semaphore
  - ✓ Unnamed semaphore → between threads or related processes
  - ✓ Shared variable in **`sem_t`** type between threads or related processes
  - ✓ Serviced by libraries or kernel
  - ✓ Most implementation doesn't support synchronization between processes yet, including Solaris and Linux

POSIX Semaphores

- POSIX semaphore libraries
  - ✓ `#include <semaphore.h>`
  - ✓ `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - ✓ `int sem_wait(sem_t *sem);`
  - ✓ `int sem_trywait(sem_t *sem);`
  - ✓ `int sem_post(sem_t *sem);`
  - ✓ `int sem_getvalue(sem_t *sem, int *sval);`
  - ✓ `int sem_destroy(sem_t *sem);`
  - ✓ return: 0 if OK, non-zero value on error

- Note: link option (dependent on semaphore packages)
  - ✓ Solaris
    - **`-lposix4`**
  - ✓ Linux
    - **`-lpthread`**

System Calls for System V Semaphores

- Obtain a semaphore set ID
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <sys/ipc.h>`
  - ✓ `#include <sys/sem.h>`
  - ✓ `int semget(key_t key, int nsems, int flag);`
  - ✓ return: semaphore ID if OK, –1 on error

- Semaphore control operations
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <sys/ipc.h>`
  - ✓ `#include <sys/sem.h>`
  - ✓ `int semctl(int semid, int semnum, int cmd, union semun arg);`
  - ✓ return: non-negative value depending on **`cmd`** if OK, –1 on error

System Calls for System V Semaphores (Cont'd)

- Semaphore control operations (Cont'd)
  - ✓ The third argument, **`cmd`**
    - **`IPC_STAT`** : fetch the **`semid_ds`** structure for this semaphore set
    - **`IPC_SET`** : set the part of **`semid_ds`** structure
    - **`IPC_RMID`** : remove the semaphore set from the system
    - **`GETVAL`** : return the semaphore value
    - **`SETVAL`** : set the semaphore value
    - **`GETPID`** : get pid of the process which do the last access to the semaphore
    - **`GETNCNT`** : return the number of processes which wait for the semaphore to increase
    - **`GETZCNT`** : return the number of processes which wait for the semaphore to be zero
    - **`GETALL`** : fetch all the semaphore values in the set
    - **`SETALL`** : set all the semaphore values in the set
  - ✓ The fourth argument, **`arg`**

```
union semun  {
  int              val;    /* for SETVAL */
  struct semid_ds *buf;    /* for IPC_STAT and IPC_SET */
  ushort          *array; /* for GETALL and SETALL */
};
```

System Calls for System V Semaphores (Cont'd)

- Semaphore operations
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <sys/ipc.h>`
  - ✓ `#include <sys/sem.h>`
  - ✓ `int semop(int semid, struct sembuf semop[], size_t nops);`
  - ✓ return: 0 if OK, –1 on error
  - ✓ The second argument, **semop**

```
struct sembuf  {
  ushort sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short  sem_op;   /* operation (negative, 0, or positive) */
  short  sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

System Calls for System V Semaphores (Cont'd)

- Semaphore operations (Cont'd)
  - ✓ if **sem_op** > 0,
    - the value of **sem_op** is added to the semaphore's value
  - ✓ if **sem_op** < 0 and the semaphore's value >= the absolute value of **sem_op**,
    - the value of **sem_op** is added to the semaphore's value
  - ✓ if **sem_op** < 0 and the semaphore's value < the absolute value of **sem_op**,
    - if **IPC_NOWAIT** is not specified, the calling process is suspended until the semaphore's value >= the absolute value of sem_op
    - if **IPC_NOWAIT** is specified, return is made with an error of **EAGAIN**
  - ✓ if **sem_op** == 0 and the semaphore's value is not zero,
    - if **IPC_NOWAIT** is not specified, the calling process is suspended until the semaphore's value becomes zero
    - if **IPC_NOWAIT** is specified, return is made with an error of **EAGAIN**

- Semaphore adjustment on **exit**
  - ✓ What happens if a process terminates while it has resources allocated through a semaphore?
  - ✓ Use **SEM_UNDO** flags

System Calls for System V Semaphores (Cont'd)

- POSIX semaphore-like library using System V semaphore
  - ✓ `semlib.h & semlib.c`

  - ✓ `int semInit(key_t key);`
  - ✓ `int semInitValue(int semid, int value);`
  - ✓ `int semWait(int semid);`
  - ✓ `int semTryWait(int semid);`
  - ✓ `int semPost(int semid);`
  - ✓ `int semGetValue(int semid);`
  - ✓ `int semDestroy(int semid);`

Exercise

- Implementation of semaphores similar to POSIX semaphores using System V semaphores & shared memory

  `$ gcc –c semlib.c (or make semlib.o)`

- Note:
  - ✓ If a process creates a system V semaphore, its data structure remains in the kernel even though the process has terminated
  - ✓ You have to remove it through **`semctl()`** with **`IPC_RMID`** parameter

    (Or, reboot the system !!!)

Exercise (Cont'd)

- Bounded buffer implementation with semaphores

Producer

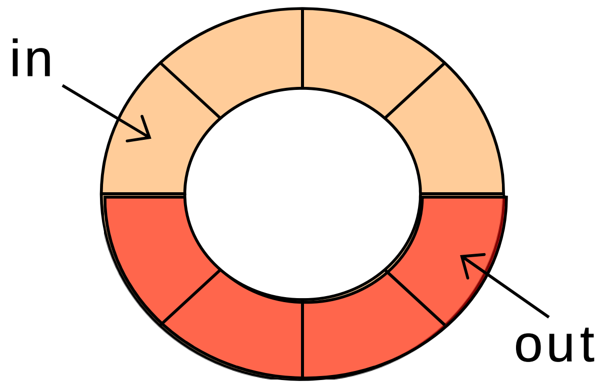
```
void produce(data)
{

   wait (empty);
   wait (mutex);
   buffer[in] = data;
   in = (in+1) % N;
   signal (mutex);
   signal (full);

}
```

```
Semaphore
    mutex = 1;
    empty = N;
    full = 0;
```

```
struct item buffer[N];
int in, out;
```



Consumer

```
void consume(data)
{

   wait (full);
   wait (mutex);
   data = buffer[out];
   out = (out+1) % N;
   signal (mutex);
   signal (empty);

}
```

Exercise

- Producer & Consumer example using pthreads & POSIX semaphores

```
$ gcc -o prodcons prodcons.c -lpthread
  (or make prodcons)
$ ./prodcons
```

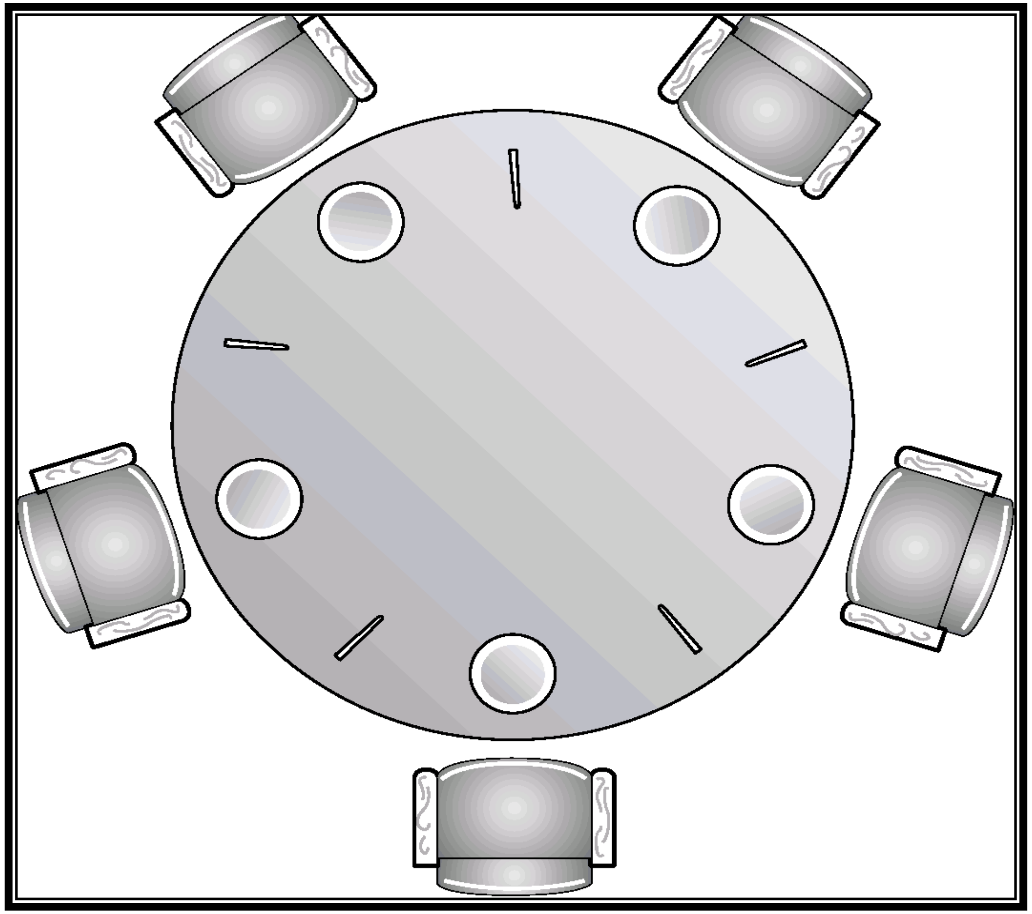
- Producer & Consumer example using **`semlib`** library

```
$ gcc -o producer_s producer_s.c semlib.c (or make producer_s)
$ gcc -o consumer_s consumer_s.c semlib.c (or make consumer_s)
$ ./consumer_s

$ ./producer_s
```

Exercise (Cont'd)

- Dining philosopher



Exercise (Cont'd)

- Dining philosopher: A simple solution

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
   while (1)  {
      think ();
      wait (chopstick[i]);
      wait (chopstick[(i+1) % N];
      eat ();
      signal (chopstick[i]);
      signal (chopstick[(i+1) % N];
   }
}
```

⇒ Problem: causes deadlock

Exercise (Cont'd)

- Dining philosopher: Deadlock-free version (starvation?)

```
#define N        5
#define L(i)     ((i+N-1)%N)
#define R(i)     ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
} }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```

Exercise

- Dining Philosopher example using pthreads & POSIX semaphores

  `$ gcc –o dining dining.c –lpthread (or make dining)`

  `$ ./dining`

  ✓ Deadlock? Yes? No? Why?

- Dining Philosopher example using pthreads & POSIX semaphores

  ✓ Deadlock-free version

  `$ gcc –o dining2 dining2.c –lpthread (or make dining2)`

  `$ ./dining2`

  ✓ Starvation? Yes? No? Why?

Mutexes

- Mutexes
  - ✓ Mutual exclusive locks for threads
  - ✓ Serviced by Pthread libraries
  - ✓ Similar to binary semaphore

- Pthread libraries for mutexes

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

  - ✓ return: 0 if OK, non-zero value on error

Condition Variables

- Condition variables
  - ✓ A synchronization device that allows threads to suspend and resume execution until some predicate on shared data is satisfied
  - ✓ Serviced by Pthread libraries
  - ✓ Associated with a mutex, to avoid the race condition

- Pthread libraries for condition variables

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

  - ✓ return: 0 if OK, non-zero value on error

Exercise

- Bounded buffer implementation with mutexes and condition variables

```
void producer(data)
{
 while (count==N) ;
 buffer[in] = data;
 in = (in+1) % N;
 count++;
}
```

```
void consumer(data)
{
 while (count==0) ;
 data = buffer[out];
 out = (out+1) % N;
 count--;
}
```

```
pthread_mutex_t mutex;
pthread_cond_t not_full, not_empty;
buffer resources[N];
void producer (resource x)  {
   pthread_mutex_lock (&mutex);
   while (array “resources” is full)
      pthread_cond_wait (&not_full, &mutex);
   add “x” to array “resources”;
   pthread_cond_signal (&not_empty);
   pthread_mutex_unlock (&mutex);
}
void consumer (resource *x)  {
   pthread_mutex_lock (&mutex);
   while (array “resources” is empty)
      pthread_cond_wait (&not_empty, &mutex);
   *x = get resource from array “resources”
   pthread_cond_signal (&not_full);
   pthread_mutex_unlock (&mutex);
}
```

Exercise (Cont'd)

- Producer & Consumer example using mutexes and condition variables

```
$ gcc -o prodcons_m prodcons_m.c -lpthread
  (or make prodcons_m)
$ ./prodcons_m
```

- Implementation of POSIX semaphores using mutexes and condition variables

```
$ gcc -c semlib2.c (or make semlib2.o)
```

- Producer & Consumer example using **semlib2** library

```
$ gcc -o prodcons_s prodcons_s.c semlib2.c -lpthread
  (or make prodcons_s)
$ ./prodcons_s
```

Summary

- Synchronization
  - ✓ Mechanisms for processes and/or threads to control their execution sequences
    - for shared resources in critical section
    - for just execution sequences (e.g. receive before send)

- System calls in Linux for high-level synchronization mechanisms
  - ✓ System V semaphore: **`semget, semctl, semop`**
  - ✓ POSIX semaphore: **`sem_init, sem_wait, sem_trywait, sem_post, sem_getvalue, sem_destroy`**
  - ✓ Mutex: **`pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock`**
  - ✓ Condition variables: **`pthread_cond_init, pthread_cond_destroy, pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast`**