

# Ateneo de Manila University

## IPC: Semaphores



Department of Information Systems and  
Computer Science

S.Y. 2001-2002

<http://sysads.ateneo.net/wyu/>

[wyy@admu.edu.ph](mailto:wyy@admu.edu.ph)

## Interprocess Communications

- ★ mechanism in which processes communicate with one another
- ★ in order to facilitate communication among different processes
- ★ special mechanisms for process communications:
  - signals
  - semaphores
  - shared memory
  - pipes
  - message passing

## Semaphores

- ★ are a programming construct designed by E. W. Dijkstra in the late 1960s
- ★ are often used to monitor and control the availability of system resources such as shared memory segments
- ★ also used for process synchronization
- ★ also use to avoid deadlocks and race conditions
- ★ are pre-specified by the system and defined in `sys/ipc.h` and `sys/sem.h`

## Semaphores

- ★ initialized using `semget ( )`
- ★ can change its ownership or permissions using `semctl ( )`
- ★ operations are performed via the `semop ( )`

## Initializing Semaphores

- ★ `int semget ( key_t key, int nsems, int semflg );`
- ★ if the call succeeds, it returns the semaphore ID (`semid`) associated to the value of `key`
- ★ `nsems` refer to the number of semaphores
- ★ returns -1 if an error occurs
- ★ the lower 9 bits of the argument `semflg` define the access permissions
- ★ `IPC_CREATE` is ORed to the `semflg` if a new semaphore is to be created

## Controlling Semaphores

```
* int semctl (int semid, int semnum, int cmd,  
union semun arg);
```

```
union semun {  
    int val;  
    /* value for SETVAL */  
    struct semid_ds *buf;  
    /* buffer for IPC_STAT, IPC_SET */  
    unsigned short int *array;  
    /* array for GETALL, SETALL */  
    struct seminfo *__buf;  
    /* buffer for IPC_INFO */  
};
```

```
struct semid_ds {  
    pid_t uid;  
    gid_t gid;  
    int mode;  
};
```

- ★ performs the control operation specified by `cmd` on the semaphore set defined by `semid`
- ★ semaphore commands:
  - `IPC_STAT` copy info from the semaphore set data structure into the structure pointed to by `arg.buf`
  - `IPC_SET` set property of the semaphore set with contents of `arg.buf` of type `struct semid_ds`
  - `IPC_RMID` remove the semaphore set and wake all waiting processes
  - `GETALL` return `semval` for all semaphores of the set into `arg.array`

- GETNCNT return `semncnt` for the `semnum`-th semaphore in the set
  - GETZCNT return `semzcnt` for the `semnum`-th semaphore in the set
  - GETPID return `sempid` for the `semnum`-th semaphore in the set
  - GETVAL return `semval` for the `semnum`-th semaphore in the set
  - SETALL set `semval` for all semaphores to the values in the `arg.array`
  - SETVAL set `semval` for the `semnum`-th semaphore in the set
- ★ returns -1 if there is an error
- ★ other return values:
- GETNCNT the value of `semncnt`
  - GETPID the value of `sempid`
  - GETVAL the value of `semval`
  - GETZCNT the value of `semzcnt`

## Semaphore Operations

```
★ int semop ( int semid, struct sembuf *sops,  
size_t nsops);  
  
struct sembuf {  
    short sem_num;  
        /* semaphore number: 0 = first */  
    short sem_op;  
        /* semaphore operation */  
    short sem_flg;  
        /* operation flags */  
};
```

★ performs operations on selected members of the semaphore set indicated by `semid`

★ each of the `nsops` elements in the array pointed to by `sops` specify an

operation to be performed on the semaphore

- ★ recognized in `sem_flg` are `IPC_NOWAIT` and `SEM_UNDO`
  - `SEM_UNDO` operation will be undone when the process exits
  - `IPC_NOWAIT` system call fails immediately if `sem_op` is zero
- ★ `sem_op` is a positive integer, the operation adds this value to `semval` of the semaphore

## Resource Sharing Best Practices

1. test the semaphore that controls the resource
2. if the value of the semaphore is positive then resource can be used and the semaphore value is decremented by one
3. if the value of the semaphore is zero the process blocks until the semaphore's value becomes positive and returns to step one
4. when a process is done it must increment the semaphore by one

## SysV Semaphores Caveats

- ★ complicated due to the availability of semaphore sets instead of a single semaphore with multiple values
- ★ semaphore creations (`semget`) is independent of semaphore initialization (`semctl`) (non-atomic)
- ★ semaphore are non-cleared upon program exit thus a possibility of leaving unused semaphore behind
- ★ Solution: encapsulate semaphore operations using Dijkstra's primitives `P()`/`Wait()` and `V()`/`Signal()`

## Proposed Lightweight Semaphore Primitives

- ★ `int sem_create (key_t key, int initval)`
  - returns a semaphore id based on the value of `key` which is initialized to `initval`
  - `initval` is typically set to one for a uninitialized semaphore
- ★ `int sem_open (key_t key)`
  - returns a semaphore id based on the value of `key_t` for an existing semaphore
- ★ `int sem_get (int semid)`
  - returns the value of the semaphore determined by `semid`
- ★ `void sem_close (int semid)`
  - decrements the semaphore if the value is not zero if it is zero it will deinitialize the data structures for the semaphore defined by `semid`

★ `void sem_rm (int semid)`

- deinitializes data structures for the semaphore defined by `semid`

★ `void sem_wait (int semid)`

- decrements a semaphore by 1
- Dijkstra's P operation. Tanenbaum's DOWN operation.

★ `void sem_signal (int semid)`

- increments a semaphore by 1
- Dijkstra's V operation. Tanenbaum's UP operation.

## POSIX Semaphore Implementation

- ★ POSIX standardized semaphore implementation
- ★ uses `semaphore.h` header file
- ★ `int sem_init (sem_t semid, int initval)`
  - initializes the semaphore `semid`
  - `initval` is typically set to one for a uninitialized semaphore used as a mutex
- ★ `void sem_destroy (sem_t semid)`
  - deinitializes data structures for the semaphore defined by `semid`
- ★ `void sem_wait (sem_t semid)`
  - decrements a semaphore by 1
  - Dijkstra's P operation. Tanenbaum's DOWN operation.
- ★ `void sem_trywait (sem_t semid)`

- non-blocking version of `sem_wait`
- this function immediately returns 0 if it can be decremented
- this function returns `EAGAIN` error if not

★ `void sem_post (sem_t semid)`

- increments a semaphore by 1
- Dijkstra's V operation. Tanenbaum's UP operation.

★ `void sem_getvalue (sem_t semid, int *sval)`

- returns the value of the incremented semaphores to `sval`

## Problems with Multiprocess Programming

- ★ Critical Sections Problems (Mutual Exclusion)
- ★ Barriers Synchronization Problems
- ★ Producer/Consumer Problems
- ★ Bounded Buffer Problems

## Critical Section Problems

- ★ problems where a shared system resource can only be accessed by a limited number of processes at a given time
- ★ also known as the mutual exclusion problem
- ★ typical shared system resources:
  - file descriptors (network connections, terminal, files, etc...)
  - shared memory segments

## Critical Section Problem Solution

```
sem mutex = 1;
process Worker1
while ( condition ) {
    P ( mutex )
    critical section
    V ( mutex )
    non-critical section
}

process Worker2
while ( condition ) {
    P ( mutex )
    critical section
    V ( mutex )
    non-critical section
}
```

## Barrier Synchronization Problems

- ★ barrier - is a stage in process execute that cannot be executed until all processes reach that stage
- ★ synchronize stages of parallel iterative programs
- ★ typically used for data parallel algorithms

## Barrier Synchronization Problem Solution

```
sem arrive1 = 0;
```

```
sem arrive2 = 0;
```

```
process Worker1 {
```

```
    ...
```

```
    V ( arrive1 );
```

```
    P ( arrive2 );
```

```
    ...
```

```
}
```

```
process Worker2 {
```

```
    ...
```

```
    V ( arrive2 );
```

```
    P ( arrive1 );
```

```
    ...
```

```
}
```

## Producer/Consumer Problem

- ★ typically has two types of processes:
  - Producer - places a single unit of data to the buffer
  - Consumer - takes a single unit of data from a buffer
- ★ problem involves the interaction of both the producer processes and consumer processes
- ★ the consumer processes must not be allowed to take data from an empty buffer
- ★ the producer processes must not be able to place data on a full buffer

## Producer/Consumer Problem Solution

```
type buffer;  
sem empty = 1, full = 0;
```

```
process Producer  
while (true) {  
    P ( empty );  
    buffer = data;  
    V ( full );  
}
```

```
process Consumers  
while (true) {  
    P ( full );  
    result = buffer;  
    V ( empty );  
}
```

## Bounded Buffer Problem

- ★ typically considered a special case of the Producer/Consumer Problem
- ★ the buffer specified in the problem is considered to be of finite capacity which is more than one
- ★ more realistic model of the Producer/Consumer Problem
- ★ the Mutex Bounded Buffer Problem is a special case of this problem when there are multiple producers and consumers

## Bounded Buffer Problem Solution

```
type buffer[n];
int front = 0, rear = 0;
sem empty = n, full = 0;

process Producer
while (true) {
    P ( empty );
    buffer[rear] = data;
    rear = (rear + 1) % n;
    V ( full );
}
```

## Bounded Buffer Problem Solution

```
process Consumers
while (true) {
    P ( full );
    result = buf[front];
    front = (front + 1) % n;
    V ( empty );
}
```

## Mutex Bounded Buffer Problem Solution

```
type buffer[n];
int front = 0, rear = 0;
sem empty = n, full = 0;
sem mutexP = 1, mutexC = 1;

process Producer
while (true) {
    P ( empty ); P ( mutexP );
    buffer[rear] = data;
    rear = (rear + 1) % n;
    V ( mutexP ); V ( full );
}
```

## Mutex Bounded Buffer Problem Solution

```
process Consumers
while (true) {
    P ( full ); P ( mutexC );
    result = buf[front];
    front = (front + 1) % n;
    V ( mutexC ); V ( empty );
}
```

## Homework: Dining Philosophers Problem

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the center of the table is a large platter of spaghetti. Since, the spaghetti is long and tangled—and the philosophers are not mechanically adept—a philosopher must use two forks to eat a helping.

Unfortunately, the philosophers can only afford five forks. One fork is placed between each pair of philosophers, and they agree that each will use only the forks to his/her immediate left and right. The problem is to write a program to simulate the behavior of the philosophers. The program must avoid the unfortunate situation in which all philosophers are hungry but none is able to acquire both forks.

## Homework: Dining Philosophers

```
process Philosopher
while ( true ) {
    think;
    acquire forks;
    eat;
    release forks;
}
```

## Bonus: Readers and Writers Problem

Two kinds of processes—reader and writers—share a database. Readers execute transactions that examine database records; writers execute transactions that both examine and update the database. The database is assumed initially to be in a consistent state ( i.e. one in which relations between data are meaningful). Each transaction, if executed in isolation, transforms the database from one consistent state to another. To preclude interference between transactions, a writer process must have exclusive access to the database. Assuming no writer is accessing the database, any number of readers may concurrently execute transactions.

Note:

- only for the first five to submit a valid solution
- will be credited as a bonus quiz



Copyright © 2000-2001 by William Emmanuel S. Yu. This material may be distributed only subject to the terms and conditions set forth in the Open Content License, v1.0 or later (the latest version is presently available at <http://opencontent.org/opl.shtml>).