System Programming

Module 10

EGCI 252
System Programming

Computer Engineering Department
Faculty of Engineering
Mahidol University

Semaphores

- Shared memory is not access controlled by the kernel
- This means critical sections must be protected from potential conflicts with multiple writers
- A critical section is a section of code that would prove problematic if two or more separate processes wrote to it simultaneously
- Semaphores were invented to provide such locking protection on shared memory segments

System V Semaphores

- You can create an array of semaphores that can be controlled as a group
- Semaphores may be binary (0/1), or counting
  1 == unlocked (available resource)
  0 == locked
- Thus:
  - To unlock a semaphore, you INCREMENT it
  - To lock a semaphore, you DECREMENT it
- Spinlocks are busy waiting semaphores that constantly poll to see if they may proceed

How Semaphores Work

- A critical section is defined
- A semaphore is created to protect it
- The first process into the critical section locks the critical section
- All subsequent processes *wait* on the semaphore, and they are added to the semaphore's "waiting list"
- When the first process is out of the critical section, it *signals* the semaphore that it is done
- The semaphore then *wakes up* one of its waiting processes to proceed into the critical section
- All waiting and signaling are done *atomically*

How Semaphores “Don’t” Work: Deadlocks and Starvation

- When two processes (a, b) are both waiting on a semaphore, and “a” cannot proceed until “b” signals, and “b” cannot continue until “a” signals. They are both asleep, waiting. Neither can signal the other, wake the other up. This is called a *deadlock*.
  - P1 locks “a” which succeeds, then waits on “b”
  - P2 locks “b” which succeeds, then waits on “a”
- Indefinite blocking, or *starvation*, occurs when one process is constantly in a wait state, and is never signaled. This often occurs in LIFO situations.

General Semaphore Example

Pseudo-code

```
semaphore sv = 1;
loop forever
{
    P(sv);
    critical code section;
    V(sv);
    non-critical code section;
}
```

Semaphore Facilities

#include <sys/sem.h>

int semget(key_t key, int num_sems, int sem_flags);

The semget function creates a new semaphore or obtains the semaphore key of an existing semaphore.

key: an integral value used to allow unrelated processes to access the same semaphore

num_sems: the number of semaphore required

sem_flags: a rights mask (0666) OR'd with one of the following:

- IPC_CREAT create a new semaphore (ignore if it already exists)
- IPC_EXCL creates a new semaphore (it will error if it already exists)

Semaphore Facilities (Cont.)

int semctl(int sem_id, int sem_num, int command, …);
The semctl function allows direct control of semaphore information
sem_id: a semaphore identifier obtained from semget
sem_num: the semaphore number
command: the following actions
SETVAL: used for initializing a semaphore to a known value
IPC_RMID: used for deleting a semaphore identifier when it is no longer required forth parameter: if present, a union semun which must have at least the following members

```
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}
```

Semaphore Facilities (Cont.)

int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);

The semop function is used for changing the value of the semaphore

sem_id: a semaphore identifier obtained from semget

sem_ops: a pointer to an array of structures, each of which will have at least the following members:

```
struct sembuf
{
    short sem_num; // the semaphore number, usually 0 unless working
                   // with an array of semaphore
    short sem_op; // the value by which the semaphore should be changed {-1, +1}
    short sem_flg; // usally set to SEM_UNDO to automatically release the
                   // semaphore after the process terminates
}
```

num_sem_ops: the number of array items

Semaphore Example

/* After the #includes, the function prototypes and the global variable, we come to the main function. There the semaphore is created with a call to semget, which returns the semaphore ID. If the program is the first to be called (i.e. it's called with a parameter and argc > 1), a call is made to set_semvalue to initialize the semaphore and op_char is set to X. [./sem & ./sem 1]*/

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);

static int sem_id;
```

```
int main(int argc, char *argv[])
{
    int i;
    int pause_time;
    char op_char = 'O';

    srand((unsigned int) getpid());

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);

    if (argc > 1) {
        if (!set_semvalue())
        {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        op_char = 'X';
        sleep(2);
    }
```

Semaphore Example (Cont.)

```
/* Then we have a loop which enters and leaves the critical section ten times.
 There, we first make a call to semaphore_p which sets the semaphore to wait, as
 this program is about to enter the critical section. */

for(i = 0; i < 10; i++)
{
        if (!semaphore_p()) exit(EXIT_FAILURE);
        printf("%c", op_char);
        fflush(stdout);
        pause_time = rand() % 3;
        sleep(pause_time);
        printf("%c", op_char);
        fflush(stdout);

 /* After the critical section, we call semaphore_v, setting
the the semaphore available, before going through the for loop
again after a random wait. After the loop, the call
to del_semvalue is made to clean up the code. */

        if (!semaphore_v()) exit(EXIT_FAILURE);

        pause_time = rand() % 2;
        sleep(pause_time);
    }

    printf("\n%d - finished\n", getpid());
    if (argc > 1)
    {
        sleep(10);
        del_semvalue();
    }
    exit(EXIT_SUCCESS);
}
```

Semaphore Example (Cont.)

/* The function set_semvalue initializes the semaphore using the SETVAL command in a semctl call. We need to do this before we can use the semaphore. */

```
static int set_semvalue(void)
{
    union semun sem_union;

    sem_union.val = 1;
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1)
        return(0);
    return(1);
}
```

/* The del_semvalue function has almost the same form, except the call to semctl uses the command IPC_RMID to remove the semaphore's ID. */

```
static void del_semvalue(void)
{
    union semun sem_union;

    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}
```

Semaphore Example (Cont.)

```
/* semaphore_p changes the semaphore
by -1 (waiting). */

static int semaphore_p(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}
```

```
/* semaphore_v is similar except for setting
the sem_op part of the sembuf structure to 1,
 so that the semaphore becomes available. */

static int semaphore_v(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}
```

Assignment

- Write a program that demonstrates a deadlock situation using two semaphores (s1 and s2) for locking two critical functions (f1 and f2).
- A program is required to create a child process.
- The parent process will call the f1 function and use the first semaphore s1 to lock the critical function then try to call the f2 function.
- The child process will call the f2 function and use the second semaphore s2 to lock the critical function then also try to call the f1 function.
- If each processes succeed, it should print “Done!” on the screen.

End of Module