Sistemas Operacionais
Laboratorio - Semáforos

Profa. Roberta L. Gomes

1 *Semaphores* no Linux

Um semáforo é uma espécie de contador associado a uma fila (geralmente implementada como FIFO), que pode ser utilizado para sincronizar múltiplos processos ou múltiplas *threads* (veremos isso mais adiante no curso! ... neste lab, por simplicidade, falaremos apenas de processos). O Linux garante que o *checking* (verificação) ou o *modifying* (modificação) do valor de um semáforo possa ser feito de forma segura e "atômica", se criar condição de corrida.

Cada semáforo contém um contador, implementado por meio de um inteiro **não-negativo**.

Um semáforo suporte duas operações básicas:

- **down() ou P() ou wait():** A operação *down()* decrementa de 1 o valor do contador do semáforo e retorna. Mas se o valor **já** é zero, a operação **não decrementa o contador**. Ao invés, ela se torna uma operação bloqueante. Isto é, o processo que chamou *down()* é bloqueado na fila do semáforo.
- **up() ou V() ou signal():** A operação *up()* incrementa de 1 o valor do contador do semáforo e retorna. Mas se houver algum processo bloqueado na fila do semáforo (observe que neste caso o contador do semáforo será 0), a operação **não** incrementa o contador. Ao invés, ela desbloqueia o primeiro processo que esteja bloqueado na fila do semáforo, retornando em seguida.

2 Thread Semaphores (EXTRA... YOU DON'T NEED IT NOW!)

A semaphore is represented by a sem$_t$*variable.Include*`<semaphore.h>`.

- `sem_init`: Initializes semaphore before use. The first parameter takes a pointer to the `sem_t` variable. The second parameter should be zero, and the third parameter is the semaphore's initial value.
- `sem_destroy`: Deallocates semaphore when finished.
- `sem_wait`: It is the *down()* operation.
- `sem_trywait`: Non-blocking version of sem_wait.If the semaphore is non-zero, decrements as in sem_wait. Otherwise, returns immediately with error EAGAIN.
- `sem_getvalue`: Stores in its second parameter the value of the semaphore passed in the first parameter. Do not use this to decide if the semaphore is available – could cause race condition.

3 Process Semaphores

Linux provides a distinct alternate implementation of semaphores that can be used for synchronizing Processes called process semaphores.

3.1 Allocation and Deallocation

`semget|` allocates a semaphore. Invoke `semget()` with a key specifying a semaphore set, the number of semaphores in the set, and permission flags. The return value is a semaphore set identifier. You can obtain the identifier of an existing semaphore set by specifying the right key value; in this case, the number of semaphores can be zero.

Semaphores continue to exist even after all processes using them have terminated. The last process to use a semaphore set must explicitly remove it to ensure that the operating system does not run out of semaphores .To do so, invoke `semctl` with the semaphore identifier, the number of semaphores in the set, `IPC_RMID` as the third argument, and any union `semun` value as the fourth argument (which is ignored).The effective user ID of the calling process must match that of the semaphore's allocator (or the caller must be root). Removing a semaphore set causes Linux to deallocate immediately.

Allocating and Deallocating a Binary Semaphore

```c
        union semun {
                int val;
                struct semid_ds *buf;
                unsigned short int *array;
        struct seminfo *__buf;
        };
      /* Initialize a binary semaphore with a value of 1. */
      int binary_semaphore_initialize (int semid)
      {
        union semun argument;
        unsigned short values[1];
        values[0] = 1;
        argument.array = values;
        return semctl (semid, 0, SETALL, argument);       }
```

Example:

```c
/* use this to store return values of system calls.   */
int rc;

/* initialize the first semaphore in our set to '3'.  */
rc = semctl(sem_set_id_2, 0, SETVAL, 3);
if (rc == -1) {
    perror("main: semctl");
    exit(1);
}

/* initialize the second semaphore in our set to '6'. */
rc = semctl(sem_set_id_2, 1, SETVAL, 6);
if (rc == -1) {
    perror("main: semctl");
    exit(1);
}

/* initialize the third semaphore in our set to '0'.  */
rc = semctl(sem_set_id_2, 2, SETVAL, 0);
if (rc == -1) {
    perror("main: semctl");
    exit(1);
}
```

3.2 Wait and Signal Operations

Each semaphore has a non-negative value and supports wait and signal operations. The semop system call implements both operations. Its first parameter specifies a semaphore set identifier. Its second parameter is an array of struct sembuf elements, which specify the operations you want to perform. The third parameter is the length of this array. The fields of struct sembuf are listed here:

- sem_num is the semaphore number in the semaphore set on which the operation is performed.
- sem_op is an integer that specifies the semaphore operation. If sem_op is a positive number, that number is added to the semaphore value immediately. If sem_op is a negative number, the absolute value of that number is subtracted from the semaphore value. If this would make the semaphore value negative, the call blocks until the semaphore value becomes as large as the absolute value of sem_op (because some other process increments it). If sem_op is zero, the operation blocks until the semaphore value becomes zero.
- sem_flg is a flag value. Specify IPC_NOWAIT to prevent the operation from blocking; if the operation would have blocked, the call to semop fails instead. If you specify SEM_UNDO, Linux automatically undoes the operation on the semaphore when the process exits.

Wait and Signal/Post Operations for a Binary Semaphore

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
/* Wait on a binary semaphore. Block until the semaphore value is
   positive, then decrement it by 1. */
int binary_semaphore_wait (int semid)
{
        struct sembuf operations[1];
        /* Use the first (and only) semaphore. */
        operations[0].sem_num = 0;
        /* Decrement by 1. */
        operations[0].sem_op = -1;
        /* Permit undoing. */
        operations[0].sem_flg = SEM_UNDO;
        return semop (semid, operations, 1);
}

/* Signal to a binary semaphore: increment its value by 1. This returns
     immediately. */
int binary_semaphore_post (int semid)
{
        struct sembuf operations[1];
        /* Use the first (and only) semaphore. */
        operations[0].sem_num = 0;
        /* Increment by 1. */
        operations[0].sem_op = 1;
        /* Permit undoing. */
        operations[0].sem_flg = SEM_UNDO;
        return semop (semid, operations, 1);
}
```

Specifying the SEM_UNDO flag permits dealing with the problem of terminating a process while it has resources allocated through a semaphore. When a process terminates, either voluntarily or involuntarily, the semaphore's values are automatically adjusted to "undo" the process's effects on the semaphore. For example, if a process that has decremented a semaphore is killed, the semaphore's value is incremented.

Example:

```c
/* this function updates the contents of the file with the given path
   name. */
void update_file(char* file_path, int number)
{
    /* structure for semaphore operations.   */
    struct sembuf sem_op;
    FILE* file;

    /* wait on the semaphore, unless its value is non-negative. */
    sem_op.sem_num = 0;
    sem_op.sem_op = -1;
    sem_op.sem_flg = 0;
    semop(sem_set_id, &sem_op, 1);

    /* we "locked" the semaphore, and are assured exclusive access to
       file.  */
    /* manipulate the file in some way. for example, write a number
       into it. */
    file = fopen(file_path, "w");
    if (file) {
        fprintf(file, "%d\n", number);
        fclose(file);
    }

    /* finally, signal the semaphore - increase its value by one. */
    sem_op.sem_num = 0;
    sem_op.sem_op = 1;
    sem_op.sem_flg = 0;
    semop(sem_set_id, &sem_op, 1);
}
```

3.3 Debugging Semaphores

Use the command `ipcs -s` to display information about existing semaphore sets. Use the `ipcrm sem` command to remove a semaphore set from the command line. For example, to remove the semaphore set with identifier 5790517, use this line:

```
% ipcrm sem 5790517
```

3.4 Assignment

You will implement a game called War by using the semaphore system calls and functions.

War will initially fork two processes, process 1 (executable of process1.c) and process2 (executable of process1.c). Each process will sleep for a random amount of time before competing on accessing the semaphore. The one that goes through first will kill the other process, increment its counter, fork a new process, and signal the semaphore.

The winning process (winner of the current battle) will then sleep for a random amount of time before competing again against the newly created process, and so on. The counter of each new process is reset before it enters the battlefield. The one process that wins 3 consecutive battles (i.e. when it's counter becomes 3) will win the War and terminate the game. Programming:

1- Write a C program (war.c) to initially launch the two processes. 2- Write one C program (process1.c) that will perform the specifications explained in the previous section.

The following statistics will be reported by the winning process:

1. WAR launching time. 2. Time of creation of the processes 3. Time of termination of the process