System Programming

Module 10.1

EGCO 252
System Programming

Computer Engineering Department
Faculty of Engineering
Mahidol University

POSIX Semaphores

- POSIX Semaphores was designed to have simpler interfaces than SYS V Semaphores
- There are two types of POSIX Semaphores
  - Named POSIX Semaphores
  - Unnamed POSIX Semaphores

Named POSIX Semaphores

- A Named POSIX Semaphore needs to have a name started with ‘/’ and followed by non-slash characters
- The new POSIX Semaphore will be created in the “/dev/shm” directory in Linux and the name of a semaphore object will have the “sem.” as a prefix followed by the name specified in the sem_open function

Creating a named POSIX Semaphore

The “sem_open()” function is used to open or create the POSIX semaphore.

```
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */
#include <semaphore.h>

sem_t sem_open(const char *name, int flag);
sem_t sem_open(const char *name, int flag,
                        mode_t mode, unsigned int value);
```

On successful completion, the sem_open() returns the semaphore object.

Creating a named POSIX Semaphore (Cont.)

- **The "name" parameter is the identifier of the semaphore that is required to start with '/' followed by non-slash characters.**
- **The "flag" parameter specifies the operation for the semaphore. It must be one of the followings:**
  O_RDONLY for reading the semaphore only.
  O_WRONLY for writing the semaphore only.
  O_RDWR for both reading and writing the semaphore.
  Additional flags can be combined with the above flags; such as,
  O_CREAT for creating the semaphore if it does not exist.
- **The "mode" parameter specifies the permissions on the new semaphore if the O_CREAT is used.**
- **The "value" parameter is the initial value for the new semaphore.**

Deleting a named POSIX Semaphore

The “sem_unlink” function is used to delete a semaphore object from the system.

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

- **The “name” parameter is the identifier of the semaphore object.**

Unnamed POSIX Semaphores

- The Unnamed POSIX Semaphore interfaces are simpler than the named POSIX Semaphore
- For multiple processes, the Unnamed POSIX Semaphore object must be in the **shared memory** area (Sys V or POSIX shared memory)
- For multiple threads, the Unnamed POSIX Semaphore object must be a global object

Creating an unnamed POSIX Semaphore

The “sem_t” type is used to create an unnamed POSIX semaphore object with the required header files.

```
#include <sys/stat.h>       /* For mode constants */
#include <fcntl.h>          /* For O_* constants */
#include <semaphore.h>
```

Initializing an unnamed POSIX Semaphore

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

The sem_init function initializes a semaphore with a specified value.

sem: the semaphore will be initialized

pshared:

a non-zero value for sharing semaphore between processes

a zero value for sharing semaphore between threads

value: a value will be used to initialize the semaphore

(the initial value of 1 for unlocked)

Deleting an unnamed POSIX Semaphore Facilities

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

```
int sem_destroy(sem_t *sem);
```

The sem_destroy function will destroy the semaphore.
sem: the semaphore will be destroyed

Locking with both named and unnamed POSIX Semaphores

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

int sem_wait(sem_t *sem);

The sem_wait function will lock the critical section.
sem: the semaphore will be used to lock.

Unlocking with both named and unnamed POSIX Semaphores

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

int sem_post(sem_t *sem);

The sem_post function will unlock the critical section.
sem: the semaphore will be used to unlock.

Named POSIX Semaphore Example

```
/* Running with the “./named & ./named 1” */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>       /* For mode constants */
#include <fcntl.h>          /* For O_* constants */
#include <semaphore.h>

int main(int argc, char *argv[])
{
    int i;
    int pause_time = 800;
    char op_char = ‘O';
    sem_t *mysem;

    mysem = sem_open(“/mysem”, O_CREAT | O_RDWR, 0660, 1);

    if (argc > 1) {
        op_char = ‘X';
        usleep(pause_time);
    }

    for(i = 0; i < 10; i++)
    {
        if (sem_wait(mysem) == -1)
            exit(EXIT_FAILURE);

        // Begining of the critical section
        printf("%c", op_char);
        fflush(stdout);
        usleep(pause_time);
        printf("%c", op_char);
        fflush(stdout);
        // End of the critical section

        if (sem_post(mysem) == -1)
                exit(EXIT_FAILURE);
            usleep(pause_time);
    }

    printf("\n%d - finished\n", getpid());
    if (argc > 1)
    {
        usleep(pause_time);
        sem_unlink(“/mysem”);
    }
    exit(EXIT_SUCCESS);
}
```

Unnamed POSIX Semaphore Example

```
/* Running with "./unnamed"*/

#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/mman.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>

#define SHM_NAME "/pshm"
#define MEM_SIZE 64
#define PAUSE_TIME 800

struct shm_st {
  sem_t sem;
  char data[MEM_SIZE];
};

int main() {
  int i, shmfd, st_size;
  void *sh_mem = NULL;
  struct shm_st *sh_area;
  char op_char;

  st_size = sizeof(struct shm_st);
  shmfd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0660);
  if (shmfd == -1) {
    perror("shm_open failed ");
    exit(EXIT_FAILURE);
  }

  i = ftruncate(shmfd, st_size);
  sh_mem = mmap(0, st_size, PROT_READ | PROT_WRITE,
                          MAP_SHARED, shmfd, 0);
  printf("Memory attached at %p\n", sh_mem);
  sh_area = (struct shm_st *)sh_mem;

  if (sem_init(&sh_area->sem, 1, 1) == -1) {
    perror("sem_init: ");
    exit(EXIT_FAILURE);
  }

  int pid = fork();
  if (pid) op_char = 'X';
  else op_char = 'O';
```

Unnamed POSIX Semaphore Example (Cont.)

```
for (i = 0; i < 10; i++) {
    if (sem_wait(&sh_area->sem) == -1)
        exit(EXIT_FAILURE);

    // Begining of the critical section
    printf("[%c", op_char);
    fflush(stdout);
    usleep(PAUSE_TIME);
    printf("%c]", op_char);
    fflush(stdout);
    // End of the critical section

    if (sem_post(&sh_area->sem) == -1)
        exit(EXIT_FAILURE);
    usleep(PAUSE_TIME);
}
```

```
if (pid) {
  usleep(PAUSE_TIME);
  sem_destroy(&sh_area->sem);
  wait(NULL);
  printf("%d - finished\n", getpid());
}
else
  printf("\n%d - finished\n", getpid());

exit(EXIT_SUCCESS);
}
```

Unnamed POSIX Semaphore Example (Cont.)

/* Running with “./thunnamed”*/

```
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>

#define PAUSE_TIME 800

sem_t mysem;
```

```
void *thread_func(void *arg)
{
  char *op_char = (char *) arg;
  for (int i = 0; i < 10; i++) {
    if (sem_wait(&mysem) == -1)
        exit(EXIT_FAILURE);

    // Begining of the critical section
    printf("[%c", *op_char);
    fflush(stdout);
    usleep(PAUSE_TIME);
    printf("%c]", *op_char);
    fflush(stdout);
    // End of the critical section

    if (sem_post(&mysem) == -1)
        exit(EXIT_FAILURE);

    // Slowing down a thread
    usleep(PAUSE_TIME);
  }
  return NULL;
}
```

```
int main() {
  pthread_t tid1, tid2;
  char op_char1 = ‘O';
  char op_char2 = 'X';

  sem_init(&mysem, 0, 1);

  pthread_create(&tid1, NULL,
      thread_func, &op_char1);
  pthread_create(&tid2, NULL,
      thread_func, &op_char2);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  sem_destroy(&mysem);
  printf("\nDone!\n");

  exit(EXIT_SUCCESS);
}
```

Assignment

- Write a program that demonstrates a deadlock situation using two POSIX 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 POSIX 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 POSIX 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