Lecture 20. Semaphores

Semaphores

Semaphores are not a form of IPC like the others: pipes, FIFOs, and message queues. A semaphore is a counter used to provide access to a shared resource for multiple processes.

To obtain a shared resource (memory, file, data structure), a process does the following:

- tests the semaphore that controls the resource
- if the value of the semaphore is positive, uses the resource, and, while using the resource, decrements the value of the semaphore by 1
- if the value of the semaphore is 0, goes to sleep until it gets >0
- once the process is done using the resource, it increments the semaphore counter by 1. Any asleep process waiting for it is awakened

The testing for >0 value and decrementing are an atomic operation, implemented in the kernel. In our example, we test `semval` explicitly.

Semaphores are implemented as sets:

1. A semaphore set is defined as a set of multiple semaphore values. The number of semaphore values in the set is specified when the semaphore is created;
2. A semaphore is first created with `semget()`, and then initialized with `semctl()`. Thus, creating a semaphore and initializing its values are not an atomic operation;
3. A program that terminates must release its semaphores.

The structure maintained in the kernel for semaphore sets is `semid_ds`:

```
struct semid_ds
{
     struct ipc_perm  sem_perm; /* semaphore permissions */
     struct sem      *sem_base;/* pointer to 1st semaphore */
     unsigned short   sem_nsems;/* # semaphores in set */
     time_t           sem_otime;/* last semop() time */
     time_t           sem_ctime;/* last change time */
     …
};
```

Each semaphore is represented by a semaphore structure:

```
struct
{
     unsigned short semval; /* semaphore value, always >=0 */
     pid_t          sempid; /* pid of last process using it */
     unsigned short semncnt;/* # processes awaiting
                               semval>current value */
     unsigned short semzcnt;/* # processes awaiting semval==0 */
     …
};
```

A semaphore is created with `semget()`:

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```
Returns semaphore ID if OK, -1 if error.

Arguments:
`key`—if specified directly, or as the result of ftok(), the default value for nsems must be 1, i.e. a set of 1 semaphore is created. Only when specified with a key of IPC_PRIVATE, and a flag of IPC_CREAT, semget() can take an nsems argument different from 1, creating a set of nsems semaphores
`nsems`—the number of semaphores in the set, always >0, but <SEMMSL (max. # of semaphores/set, system specific)
`flag`—the mode of a newly created IPC object is determined by ORing the following constants into the flag argument:

SEM_R	Read access for user.
SEM_A	Alter access for user.
(SEM_R>>3)	Read access for group.
(SEM_A>>3)	Alter access for group.
(SEM_R>>6)	Read access for others.
(SEM_A>>6)	Alter access for others.

When a semaphore set is created, the following members of structure `semid_ds` are initialized:

- `ipc_perm`—the mode member of this structure is initialized to the permission bits of flag
- `sem_otime`—is set to 0
- `sem_ctime`—is set to the current time
- `sem_nsems`—is set to `nsems` (# of semaphore values in the set)

The following function performs a series of commands on the semaphore set (really, returns information about the semaphore set):

```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);
```
Returns `err=-1` and sets `errno` if `arg` is specified, or `err=value` when `arg=NULL`

Arguments:
semid—semaphore ID
semnum—the # of the `n`-th semaphore in the set, for `n=0,nsems-1`
cmd—command from a list of 10 possibilities:

IPC_STAT Retrieve the `semid_ds` structure for the semaphore set, storing it in the structure pointed to by `arg.buf`
IPC_SET Set the `sem_perm.uid`, `sem_perm.gid`, `sem_perm.mode` from the `arg.buf` structure into the `semid_ds` structure
IPC_RMID Remove the semaphore set from the system
GETVAL Return the value of *semval* for the member *semnum*
SETVAL Set the value of *semval*, specified by `arg.val`
GETPID Return the value of *sempid* for the member *semnum*
GETNCNT Return the value of *semncnt* for the member *semnum*
GETZCNT Return the value of *semzcnt* for the member *semnum*
GETALL Retrieve all the semaphore values in the set. They are stored in the array pointed to by *arg.array*
SETALL Set all the semaphore values to the values pointed to by *arg.array*

arg—this fourth argument is optional, and its type is `semun`, a union of command-specific arguments:

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

Return values:
- for the GET* commands—the corresponding value
- all other commands—0 if OK, -1 if failure, and `errno` set

The function `semop()` performs an array of operations on a semaphore set:

```
#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[], size_t nops);
```
Returns 0 if OK, -1 if error.

Arguments:
- *semid*—semaphore ID, returned with `semget()`
- *semoparray[]*—pointer to an array of structures of semaphore operations, each of which is a `sembuf` type (see below)
- *nops*—number of operations (elements) in the array

Each structure of semaphore operations has the following members:

```
struct sembuf
{
     unsigned_short sem_num;  /* member # in semaphore set */
     short          sem_op;   /* operation <0, 0, >0 */
     short          sem_flag; /* IPC_NOWAIT, SEM_UNDO */
}
```

The operation on a semaphore in the set is specified by the `sem_op` value:
1. `sem_op>0`—"Release the resources used by the process to other processes". The value `sem_op` is added *back* to `semval`.

2. `sem_op<0`—"Acquire the resources from another process". A negative integer decrements the semaphore value by an amount equal to `|sem_op|`.
    - If `semval>=abs(sem_op)`, the shared resources are available, and `abs(sem_op)` is subtracted from `semval`, resulting in a `semval>=0`.

    - If `semval<abs(sem_op)`, the resources are not available, and:
        - if `sem_flag=IPC_NOWAIT`, `semop()` returns with `errno=EAGAIN` ("Do not wait until resource is available", calling process returns)

        - if IPC_NOWAIT is not specified, `semncnt` is incremented ("One more process waiting in line for this resource", calling process is suspended until resource becomes available). The calling process is suspended until one of the following occurs:

(i)
- `semval>=abs(sem_op)` (some other process has released the resources)
- `semncnt` is decremented (calling process is done waiting and will be using the resource), and `abs(sem_op)` is subtracted from `semval` (calling process is acquiring the resource)

(ii)
- the semaphore is removed from the system
- `semop()` returns EIDRM

(iii)
- a signal is caught by the process, handler returns
- `semncnt` is decremented (caller process is no longer waiting)
- `semop()` returns EINTR

3. `sem_op=0`—"Wait until the resource has been released by all other processes". The caller process wants to wait until `semval=0`. A `sem_op` value of zero means wait for the semaphore value to reach zero. This means that the calling process wants the resource only after all other processes have finished using it.

The following program creates a semaphore set:

```
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

void SemaphoreStat(struct semid_ds *buff);

int main(void)
{
     int semid;
     key_t key;
     int err;
     union semun arg;
     arg.buf=malloc(sizeof(struct semid_ds));

     key=ftok("semkeyfile.txt", 15);
     printf("Created key=%zd.\n", key);
     printf("errno=%d: %s.\n", errno, strerror(errno));

     semid=semget(IPC_PRIVATE, 3, 0666);
```

```
     printf("Created semaphore with semid=%d.\n", semid);
     printf("errno=%d: %s.\n", errno, strerror(errno));

     err=semctl(semid, 3, IPC_STAT, arg);
     printf("arg.val=%d\n", arg.val);
     printf("arg.buf->sem_nsems=%d\n", arg.buf->sem_nsems);
     //SemaphoreStat(arg.buf);
return 0;
}

void SemaphoreStat(struct semid_ds *buff)
{
     printf("In SemaphoreStat()...\n");
     printf("buff->sem_nsems=%d\n", buff->sem_nsems);
}
```

The output shows the semaphore ID, and the number of semaphores in the set:

```
$ ./mysemaphore
Created key=251813200.
errno=0: Undefined error: 0.
Created semaphore with semid=65536.
errno=0: Undefined error: 0.
arg.val=557857168
arg.buf->sem_nsems=3
```

The following program creates 1 semaphore set with 1 semaphore in each, with 1 operations for each set. Two processes acquire a shared resource (a text file to write into) by testing the semaphore value and decrementing it by 1 unit. After writing into the file, each process increments back the semaphore value by 1 unit, thus releasing the resource to the other process.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>      /* for malloc() */
#include <unistd.h>      /* for fork() */
#include <fcntl.h>       /* for open() */
#include <string.h>      /* for strlen() */

/*
union semun
{
     int val;
```

```
     struct semid_ds *buf;
     ushort *array;
};
*/

int main(void)
{
     int i,j;
     int pid;
     int semid;               /* semid of semaphore set */
     key_t key=IPC_PRIVATE;   /* key to pass to semget() */
     int semflg=IPC_CREAT | 0666;  /* semflg to pass to semget()
*/
     int nsems=1;             /* nsems (# semaphores per set) to
pass to semget() */
     int nops;                /* number of operations to do */
     struct sembuf *semoparray=(struct sembuf *)
malloc(2*sizeof(struct sembuf)); /* ptr to operations to perform
*/
     int fd=open("/Users/awise/Stevens/Lecture20/file.txt",
O_CREAT|O_RDWR|O_APPEND);
     ssize_t numBytes;
     char **childInput=calloc(3, 20*sizeof(char));
     childInput[0]="Adriana Wise\n";
     childInput[1]="Richard Stevens\n";
     childInput[2]="Evi Nemeth\n";

     char **parentInput=calloc(3, 20*sizeof(char));
     parentInput[0]="Bridget Wise\n";
     parentInput[1]="Rosanna Wise\n";
     parentInput[2]="James Russell Wise\n";

     /* Set up semaphore set */
     fprintf(stderr, "Calling semget() with key=%x, nsems=%d,
semflg=%o.\n", key, nsems, semflg);
     if ((semid=semget(key, nsems, semflg))==-1)
     {
          perror("Call to semget() failed.");
          exit(1);
     }
     else
          (void) fprintf(stderr, "Call to semget() succeeded:
semid=\%d\n", semid);

     int err;
     static union semun arg;
```

```
     arg.buf=malloc(sizeof(struct semid_ds));

     arg.val=1;
     err=semctl(semid, 0, SETVAL, arg);
     printf("Semaphore 0 has semval=err=%d.\n", err);
     printf("Semaphore 0 has semval=arg.val=%d.\n", arg.val);
     err=semctl(semid, 0, GETVAL, NULL);
     printf("Semaphore 0 has semval=err=%d.\n", err);
     printf("Semaphore 0 has semval=arg.val=%d.\n", arg.val);

     if ((pid=fork())<0)
     {
          perror("Call to fork() failed.");
          exit(1);
     }
     if (pid==0) /* child */
     {
          i=0;
          while (i<3) /* There is 1 semaphore in the set, to
acquire/release 3 times */
          {
          /* Test the semaphore */
          if (err>0)
          {
               /* Set arguments for 1st call to semop() on the
set */
               /* 1. First argument: semid */
               /* 2. Second argument: semoparray[nops] */
               /* Operation -1: “acquire resource” */
               semoparray[0].sem_num=0;      /* Semaphore # in
set=only use 1 "track": 1 semaphore in semaphore set */
               semoparray[0].sem_op=-1;      /* Operation
-1="DECREMENT SEMAPHORE: resource IN USE" */
               semoparray[0].sem_flg=SEM_UNDO; /* Flag="take off
semaphore asynchronous" */
               /* 3. Third argument: nops */
               nops=1;
               /* Recap the call to be made. */
               fprintf(stderr,"\nsemop: CHILD calling semop(%d,
&semoparray, %d) with: ", semid, nops);
               for (j=0; j<nops; j++)
               {
                    fprintf(stderr, "\n\tsemoparray[%d].sem_num=
%d, ", j, semoparray[j].sem_num);
                    fprintf(stderr, "sem_op=%d, ",
semoparray[j].sem_op);
```

```
                    fprintf(stderr, "sem_flg=%o\n",
semoparray[j].sem_flg);
               }

               /* First semop() call to acquire the resource */
               if ((j=semop(semid, semoparray, nops))==-1)
               {
                    perror("Call to semop() failed.");
               }
               else
               {
                    err=semctl(semid, 0, GETVAL, NULL);
                    printf("After DECREMENT CHILD: semval=%d\n",
err);
                    fprintf(stderr, "\n\nChild process taking
control of track: %d/3 times\n", i+1);
                    numBytes=write(fd, childInput[i],
strlen(childInput[i]));
                    sleep(5);
               }

               /* Set arguments for 2nd call to semop() on the
set */
               /* 1. First argument: semid */
               /* 2. Second argument: semoparray[nops] */
               /* Operation 1: “release the resource” */
               semoparray[0].sem_num=0; /* Semaphore #=0 */
               semoparray[0].sem_op=1;  /* Operation="release
resource" */
               semoparray[0].sem_flg=SEM_UNDO | IPC_NOWAIT; /*
Flag="take off semaphore, asynchronous" */
               /* 3. Third argument: nops */
               nops=1;

               /* Recap the call to be made. */
               fprintf(stderr,"\nsemop: CHILD calling semop(%d,
&semoparray, %d) with: ", semid, nops);
               for (j=0; j<nops; j++)
               {
                    fprintf(stderr, "\n\tsemoparray[%d].sem_num=
%d, ", j, semoparray[j].sem_num);
                    fprintf(stderr, "sem_op=%d, ",
semoparray[j].sem_op);
                    fprintf(stderr, "sem_flg=%o\n",
semoparray[j].sem_flg);
               }
```

```
                /* Second semop() call to release the resource */
                if ((j=semop(semid, semoparray, nops))==-1)
                {
                     perror("Call to semop() failed.");
                }
                else
                {
                     err=semctl(semid, 0, GETVAL, NULL);
                     printf("After INCREMENT CHILD: semval=%d\n",
err);
                }
           }
           else if (err==0)
           {
                fprintf(stderr, "Waiting for resource to be
released by PARENT.\n");
                sleep(5);
           }
           ++i;
      } /* end while */
     }
     else /* parent */
     {
           i=0;
           while (i<3)
           {
           /* Test the semaphore */
           if (err>0)
           {
                /* Set arguments for 1st call to semop() on the
set */
                /* 1. First argument: semid */
                /* 2. Second argument: semoparray[nops] */
                /* Operation -1: “acquire resource” */
                semoparray[0].sem_num=0;      /* Semaphore # in
set=only use 1 "track": 1 semaphore in semaphore set */
                semoparray[0].sem_op=-1;      /* Operation
-1="DECREMENT SEMAPHORE: resource IN USE" */
                semoparray[0].sem_flg=SEM_UNDO; /* Flag="take off
semaphore asynchronous" */
                /* 3. Third argument: nops */
                nops=1;
                /* Recap the call to be made. */
                fprintf(stderr,"\nsemop: PARENT calling semop(%d,
&semoparray, %d) with: ", semid, nops);
                for (j=0; j<nops; j++)
```

```
                {
                     fprintf(stderr, "\n\tsemoparray[%d].sem_num=
%d, ", j, semoparray[j].sem_num);
                     fprintf(stderr, "sem_op=%d, ",
semoparray[j].sem_op);
                     fprintf(stderr, "sem_flg=%o\n",
semoparray[j].sem_flg);
                }

                /* First semop() call to acquire the resource */
                if ((j=semop(semid, semoparray, nops))==-1)
                {
                     perror("Call to semop() failed.");
                }
                else
                {
                     err=semctl(semid, 0, GETVAL, NULL);
                     printf("After DECREMENT PARENT: semval=%d
\n", err);
                     fprintf(stderr, "\n\nParent process taking
control of track: %d/3 times\n", i+1);
                     numBytes=write(fd, parentInput[i],
strlen(parentInput[i]));
                     sleep(5);
                }

                /* Set arguments for 2nd call to semop() on the
set */
                /* 1. First argument: semid */
                /* 2. Second argument: semoparray[nops] */
                /* Operation 1: “release resource” */
                semoparray[0].sem_num=0;           /* Semaphore
#=0 */
                semoparray[0].sem_op=1;            /*
Operation="release resource" */
                semoparray[0].sem_flg=SEM_UNDO | IPC_NOWAIT;
     /* Flag="take off semaphore, asynchronous" */
                /* 3. Third argument: nops */
                nops=1;

                /* Recap the call to be made. */
                fprintf(stderr,"\nsemop: PARENT calling semop(%d,
&semoparray, %d) with: ", semid, nops);
                for (j=0; j<nops; j++)
                {
```

```
                    fprintf(stderr, "\n\tsemoparray[%d].sem_num=
%d, ", j, semoparray[j].sem_num);
                    fprintf(stderr, "sem_op=%d, ",
semoparray[j].sem_op);
                    fprintf(stderr, "sem_flg=%o\n",
semoparray[j].sem_flg);
               }
               /* Second semop() call to release the resource */
               if ((j=semop(semid, semoparray, nops))==-1)
               {
                    perror("Call to semop() failed.");
               }
               else
               {
                    err=semctl(semid, 0, GETVAL, NULL);
                    printf("After INCREMENT PARENT: semval=%d
\n", err);
               }
          }
          else if (err==0)
          {
               fprintf(stderr, "Waiting for resource to be
released by CHILD.\n");
               sleep(5);
          }
          ++i;
     } /* end while */
     }
return 0;
}
```

The key elements of this program are as follows:

After a semaphore is created, two processes are forked. Each process (parent and child) essentially performs the same operations:

- Each process accesses the same semaphore track (`sops[].sem_num=0`), because there is only 1 semaphore in the set.
- Each process waits for the track to become free and then attempts to take control of track. This is achieved by testing semval for the semaphore, and, if finding it >0, by setting `sops[].sem_op=-1` in the one-op array to decrement `semval` by 1 unit
- Once the process has control it writes one line into a text file, then sleeps for 5 seconds

- The process then gives up control of the track by adding back the 1 unit to the `semval` semaphore value `sops[1].sem_op=1`
- An additional sleep operation is then performed to ensure that the other process has time to access the semaphore before a subsequent (same process) semaphore read.

Important:
- decrementing `semval` by `|sem_op|` signifies acquiring the resource
- incrementing `semval` by `|sem_op|` means releasing the resource

This means that, initially, the `semval` semaphore value must be set to a positive value, from which each user process can decrement units each time it uses the resource. The initial setup is done with the `semctl()` function having SETVAL for its 2nd argument, and with the ***`arg`*** argument specified. The ***`arg`*** argument must have been previously set (its `val` member set to the semaphore value we want to initialize `semval` to).

Creating multiple semaphores in a set would be useful for a number of different resources to be accessed by the same process.

Homework (due Tuesday, May-17-2016): Write a program using multiple semaphores in a semaphore set, and using semaphore values greater than 1. Have your program access a file for reading/writing as your shared resource.