Embedded Operating Systems

Agenda

- Race condition
- Synchronization
- Semaphore
    - Concepts
    - Usage
    - System calls

Synchronization

- Multiple processes accessing same resource at the same time, is known as "race condition".
- When race condition occurs, resource may get corrupted (unexpected results).
- Peterson's problem, if two processes are trying to modify same variable at the same time, it can produce unexpected results.
- Code block to be executed by only one process at a time is referred as Critical section. If multiple processes execute the same code concurrently it may produce undesired results.
- To resolve race condition problem, one process can access resource at a time. This can be done using sync objects/primitives given by OS.
- OS Synchronization objects are:
    - Semaphore, Mutex, Condition variables

Semaphore

- Semaphore is a sync primitive given by OS.
- Internally semaphore is a counter. On semaphore two operations are supported:
    - wait operation: dec op: P operation:
        - semaphore count is decremented by 1.
        - if cnt < 0, then calling process is blocked.
        - typically wait operation is performed before accessing the resource.
    - signal operation: inc op: V operation:

        - semaphore count is incremented by 1.
        - if one or more processes are blocked on the semaphore, then one of the process will be resumed.
        - typically signal operation is performed after releasing the resource.
- Q. If sema count = -n, how many processes are waiting on that semaphore?
    - Answer: "n" processes waiting
- Q. If sema count = 5 and 3 P & 4 V operations are performed, then what will be final count of semaphore?
    - Ans: 5 - 3 + 4 = 6

Semaphore types

- Counting Semaphore
    - Allow "n" number of processes to access resource at a time.
    - Or allow "n" resources to be allocated to the process.
- Binary Semaphore
    - Allows only 1 process to access resource at a time or used as a flag/condition.

Typical usage of Semaphore is for

- **Counting resources**
    - Initially sem=n.

Process1
P(sem);
...
...
V(sem);

- **Mutual exclusion**

- Initially sem=1.

Process1	Process2
P(sem);	P(sem);
...	...
...	...
V(sem);	V(sem);

- **Flag/Event**
  - Initially sem=0.

Process1	Process2
...	...
P(sem);	...
...	...
...	V(sem);
...	...

Semaphore System calls

semget() syscall

- Create semaphore with number of semaphore counters and given permissions.
- semid = semget(sem_key, num_of_counter, flags);
  - arg1: unique key for semaphore

- arg2: number of semaphore counters in this semaphore object
- arg3: IPC_CREAT | 0600 --> to create semaphore with rw- --- --- permissions.
- returns semaphore id on sucess.

semctl() syscall

Initialize semaphore counter

- semctl(semid, cntr, SETVAL, su);
  - arg1: id of semaphore whose counter to be initialized.
  - arg2: semaphore counter index (zero-based)
  - arg3: command = SETVAL to set value of single semaphore counter.
  - arg4: user-defined semaphore union

```
union semun {
    int              val;    // Value for SETVAL **
    struct semid_ds *buf;    // Buffer for IPC_STAT
    unsigned short  *array;  // Array for GETALL, SETALL
};

union semun su;
su.val = init_cnt;
semctl(semid, cntr_index, SETVAL, su);
```

Initialize all semaphore counters

- semctl(semid, cntr, SETALL, su);
  - arg1: id of semaphore whose counter to be initialized.

- arg2: semaphore counter index = 0.
- arg3: command = SETALL to set values of all counters at once.
- arg4: user-defined semaphore union

```
union semun {
    int              val;    // Value for SETVAL
    struct semid_ds *buf;    // Buffer for IPC_STAT
    unsigned short  *array;  // Array for GETALL, SETALL **
};

unsigned short init_cntrs = {0, 1, 5}; // array size should be same as number of semaphore counters
union semun su;
su.array = init_cntrs;
semctl(semid, 0, SETALL, su);
```

Get semaphore information

- semctl(semid, cntr, IPC_STAT, su);
  - arg1: id of semaphore whose counter to be initialized.
  - arg2: semaphore counter index = 0.
  - arg3: command = IPC_STAT to get info about semaphore.
  - arg4: user-defined semaphore union

```
union semun {
    int              val;    // Value for SETVAL
    struct semid_ds *buf;    // Buffer for IPC_STAT **
    unsigned short  *array;  // Array for GETALL, SETALL
```

```
};

struct semid_ds sem_info;
union semun su;
su.buf = &sem_info;
semctl(semid, 0, IPC_STAT, su);
```

Destroy semaphore

- semctl(semid, 0, IPC_RMID);
  - arg1: id of semaphore to be destroyed.
  - arg2: semaphore counter index (ignored while IPC_RMID)
  - arg3: command = IPC_RMID to destroy the semaphore.

semop() SysCall

- semop(semid, ops, nops);
  - arg1: semid whose counter to be incremented/decremented.
  - arg2: array of sembuf struct -- operations to be done on semaphore counters.

```
struct sembuf { // pre-defined
    sem_num; // semaphore counter index (zero-based)
    sem_op; // V(s) = +1 or P(s) = -1
    sem_flg; // 0
};
```

  - arg3: number of operations in arg2 (i.e. number of elements in the array).
- Example: Semaphore with 3 counters: 0=empty, 1=mutex, 2=filled

```
// P(sf,sm);
struct sembuf ops[2];
ops[0].sem_num = 2;
ops[0].sem_op = -1;
ops[0].sem_flg = 0;
ops[1].sem_num = 1;
ops[1].sem_op = -1;
ops[1].sem_flg = 0;
semop(semid, ops, 2);
```

Semaphore Reading

- Bach: semget(), semop()

Assignments

1. Parent prints "Sunbeam" and then child prints "Infotech". Print each letter after a delay of 1 second.

```
char *str = "Sunbeam";
for(i=0; str[i]!='\0'; i++) {
  putchar(str[i]);
  fflush(stdout);
  sleep(1);
}
```