Embedded Operating System

Agenda

- Sockets
- Shared memory
- Semaphore
- Process Synchronization
- Deadlock

Socket

- Socket is defined as communication endpoint.
- Sockets can be used for bi-directional communication.
- Using socket one process can communicate with another process on same machine (UNIX socket) or different machine (INET sockets) in the network.
- Sockets can also be used for communication over bluetooth, CAN, etc.
- terminal> man 7 socket
- Reading: Beginning Linux Programming

UNIX socket

- Bi-directional stream based communication mechanism to communicate between two processes running on the same computer.
- UNIX socket is a special file (s).
- terminal> man 7 unix

INET (Internet) socket

- Bi-directional communication mechanism to communicate between two processes running on the same computer or different computers in a network.
- INET socket do not have special file. It is identified by IP address and port number combination.
  - IP address uniquely identify each machine in the network.
    - IPv4 - 32-bit IP address e.g. 192.168.0.101

        - IPv6 - 128-bit IP address e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334
    - PORT number is 16-bit logical number (0-65535) to identify socket uniquely on a system.
        - 0-1023 - Used for well-known services like http (80), https (443), ftp (21), ssh (22), telnet (23), etc.
        - 1024+ - Used for applications like mysql (3306), mqtt(1883), etc.
- INET sockets can be stream based or packet based depending on the protocol.
    - TCP -- Stream based protocol
        - Acknowledgement
        - Order of data transfer
        - Reliable
    - UDP -- Packet based protocol
        - No acknowledgement
        - No guarantee of data transfer order
        - Packets may be lost
        - Faster

Important networking commands

```
sudo apt install net-tools

netstat -tln
# list all tcp listening sockets

netstat -tn
# list all tcp client sockets

netstat -uln
# list all udp listening sockets

netstat -un
# list all udp client sockets

netstat -a
# list all sockets
```

```
sudo netstat -ap
# list all sockets
# -p : name of process

ifconfig
# get ip-address of system
```

Socket System Calls

- fd = socket(addr_family, sock_type, protocol);
  - arg1: addr_family = AF_UNIX or AF_INET
  - arg2: sock_type = SOCK_STREAM or SOCK_DGRAM
  - arg3: protocol = 0
- Socket information is maintained in struct sockaddr. It will differ for each address family.
  - struct sockaddr
    - sa_family -- address family (AF_UNIX or AF_INET)
  - struct sockaddr_un
    - sun_family -- address family = AF_UNIX
    - sun_path -- UNIX socket file path
  - struct sockaddr_in
    - sin_family -- address family = AF_INET
    - sin_port -- port number in big endian format
    - sin_addr -- struct in_addr -- ip address in big endian format
- ret = bind(fd, &sock_addr, sock_length)
  - arg1: socket fd (created by socket() call)
  - arg2: address of sockaddr_un or sockaddr_in
  - arg3: sizeof(sockaddr_un) or sizeof(sockaddr_in)
  - returns 0 on success.
- ret = connect(fd, &sock_addr, sock_length)
  - arg1: socket fd (created by socket() call)

- arg2: address of sockaddr_un or sockaddr_in
- arg3: sizeof(sockaddr_un) or sizeof(sockaddr_in)
- returns 0 on success.

- cli_fd = accept(fd, &sock_addr, &sock_length)
  - arg1: socket fd (created by socket() call)
  - arg2: (out-param) address of sockaddr_un or sockaddr_in
  - arg3: (in-out param) sizeof(sockaddr_un) or sizeof(sockaddr_in)
  - returns 0 on success.
- listen(fd, pending);
  - arg1: socket fd (created by socket() call)
  - arg2: max pending socket conection requests
- shutdown(fd, SHUT_RDWR);
  - arg1: socket fd (created by socket() call)
  - arg2: which connections to be closed -- SHUT_RDWR.

Socket Programming Flow

- https://youtu.be/3RwFfquNgNc

Shared memory

- OS can create a special memory region which is accessible to multiple processes (willing to communicate with each other). This region is called as "Shared memory".
- This memory space/region is created in user space. Whole communication happends in user space itself. So this is fastest IPC mechanism.
- The allocated shared memory region needs to be mapped to virtual address space of the process.
  - cmd> cat /proc/pid/maps
- This is Sys V IPC mechanism (like Message queue and Semaphore).
  - cmd> ipcs
- Shared memory programming steps
  - Create shared memory region -- shmget().
  - Associate it with current process and get pointer to it -- shmat().
  - Read/Write in memory using the pointer.

    - Release pointer to the shared memory -- shmdt().
    - Destroy shared memory region -- shmctl().

Shared Memory SysCalls

shmget() syscall

- Create the shared memory object.
- shmid = shmget(shm_key, size, flags);
    - arg1: unique key for shared memory object
    - arg2: size of shared memory region (bytes)
    - arg3: to create shared memory -- IPC_CREAT | perm
        - perm --> octal number --> 0600 (rw- --- ---)
    - returns: shm id (index to shared memory table) on success.

shmctl() syscall

- To control shared memory object/region e.g. mark shm object for deletion, get info about shared memory.

Mark for deletion

- ret = shmctl(shmid, IPC_RMID, NULL);
    - arg1: shared memory id to be deleted.
    - arg2: command = IPC_RMID to delete the shared memory object
    - arg3: for deletion third arg is not required.
- This will mark shared memory object for destruction.
- The shared memory object will be deleted when no processes are attached to it (nattach = 0).

Get info about shared memory

- ret = shmctl(shmid, IPC_STAT, &shmid_ds);
    - arg1: shared memory id whose info to be retrieved.

    - arg2: command = IPC_STAT to get the info about shared memory.
    - arg3: out param to collect info about shared memory.

shmat() syscall

- Get the address of the shared memory region i.e. attach shm region to the current process.
    - Internally increments nattach count in shared memory object.
- ptr = stmat(shmid, virt_addr, flags);
    - arg1: shared memory id of region to be attached to the process.
    - arg2: base virtual address in address space of the current process to be mapped to shared memory region.
        - NULL means use any available address.
    - arg3: flags (extra information/behaviour).
        - 0 means default behaviour.
    - returns pointer to the shared memory region on success. -1 is returned on failure.

shmdt() syscall

- Release the shared memory pointer i.e. detach shm region from the current process.
    - Internally decrements nattach count in shared memory object.
    - If nattach count become zero and shared memory region is marked for deletion, delete the shared memory.
- shmdt(ptr);
    - arg1: shared memory pointer to be detached.

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.

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()

Semaphore

- 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);

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

- Flag/Event
  - Initially sem=0.

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

Assignments

1. Create a server that accept multiple clients connected over UNIX sockets. Each client send two numbers, server do the addition and send the result back. The client connection should be closed immediately after sending response.

```
// server code
// ...
while(1) {
    cli_fd = accept(serv_fd, &cli_addr, &socklen);

    read(cli_fd, &num1, sizeof(int));
    read(cli_fd, &num2, sizeof(int));
    result = num1 + num2;
    write(cli_fd, &result, sizeof(int));
    close(cli_fd);
}
```