Tutorial on the Unix/Linux operating system: Concurrency features

1. Introduction

In this tutorial, a brief description of process management and interprocess communication in the Unix/Linux operating systems is presented. In section two, the issue of process creation and termination is discussed and in section three the interprocess communication is looked at. All these issues are discussed in the context of a multiprogrammed uniprocessor system. In section five, some of the system calls are explained in more detail.

2. Process creation and termination

The only way to create a new process under Unix/Linux operating system is to call the fork() system call. The new process created by fork() is called a child process. This function is called once but returns twice. One return is to the new created child process and the other one to the parent (the process that calls fork()). The only difference in the returns is that the return value in the child is 0 while the return value in the parent is the process ID of the child process.

Both the child and parent continue executing with the instruction that follows the fork() instruction. The child is a clone of the parent. The two concurrent processes can check the return value and determine which one is the parent and which one is the child.

There are two uses for the fork() function:

1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers where the parent waits for a service request from the client. When the request arrives, the parent calls fork() and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case the child does an exec() to load and run a new program.

When a process calls the exec() function, that process is completely replaced by the new program, and the new program starts executing at its main() function. exec() merely replaces the current process (its code, data, heap, and stack segments) with a new program from disk. For doing exec() function, different system calls exists on the Unix/Linux operating systems. In this tutorial, we only introduce execlp() system call. This system call can be used for doing all of the assignments. Later we will describe the detail of this system call.
There are two ways for a process to terminate normally:
1. by completing the execution of the main() function (the main() function is a function inside the process that gets control after the process is created and its code is replaced by the exec() function.
2. calling the exit() function.

3. Interprocess communication and synchronization

In the last section, we described how processes are created and terminated on the Unix/Linux operating systems. We will now describe System V IPC used by processes to communicate with each other.

3.1 System V IPC

There are many similarities between the three types of IPC that we call V IPC-message queues, semaphores, and shared memory. Each IPC structure in the kernel is referred to by a non-negative identifier. To send or fetch a message to or from a message queue, for example, all we need to know is the identifier of the queue.

3.1.1. Message Queues

Message queues are a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created, or an existing queue is opened by **msgget()**. New messages are added to the end of a queue by **msgsnd()**. Messages are fetched from a queue by **msgrcv()**. With the use of message queues, it is possible for a process to write some messages to a queue and have the messages read by another process at a later time. Message queues can be used for IPC when processes are unrelated (they are created by separate parents).

3.1.2. Semaphores

A semaphore is a counter used to provide access to a shared data object for multiple processes.

The first function to call is **semget()** to obtain a semaphore ID and then the function **semctl()** should be used for initializing the semaphore. Note that the creation of a semaphore with **semget()** is independent of its initialization using **semctl()**.

The function **semop()** could be used for doing different operations on a semaphore. Calling the function **semop()** may result in the blocking of the caller process depending on the kind of operation and the current value of the semaphore. Basically, the function **semop()** asks for adding or subtracting a value **(SEM_OP)** from the current value of semaphore. **If the SEM_OP is positive, it will be added to the current value of semaphore.** This corresponds to the release of resources that a semaphores controls. **If the SEM_OP is negative, the caller wants to wait until the semaphore's value becomes greater than or equal to the absolute value of SEM_OP. Then the absolute value of SEM_OP is subtracted from the semaphore current value**. This corresponds to the allocation of resources. **If SEM_OP is zero, the caller wants to wait until the semaphore's value becomes zero.**

3.1.3. Shared memory

Shared memory allows two or more processes to share a common area of memory. This is the fastest form of IPC because data does not need to be copied between processes. The only trick in using shared memory is synchronizing access to shared memory among processes.

The first function to call is **shmget()**, to obtain a shared memory identifier. Once a shared memory segment is created, it should be attached to the process address space. This is done by calling the function **shmat()**. This function returns a pointer to the beginning of the memory segment. The shared memory identifier should be passed to all processes that want to share the segment. These processes also should attach the shared segment to their address space by calling **shmat()** function. For example, the segment could be created by a parent process and then the shared memory identifier is passed to the child processes through the argument list of **execlp()** system call. The child processes can call **shmat()** function to connect the shared memory to their address space.

4. Sleeping-barber problem

A modified sleeping-barber problem (Q6.7, pg. 212, Silberschatz & Galvin) is selected as an example to become more familiar with how process management and inter-process communication are done on the Unix/Linux operating systems. The number of chairs in the waiting room is assumed to be 1.

The Customer Generator process generates customers and checks for the availability of the chair. If none exists, the process waits. Otherwise, it occupies the chair occupied and if necessary wakes up the sleeping-barber. The Serve

Customers process checks for the availability of a customer. If none available, it waits (sleeps) for an arrival. Otherwise, the chair in the waiting room is made vacant and the customer is served.

Two semaphores are created. One of the semaphores is used to stop the Customer Generator when the chair is taken and the next one is used to stop the Serve Customer when the chair is vacant.

NOTE: A fundamental problem with sys tem V IPC is that the IPC structures are system-wide. For example, if a process creates a semaphore and then terminates, the semaphore and its associated data structures in the kernel remains alive. The removal may be done by executing the ipcrm shell command. Before using ipcrm, the ID of the existing semaphores should be identified by the ipcs shell command. ipcs provides you with the IDs of all active semaphores, message queues and shared memory segments. By typing the command ipcrm followed by the particular ID number, you can remove that IPC structure from the kernel. During debugging the program, it could happen that we create a bunch of semaphores or shared memory segments without killing them at program termination. Note that there is a limit of ten on the number of semaphores, we could create. This limit for shared memory is one hundred. After running the program make sure that all semaphores and shared memory segments are killed.

5. Detailed description of some system calls

In this section, some of the system calls which are used more frequently for writing concurrent application is explained in more detail.

5.1 Semaphore related system calls

The semaphore related system calls are **semget(), semctl(), and semop()**. The function **semget()** is used to create a set of semaphores and to obtain the semaphore ID for the set. The syntax of this system call is:

```
int semget(key, nsems, semflg)
        key t key;
        int nsems, semflg;
```

The value returned by **semget()** is the semaphore identifier, or -1 if an error occurred. The **nsems** argument specifies the number of semaphores in the set and the **semflg** value defines the access right to the semaphore. For example, the instruction

```
sync_sem1 = semget(IPC_PRIVATE, 1, SEM_MODE);
```

will create a semaphore set with one semaphore. The key value should always be set to **IPC_PRIVATE**. Choosing **IPC_PRIVATE** for the key value guarantees that a brand new semaphore will be created. The semflg is set to SEM_MODE which gives R/W access mode to the semaphore. The values of IPC_PRIVATE and SEM_MODE are defined in <sys/ipc.h> and in the sleeping-barber example.

Once a semaphore set is created with **semget()**, operations are performed on one or more of the semaphores in the set using the **semop()** system call. The syntax of this system call is shown below:

```
int semop(semid, sops, nsops)

        int semid;
        struct sembuf    *sops;
        int nsops;
```

The **semid** is the semaphore set ID and is the output of the system call **semget()**. The pointer **sops** points to an array each element of which has the following data structure.

```
struct sembuf {
            ushort sem_num;
            short sem_op;
            short sem_flg;}
```

The number of operations to be performed is specified by the **nsops** argument. Each element in this array specifies an operation for one particular semaphore in the set. The particular semaphore is specified by the **sem_num** value, which is zero for the first element and **nsems-1** for the last one. **nsems** is the number of semaphores in the set. Each particular operation is specified by a **sem_op** value. There are multitude of options available to the **semop()** system call, which is controlled through the **sem_flg**. For example, the **IPC_NOWAIT** flag for the **sem_flg** tells the systems to not block the caller if the operation cannot be completed. Whenever we specify SEM_UNDO flag for the **sem_flg** and we allocate resources, the kernel remembers how many resources are allocated to that particular semaphore and when the process terminates, either voluntarily or involuntarily, the kernel would release all allocated resources. Following instruction shows an example of **semop** system call. In this example, the **sync_sem1** is the semaphore set ID. The number of operation that should be done is 1 and **semopbuf** points to the data structure that defines the operation that should be done. The **sem_op** field determines the operation that should be done. If the

```
semop(sync_sem1, &semopbuf, 1)
```

sem_op is positive, it will be added to the current value of semaphore. This corresponds to the release of resources that a semaphores controls. If the **sem_op** is negative, the caller wants to wait until the semaphore's value becomes greater than or equal to the absolute value of **sem_op**. Then the absolute value of **sem_op** is subtracted from the semaphore current value. This corresponds to the allocation of resources.
The **semctl()** system call is the catchcall for various semaphore operations. Usually, this system call is used for initializing the semaphore set. The syntax of this system call is as follows:

```
int semctl(semid, semnum, cmd, arg)

    int semid, semnum, cmd;
    union semun {
                    int val;
                    struct semid_ds *buf;
                    ushort *array;
    } arg;
```

The **cmd** argument specifies the command that should be performed on the set specified by the **semid**. Usually, we use the **semctl()** system call for initializing the semaphores; the **cmd** should set to **SETVAL** for semaphore initialization. The **semnum** argument specifies one of the semaphores in the semaphore set. The initialization value for the semaphore is taken from the val field of argument **arg**. The following instruction shows an example of using **semctl()** function call.

```
semctl(sync_sem1,0, SETVAL, sem_init);
```

The value of **semnum** is set to 0 because we have only one semaphore in the semaphore set. Note that **cmd** is set to **SETVAL**. By doing this, we asked the kernel to initialize the semaphores. The **sem_init** is a data structure of type **semun** and contains the initialization value for the semaphore. Because the command is set to **SETVAL** the

val field of the data structure semun is important for us; so you need to set only the **val** field in this data structure and need not be concerned with the content of other fields.

NOTE: Although Unix/Linux allows us to create an array of semaphores by using the **semget()** system call. You should create only one semaphore at a time. So if you need to create N semaphores call **semget()** N times, each time with **nsems** equal to 1.

5.2 Shared memory related system call

The shared memory system calls are **shmget()**, **shmctl()**, and **shmat()**. In this section, we just explain the details of the two system calls **shmget()** and **shmat()**.

shmget() is used to obtain a shared memory identifier. The syntax of this system call is as follows:

```
int shmget(key, size, shmflg)

        key_t   key;
        int size,shmflg;
```

The **size** argument specifies the size of shared memory in bytes and the **shmflg** determines access right to the shared memory. You should set the **shmflg** to **SHM_MODE**. **SHM_MODE** is defined in sleeping-barber example. This provides R/W access to the shared memory segment. This system call returns an identifier for the created shared memory. Once a shared memory segment has been created, a process attaches it to its address space by calling **shmat()** system call. The syntax of this system call is as follows:

```
void *shmat(shmid, addr, flag)

        int shmid;
        void *addr;
        int flag;
```

This system call returns a pointer to an existing shared memory segment. The address in the calling process at which the segment is attached depends on the **addr** and **flag** argument. If **addr** is 0, the segment is attached at the first available address selected by the kernel regardless of the value of the flag. This is recommended technique. The **shmid** is the shared memory identifier obtained through the **shmget()** system call.

5.3 execlp() system call

The syntax of **execlp()** system call is defined below. The first argument is a pointer to a character array containing the name of a executable file. This executable file will replace the code of the process that calls the **execlp()** system call. The succeeding arguments are also pointers to character arrays. These arguments will be passed to the new code through the argument list of **main()** function. The number and nature of the nature of arguments depend on the application. The last argument is null pointer that determines the end of the argument list.

```
int execlp( char *filename, char *arg0,... .. ........ ...,(char *) 0):
```