Systems Programming V
(Shared Memory, Semaphores, Concurrency Issues)

Iqbal Mohomed

CSC 209 – Summer 2004

Week 10

Shared Memory

- Shared Memory allows two or more processes to share a given region of memory – this is the FASTEST form of IPC because the data does not need to be copied between communicating processes
- The only trick in using shared memory is synchronizing access to a given region among multiple processes – if the server/producer process is placing data into a shared memory region, the client/consumer process shouldn't try to access it until the server is done
- Often, semaphores are used to synchronize shared memory access

shmget()

- shmget() is used to obtain a shared memory identifier

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmget(key_t key, int size, int flag);
```

- shmget() returns a shared memory ID if OK, -1 on error
- Key is typically the constant "IPC_PRIVATE", which lets the kernel choose a new key – keys are non-negative integer identifier, but unlike fds they are system-wide, and their value continually increases to a maximum value, where it then wraps around to zero
- Size is the size of shared memory segment in bytes
- Flag can be "SHM_R", "SHM_W" or "SHM_R | SHM_W"

shmat()

- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat():
- void *shmat(int shmid, void* addr, int flag);
- shmat() returns a pointer to shared memory segment if OK, -1 on error
- The recommended technique is to set addr and flag to zero, i.e.:
- char* buf = (char*)shmat(shmid,0,0);
- The UNIX commands "ipcs" and "ipcrm" are used to list and remove shared memory segments on the current machine
- The default action is for a shared memory segment to remain in the system even after the processs dies – a better technique is to use shmctl() to set up a shared memory segment to remove itself once the process dies

shmctl()

- shmctl() performs various shared memory operations:
- int shmctl (int shmid, int cmd, struct shmid_ds *buf);
- cmd can be one of IPC_STAT, IPC_SET, or IPC_RMID:
  - IPC_STAT fills the buf data structure
  - IPC_SET can change the uid, gid, and mode of the shmid
  - IPC_RMID sets up the shared memory segment to be removed from the system once the last process using the segment terminates or detaches from it – a process detaches from a shared memory segment using shmdt(void* addr), which is similar to free()
- shmctl() returns 0 if OK, -1 on error

Shared Memory Example

```
char* ShareMalloc(int size)
{
    int shmId;
    char* returnPtr;
    if ((shmId=shmget(IPC_PRIVATE, size, (SHM_R | SHM_W)) < 0)
        Abort("Failure on shmget\n");
    if (returnPtr=(char*)shmat(shmId,0,0)) == (void*) -1)
        Abort("Failure on shmat\n");
    shmctl(shmId, IPC_RMID, (struct shmid_ds *) NULL);
    return (returnPtr);
}
```

mmap()

- An alternative to shared memory is memory mapped I/O, which maps a file on disk into a buffer in memory, so that when bytes are fetched from the buffer corresponding bytes of the file are read
- One advantage is that the contents if files are non-volatile
- Usage:
- caddr_t mmap(caddr_t addr, size_t len, int prot, int flag, int filedes, off_t off);
- addr and off should be set to 0
- len is the number of bytes to allocate
- prot is the file protection, typically (PROT_READ|PROT_WRITE)
- flag should be set to MAP_SHARED emulate shared memory
- filedes is a file descriptor that should be opened previously

Memory Mapped I/O Example

```
char* ShareMalloc(int size)
{
    int fd;
    char* returnPtr;
    if (fd=open("/tmp/mmap",O_CREAT | O_RDWR,0666) < 0)
          Abort("Failure to open\n");
    if(lseek(fd,size-1,SEEK_SET) == -1)
          Abort("Failure on lseek");
    if (write(fd,"",1) != 1)
          Abort("Failure on write");
    if ((returnPtr =
(char*)mmap(0,size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0)) == (caddr_t)
-1)
          Abort("Failure on mmap");

    return (returnPtr);
}
```

Concurrency

- The two key concepts driving computer systems and applications are
  - communication: the conveying of information from one entity to another
  - concurrency: the sharing of resources in the same time frame
- Concurrency can exist in a single processor as well as in a multiprocessor system
- Managing concurrency is difficult, as execution behaviour is not always reproducible

Example

Suppose we have this function:

```
void charatatime(char* str)
{
   char* ptr;
   int c;
   setbuf(stdout,NULL);
   for(ptr=str;c=*ptr++;)
        putc(c,stdout);
}
```

What Happens?

```
int main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
          Abort("Fork Error");
    else if (pid == 0)
          charatatime("output from child\n");
    else
          charatatime("output from parent\n");

    exit(0);
}
```

A Race Condition!

- The text might be displayed separate OR it might be interspersed!
- Running the program multiple times may produce different outputs!!
- Race conditions often cause compile time non-determinism

Race Conditions

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- E.g., If any code after a fork depends on whether the parent or child runs first
  - A parent process can call wait() to wait for child's termination (may block)
  - A child process can wait for parent to terminate by polling (wasteful)
- Standard solution is to use signals

Producer/Consumer Problem

- Simple example: who | wc –l
- Both the writing process (who) and the reading process (wc) of a pipeline execute concurrently
- A pipe is usually implemented as an internal OS buffer
- It is a resource that is concurrently accessed by the reader and the writer, so it must be managed carefully

Producer/Consumer Issues

- Consumer should be blocked when buffer is empty
- Producer should be blocked when buffer is full
- Producer and Consumer should run independently as far as buffer capacity and contents permit
- Producer and Consumer should never be updating the buffer at the same instant (otherwise data integrity cannot be guaranteed)
- Producer/Consumer is a harder problem if there is more than one Consumer and/or more than one Producer

Protecting Shared Resources

- Programs that manage shared resources must protect the integrity of the shared resources.
- Operations that modify the shared resource are called critical sections.
- Critical section must be executed in a mutually exclusive manner.
- Semaphores are commonly used to protect critical sections.

Semantics for Proper Shared Resource Access

- Code that modifies shared data usually has the following parts:
  - Entry section: The code that requests permission to modify the shared data.
  - Critical Section: The code that modifies the shared variable.
  - Exit Section: The code that releases access to the shared data.
  - Remainder: The remaining code

The Critical Section Problem

- The critical section problem refers to the problem of executing critical sections in a fair, symmetric manner. Solutions to the critical section problem must satisfy each of the following:
  - Mutual Exclusion: At most one process is in its critical section at any time
  - Progress: If no process is executing its critical section, a process that wishes to enter can get in
  - Bounded Waiting: No process is postponed indefinitely
- An atomic operation is an operation that, once started, completes in a logical indivisible way. Most solutions to the critical section problem rely on the existence of certain atomic operations

Semaphores

- A semaphore is an integer variable with two atomic operations: wait and signal. Other names for wait are down, P and lock. Other names for signal are up, V, unlock and post.
- A process that executes a wait on a semaphore variable S cannot proceed until the value of S is positive. It then decrements the value of S. The signal operation increments the value of the semaphore variable

Some FLAWED Pseudocode

```
void wait(int *s)
{
    while (*s <= 0); /* END WHILE*/
    (*s)--;
}

void signal(int *s)
{
   (*s)++;
}
```

Semaphores (contd.)

- Three problems with the previous slide's wait() and signal():
  1. busy waiting is inefficient
  2. doesn't guarantee bounded waiting
  3. "++" and "--" operations aren't necessarily atomic!
- Solution: Use system calls semget() and semop()!
- The following pseudocode protects a critical section:

```
wait(&s);
/* critical section */
signal(&s);
/* remainder section */
```

- What happens if S is initially 0? What happens if S is initially 8?

semget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

- Creates a semaphore set and initializes each element to zero
- Example:

int semID = semget(IPC_PRIVATE, 1, S_IRUSR | S_IWUSR);

- Like shared memory, ipcs and ipcrm can list and remove semaphores

semop()

- int semop(int semid, struct sembuf *sops, unsigned nsops);
- Increment, decrement, or test semaphore elements for a zero value
- From <sys/sem.h>
  sops->sem_num, sops->sem_op, sops->sem_flg
- If sem_op is positive, semop() adds value to the semaphore element and awakens the process waiting for the element to increase
- If sem_op is negative, semop() adds the value to the semaphore element and if <0, semop() sets to 0 and blocks until it increases
- If sem_op is zero and the semaphore element value is not zero, semop() blocks the calling process until the value becomes zero
- If semop() is interrupted by a signal, it returns –1 with errno = EINTR

Semaphore Example

```
struct sembuf    semWait[1] = {0,-1,0};
          semSignal[1] = {0,1,0};
int semID;

semop(semID,semSignal,1); /* init to 1 */

while((semop(semID,semWait,1) == -1) && (errno == EINTR))
    ;
{/* Critical Section */}

while((semop(semID,semSignal,1) == -1) && (errno == EINTR))
    ;
```