Shared Memory

Motivation

- 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 the client and server
- The only trick in using shared memory is synchronizing access to a given region among multiple processes -- if the server is placing data into a shared memory region, the client shouldn't try to access it until the server is done
- Often, semaphores are used to synchronize shared memory access (… *semaphores will be covered a few lectures from now*)
- not covered in Wang, lookup in Stevens (APUE)

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 identifiers, 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 the 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 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 segments to remain in the system even after the process dies -- a better technique is to use **`shmctl()`** to set up a shared memory segment to remove itself once the process dies (… *see next slide*)

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 (see **<sys/shm.h>**)
  - **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 detached from it — a process detaches 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 {size is %d}\n", size );

   if( (returnPtr=(char*) shmat( shmId, 0, 0 )) == (void*) -1 )
      Abort( "Failure on Shared Mem (shmat)" );

   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 the corresponding bytes of the file are read
- One advantage is that the contents of 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 zero,
  - **`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`** to 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 on open" );
   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 );
}
```

Semaphores

Motivation

- Programs that manage shared resources must execute portions of code called critical sections in a mutually exclusive manner. A common method of protecting critical sections is to use semaphores

- 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 Section: 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 <u>atomic operation</u> 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 <u>semaphore</u> is an integer variable with two atomic operations: <u>wait</u> and <u>signal</u>. 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 <u>decrements</u> the value of **S**. The *signal* operation <u>increments</u> the value of the semaphore variable.
- Some (flawed) pseudocode:

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

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

Semaphores (cont.)

- Three problems with the previous slide's **`wait()`** and **`signal()`**:
  - busy waiting is inefficient
  - doesn't guarantee bounded waiting
  - "**`++`**" and "**`--`**" operations aren't necessarily atomic!
- Solution: use system calls **`semget()`** and **`semop()`** (… *see next slide*)
- 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()

- Usage:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.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, **icps** and **ipcrm** can list and remove semaphores

semop()

- Usage: **int semop(int semid, struct sembuf *sops,**
  int nsops);
- Increment, decrement, or test semaphores 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 semaphore element and awakens processes 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**

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) )
   ;
```