# 2803ICT Systems Programming

## Inter-Process Communication (IPC)

Dr René Hexel

# Interprocess Communication

- How do separate processes communicate and share data?
  - Files
  - Pipes
  - Signals
  - Shared Memory
  - Message Queues
- How do you choose which one to use?
  - Consider message passing example

# Message Passing

- Message passing is normally performed using a pair of primitives:
  - send (destination, message)
  - receive (source, message)
- Communication requires synchronisation
  - Sender must send before receiver can receive
- Message passing may be blocking or non blocking

# Blocking or Non-blocking

- Blocking (Synchronous)
  - Send does not return control to the sending process until…
    - the message has been transmitted
    - OR an acknowledgment is received
  - Receive does not return until a message has been placed in the allocated buffer
- Nonblocking (Asynchronous)
  - Process is not suspended as a result of issuing a Send or Receive
  - Efficient and flexible
  - Difficult to debug

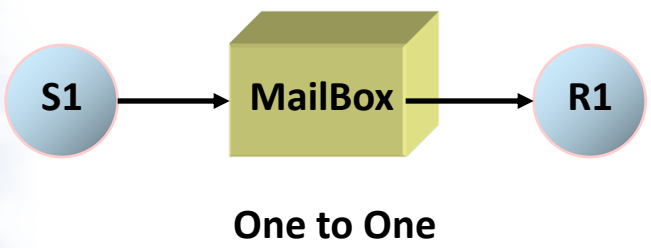
# Message Synchronisation Options

- *Rendezvous*
  - Both sender and receiver are blocked until message is delivered
  - Allows for tight synchronisation between processes.
- Nonblocking send
  - More natural for many tasks.
- Nonblocking send, blocking receive
  - Send and continue
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

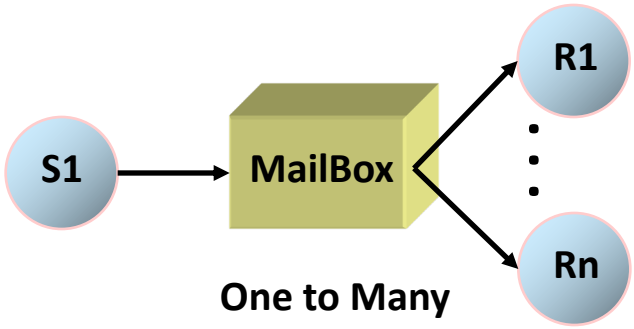
# Message Addressing

- Sending process needs to be able to specify which process should receive the message
- Direct addressing
  - Send() includes a specific ID of the destination process
  - Receive() could know ahead of time which process a message is expected from
  - Receiver could use source address to return a value after a message is received
- Indirect Addressing
  - Messages are sent to a shared queue (or mailbox)
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox

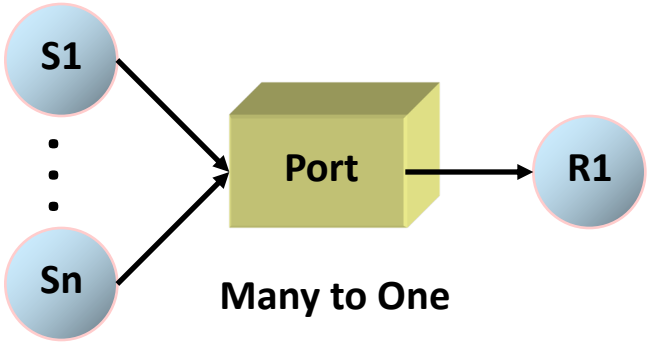
# Message Passing Models



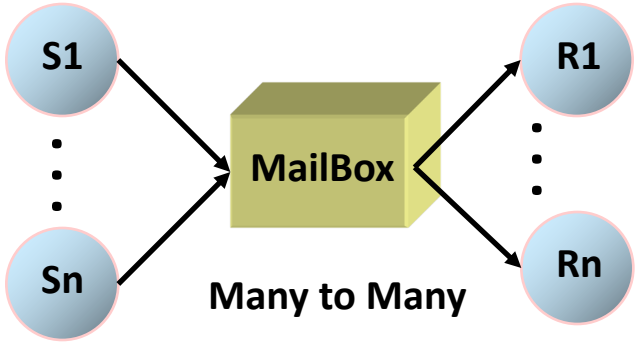

One to One




One to Many




Many to One




Many to Many

# File IO

- Server writes data to a file
- Client reads data from a file
- System calls
  - open() & close() - Both client and server
  - write() – Server
  - read() – Client
  - lseek() – Server
- Features
  - Easy to program
  - Supports multiple clients
  - Can be across different machines using NFS / file sharing
  - Uses standard file permissions

# File IO Synchronisation

**Problem**:

- What if 2 process want to write to the same file?
- What if 2 process want to read to the same file?
- What happens to the file pointer?
- How to coordinate reading and writing

**Answer**:

- Use File Locking to control read/write access to the file
- Reset file pointers for each reader

# File Lock Operations

| | Requested Lock Type | |
|---|---|---|
| **Existing Lock** | **Shared** | **Exclusive** |
| **None** | Granted | Granted |
| **Shared Lock** | Granted | Refused |
| **Exclusive Lock** | Refused | Refused |

| | Requested Operation | |
|---|---|---|
| **Existing Lock** | **Read** | **Write** |
| **None** | Succeeds | Succeeds |
| **Shared Lock** | Succeeds | Fails |
| **Exclusive Lock** | Only the lock owner has access, others are denied | |

# Pipes

- A pipe is an input/output channel that a process can use to pass information to other process .
- A pipe is similar to a shared file
  - it has a file pointer, a file descriptor, or both,
  - It can be read from or written to using standard file I/O functions (read, write, close)
  - Having multiple readers can cause havoc with the file pointer
- A pipe does not represent a specific file or device.
- Instead, it represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the operating system
- There are two kinds of pipes;
  - Anonymous – one way – character oriented
  - Named – bidirectional (Windows specific: Unix - FIFOs)

# Anonymous Pipes

- An anonymous pipe has 2 handles
  - Read handle
  - Write handle
- The program can use both sides (handles) of the pipe or close the one it does not need.
- A read on a pipe will block if the pipe is empty
- A write operation blocks until all bytes are written
- When all writers close the writing end trying to read will return EOF
- When all readers close the reading end trying to write will cause a `SIGPIPE` exception
- The pipe capacity is OS dependent at least 4096 bytes
- If a write fills the pipe further writing is blocked until data is read out of it.

# Named Pipe

- Called a FIFO in UNIX speak
- Can be accessed over a local network
- Full-duplex (bidirectional) message oriented
  - Reading process can read varying-length messages precisely as sent by the writing process
- Multiple, independent instances of a named pipe:
  - Several clients can communicate with a single server using the same instance
- Usage
  - When the last writer closes a named pipe an end of file is generated for the reader
  - If a named pipe is opened for read only it will block until another process opens it for writing
  - Opening a named pipe for writing will block until another process opens it for reading

# Shared Memory

- A user defined portion of memory shared between multiple processes in the form of a byte array
  - The shared memory segment is mapped to the same part of each process's virtual address space
  - One process creates a shared segment and gets a pointer.
  - Other processes can get a copy of the pointer to access the memory
- Server writes data to a shared part of memory
- Client reads data from shared memory
- Features
  - Fastest method as data shared not copied
  - System calls not needed for transfers
  - Supports multiple clients

# Mailslots (Win32 only)

- Broadcast IPC mechanism:
  - One-directional
  - Multiple writers/multiple readers (mainly: one-to-many)
  - Message delivery is unreliable
  - Can be located over a network domain
  - Message lengths are limited (w2k: < 426 byte)
- Operations on the mailslot:
  - Each reader (server) creates a mailslot
  - A writer (client) opens mailslot - will fail if there are no waiting readers
  - A writer's message can be read by all readers (servers)
- Client lookup: *\mailslot\mailslotname
  - Client will connect to every server in network domain

# Message Queues

- A linked list of messages stored within the kernel…
- Bidirectional - any process can read or write to any queue
- Messages can be read in order based on their field values
- Unix System wide limitations (OS dependent)
  - Largest message: 2048 bytes
  - Max queue size: 2048 bytes
  - Max number of messages in all queues: 40
- Not supported by MacOS X

- Windows uses message queues () for its GUI

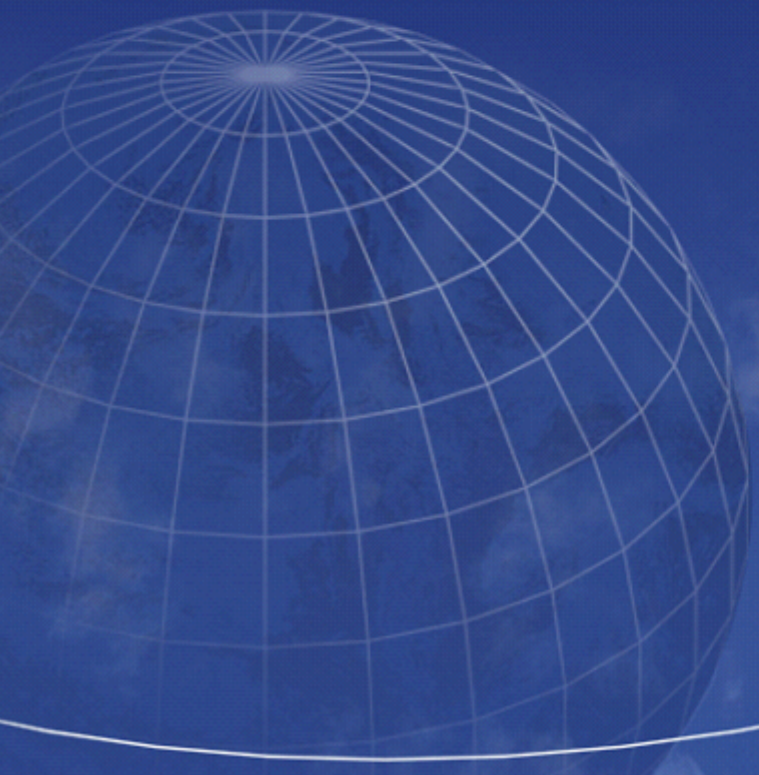
# Signals

- A signal is a one word message.
- Essentially events generated by another process or the kernel
  - User defined signals
  - Keyboard driver – eg Ctrl-C
  - Kernel – segmentation fault
- Example: What does ^C do?
  - The kernel sends a `SIGINT` to the process
  - The process terminates
- Process for using signals is
  - Client process defines a signal handler
  - Server process generates a signal

# Comparing IPC methods

| | Files | Pipes | Queues | Memory | Signals | Sockets |
|---|---|---|---|---|---|---|
| Range | LAN | LAN | Host | Host | Host | Internet |
| Speed | Slow | Slow | Medium | Fast | Fast | Slow |
| Capacity | Large | kbytes | 2kbytes | MBytes | 4 bytes | unlimited |
| Duplex? | Duplex | Simplex | Duplex | Duplex | Simplex | Duplex |
| Mode | M to M | 1 to 1 | 1 to 1 | M to M | 1 to 1 | 1 to 1 |
| Blocking | Possible | Yes | Yes | No | No | Possible |
| Random Access | Yes | No | Partly | Yes | No | No |
| Require Connection | No | No | Yes | Yes | No | Possible |
| Race Condition | Possible | | | Possible | | |

# Unix IPC



# File IO Locking

- Regions of files can be marked as being read or being written using the `fcntl()` function
- File locks under UNIX are by default not enforced – programs are free to ignore them
- Any number of processes can have a shared read lock
- Only one process can have exclusive write
- If there is an existing read lock, requests for a write locks are denied.
- If there is an existing write lock request for any other locks will be denied

# File IO Locking

```
int fcntl(int fd, int cmd, struct flock* pFlock);
```

- ***cmd*** is one of
  - `F_GETLK`
  - `F_SETLK` – set a lock
  - `F_SETLKW` – blocking form of *setlk* – waits until lock free
- ***pFlock*** is a pointer to a flock structure:

```
struct flock {
    short l_type;      //- F_RDLCK, F_WRLCK, F_UNLCK
    off_t l_start;     //- offset relative to whence
    short l_whence;    //- SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_len;       //- how much to lock, 0 = to EOF
    pid_t l_pid;       //- F_GETLK process ID
}
```

# File IO Locking Example

```
#include <fcntl.h>
#include <unistd.h>
struct flock lock;
lock.l_type = type ;  lock.l_whence = whence ;
lock.l_start = 0 ;    lock.l_len = 0 ;
lock.l_pid = getpid() ;
//- a shared lock on an entire file
lock.l_type = F_RDLCK; lock. l_whence = SEEK_SET
fcntl(fd, F_SETLKW, lock);
//- an exclusive lock on an entire file
lock.l_type = F_WRLCK; lock. l_whence = SEEK_SET
fcntl(fd, F_SETLKW, lock));
//- a lock on the _end_ of a file –
//- other processes may access existing records
lock.l_type = F_WRLCK; lock. l_whence = SEEK_END
fcntl(fd, F_SETLKW, lock);
```

# Anonymous Pipes

- Typical usage scenario
  - A process creates a pipe, and then does a…
  - `fork()` – so the child inherits the pipe handles
  - The parent closes the read handle (or vice versa)
  - The child closes the write handle
  - Parent can now write and child read

- A one-way pipe is created using:

```
int pipe(int pd[2]);
```

- For duplex communications you need 2 pipes

# Anonymous Pipe Example

```
#include <stdio.h>
#include <unistd.h>
#define ERROR(x)      { perror(x); exit(1); }
char buf[200];
int apipe[2];                    //- pipe handles

If (pipe(  apipe ) < 0) ERROR("can't get pipe");
while (fgets(buf, 200, stdin))    // read from stdin
{
    int len = strlen(buf);
    If (write(apipe[1], buf , len) != len)
          ERROR("cant write data to pipe");
    If (len = read(apipe[0], buf, 200))<0)
          ERROR("can't read data from pipe");
    Printf("data read from pipe: %s", buf);
}
```

# Pipes and Processes – 1

```
#define CHILD_MSG    “Feed Me!\n"
#define PAR_MSG      "testing..\n"
#define oops(m, x)   { perror (m); exit (x); }

int main()
{
  int pipefd[2];     //- the pipe
  int len;           //- for write
  char buf[BUFSIZ]; //- for read
  if ( pipe(pipefd) == -1 )
       oops("cannot get a pipe", 1);
  switch ( fork() )
  {
  case -1: oops ( "cannot fork", 2 );

  //-- continued next slide
```

# Pipes and Processes – 2

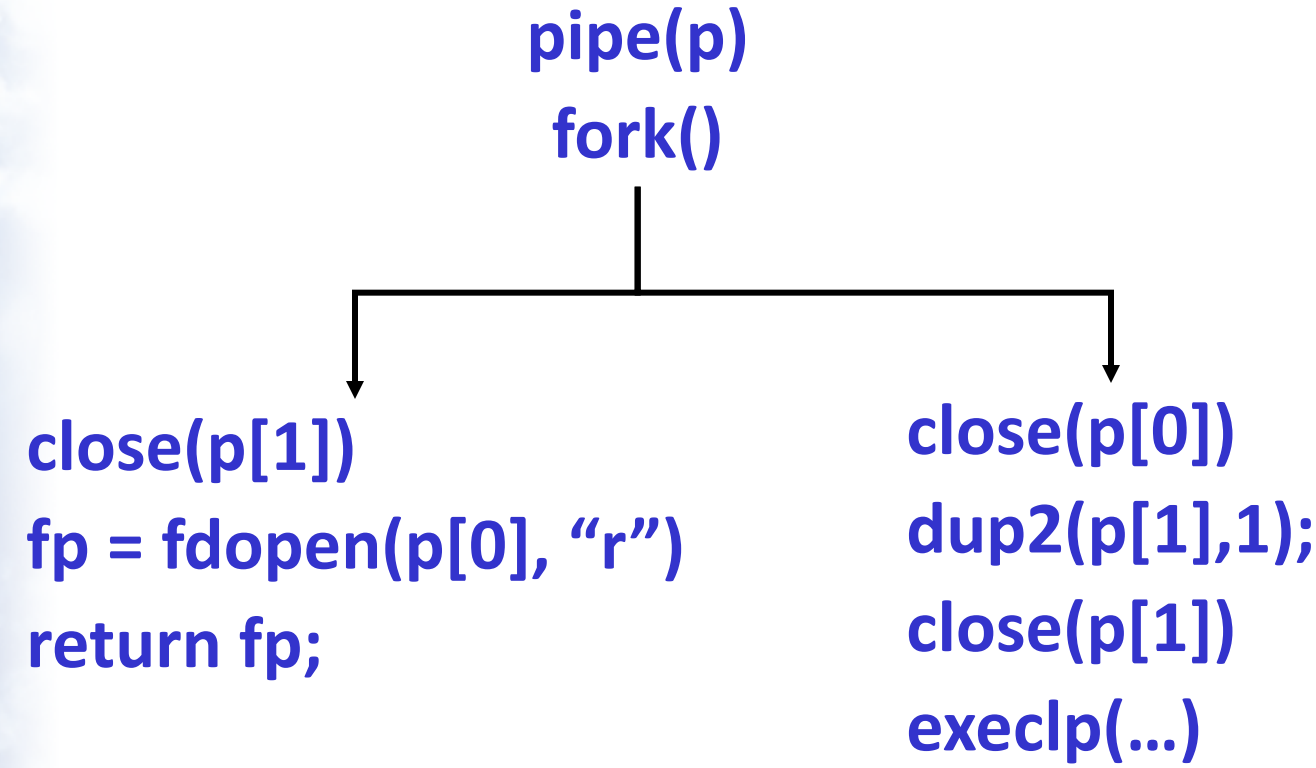
```
//-- from previous slide

//- child writes to pipe every 5 secs
   case 0:  len = strlen(CHILD_ MSG);
      while (1)
      {
          if (write( pipefd[l] , CHILD_ MSG, len) != len )
               oops("write", 3);
          sleep(5) ;
      }
//-- parent reads & writes to pipe
   default: len = strlen(PAR_MSG);
      while (1)
      {
          if ( write( pipefd[l] , PAR_MSG, len) ! =len )
               oops ("write", 4) ;
          sleep(1) ;
          int read_len = read( pipefd[O] , buf , BUFSIZ ) ;
          if ( read-len <= 0 ) break;
          write( 1 , buf, read_len );
      }
   }
}
```

# popen() and pclose()

- A typical use of a pipe involves
  - Creating a pipe
  - Forking a child
  - Closing the unused ends of the pipe
  - Executing a shell to execute a command
  - Waiting for the command to terminate
- All these steps are performed by popen()
  - fopen() creates a buffered stream to a file
  - popen() creates a buff'd stream to a process
- Supported on both windows and Unix

# popen() and pclose()

- Basic functionality of popen() for reading…




Remap p[1] to stdout

# popen() and pclose()

```
FILE* popen(char *cmd, char *type)
```

- ***cmd*** – is any command string to be executed
- ***type*** – one of “r”, “w” (never “a”)
- After opening a processes using popen() you must wait for it to terminate (otherwise it becomes a zombie process).

```
int pclose(FILE *fp)
```

- Example:

```
FILE* fp = popen(“ls –a”, “r”);
fgets(buf, len, fp);
pclose(fp);
```

# FIFOs (Named Pipes)

- Creating a FIFO

  ```
  int mkfifo(char* name, mode_t mode);
  ```

  - Mode – is the same as for open()
- deleting a FIFO (just like a regular file)

  ```
  unlink(char* fifoname)
  ```
- Listen for connections

  ```
  open(fifoname, O_RDONLY);
  ```

  - Blocks until a process opens the FIFO for writing
- Waiting for a connection to speak to

  ```
  open(fifoname, O_WRONLY);
  ```

  - Blocks until a process opens the FIFO for reading
- Communicate over the pipe

  ```
  read(); & write();
  ```

  - Call `close()` to send an end of file to the other party

# Reading Multiple Inputs

- When waiting for input from multiple files/pipes
  - You don't know which one will be ready first
  - Round robin polling (blocking) is inefficient
  - Better to watch the inputs using select()

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

- Using select()
  1. Make a list of the fd's you want to watch using FD_SET()
  2. Pick a timeout value
  3. Call select() with the fd list and timeout value
  4. Select will return when one of the fds is ready or the timeout elapses
  5. Use FD_ISSET() to check which fd is ready

# Reading Multiple Inputs Example

```
fd_set      watch_list;
struct timeval    timeout;

FD_ZERO(&watch_list);
FD_SET(fd1, &watch_list);
FD_SET(fd2, &watch_list);

timeout.tv_sec = 10;
timeout.tv_usec = 10;
int res = select(2, &watch_list, NULL, NULL, timeout);

if (res==0) printf("no inputs in 10 secs");
if (FD_ISSET(fd1, &watch_list)) printf("inputs on fd1");
if (FD_ISSET(fd2, &watch_list)) printf("inputs on fd2");
```

# Message Queues

```
#include <sys/msg.h>
```

- Create a new queue or open an existing one

```
int msgget(key_t key, int mode);
```

- ***Returns*** – message queue ID or –1 if unsuccessful
  - Each queue has a msqid_ds structure associated with it that stores data about the state of, and recent actions on the queue including #msgs, pid of last msgsnd, last send time etc
- ***Mode*** – permission mode bits
- ***Key*** - Can be either
  - `IPC_PRIVATE` - The kernel creates a new key that is guaranteed to be unique which will need to be passed to the other process
  - Any agreed (predefined) integer value – this does not need to be passed but you need to ensure that it is not already used,
  - Use `key_t ftok(char* path, int id);` to create a key from the path of an existing file – a bit safer than just picking a value

# Message Queues

- To add a message to end of the queue, use

```
int msgsnd(int msgid, void* ptr, size_t nbytes, int flag)
```

- ***ptr*** - points to a message in the form of…

```
struct myMsg { long mtype; char mtext[512];}
```

- ***Flag*** – can be either 0 or
  - IPC_NOWAIT –nonblocking option, if send fails returns EAGAIN
- Read a message from a queue

```
size_t msgrcv(int msgid, void* ptr,
    size_t nbytes, long type, int flag)
```

- ***type*** – specifies what kind of message to get
  - type == 0 – return first message on queue
  - type > 0 – the first msg that has the given type
  - type < 0 – the first msg with lowest value type

# Message Queues

- You can delete the queue, get its current state or overwrite its state using..

```
int msgctl(int msgid, int cmd, struct msqid_ds* buff)
```

- ***Cmd*** - can be one of
  - IPC_STAT- copy the msquid_id structure to ***buff***
  - IPC_SET - set the uid, gid, mode & qbytes from ***buff***
  - IPC_RMID - remove the msg queue from the system
- Example

```
struct myMsg msg = {1, "Test Message"};
int qid = msgget(IPC_PRIVATE, 0xFFFF);     //- make queue
msgsnd(qid, msg, sizeof(myMsg), IPC_NOWAIT);     //- send
int sz = msgrcv(qid, msg, sizeof(myMsg), 1, IPC_NOWAIT);
msgctl(qid, IPC_RMID, NULL);                 //- free queue
```

# Unix Shared Memory

- `#include <sys/types.h>`
- `#include <sys/ipc.h>`
- `#include <sys/shm.h>`
- To create a shared memory segment

  **`int shmget(key_t key, int size, int mode)`**
- ***Returns -*** the shared memory identifier associated with 'key'
- ***Key*** – same as in message queues
- ***Size*** – amount of memory to be shared in bytes
- ***Mode*** – permission mode bits

- To delete a shared memory segment

  **`int shmdt(void *addr);`**
- ***Addr*** – the location of the shared memory segment to delete (that was previously returned by shmat())

# Unix Shared Memory

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

- ***Returns*** – if successful a pointer to an existing shared memory segment to access or NULL on failure
- ***shmid*** – the identifier of the shared memory segment to access to access
- ***Flag*** – can be one of
  - SHM_RND - if set round the addr value
  - SHM_RDONLY - if set make the memory read only
- ***Addr*** – preferred address where the SM will be attached to
  - 0 - the kernel selects the first available address
  - Otherwise if SHM_RND is not specified at the given address
  - Otherwise at an address derived from the given address

# Unix Shared Memory

- You can delete the shared memory segment, get its current state or overwrite its state using..

  **int shmctl(int id, int cmd, struct shmid_ds *buf);**

- ***Cmd*** - can be one of
  - IPC_STAT- copy the shmid_ds structure to ***buf***
  - IPC_SET - set the uid, gid, & mode from the values in ***buf***
  - IPC_RMID – marks the shared memory segment for removal from the system when the last process has detached from it
- Example

```
//- create 1024 byte shared memory segment in process 1
int shmid = shmget(IPC_PRIVATE, 1024, 0xffff);
//- access the shared memory in another process using shmid
char *memptr = shmat( shmid, NULL, 0);
strcpy(memptr, “Test Msg”);      //- write to shared mem
shmdt( memptr );                 //- delete shared memory
```

# Unix Signals

- Signals are used to handle unpredictable events such as, program exceptions (crashes), timer and user events.
- When these events occur the operating system sends the corresponding process a signal. Each different type of event has a unique identification number from 0..32. (OS dependent)

| | | |
|---|---|---|
| **SIGINT** | **2** | CTRL+C Signal |
| **SIGQUIT** | **3** | Quit |
| **SIGILL** | **4** | Illegal instruction |
| **SIGFPE** | **8** | Floating-point error |
| **SIGKILL** | **9** | Terminate Process |
| **SIGBUS** | **10** | Bus Error |
| **SIGSEGV** | **11** | Memory access Violation |
| **SIGCHLD** | **20** | Child exit notification |
| **SIGUSR1** | **30** | User Defined Signal 1 |
| **SIGUSR2** | **31** | User Defined Signal 2 |

# Signal Handling

- A program can detect that these events have occurred and choose how to handle them.
- Processes tell the kernel how they want to respond to signals using the signal() function
- They can either
  - Accept the default action (die) by doing nothing about them
  - Ignore the signals
  - Ask that another function be run automatically whenever the event occurs that can do anything you like
- A process can also send any one of the 32 valid signal events to any executing program.

  int raise( int sig );
- ***Sig*** – The id number of the event to be raised

# Handling Signal

void (*signal( int *sig*, void (**func*) ( int ))) ( int );

- ***Sig*** – the name of the signal to be “handled”
- ***Func*** – can be either
  - A pointer to a user defined function takes an int & returns void
  - SIG_IGN – ignore and disregard specified signal
  - SIG_DFL – restore the OS kernel’s default handler
- ***Returns*** - a pointer to the previous event handling function assigned to the particular signal.
- When the signal handler returns, the calling process resumes execution immediately following the point at which it received the interrupt signal.
- Examples

```
signal(SIGINT, SIG_IGN);      //- ignore a signal
signal(SIGINT, SIG_DFL);      //- reset default handler
signal(SIGINT, functptr);     //- run custom function
```

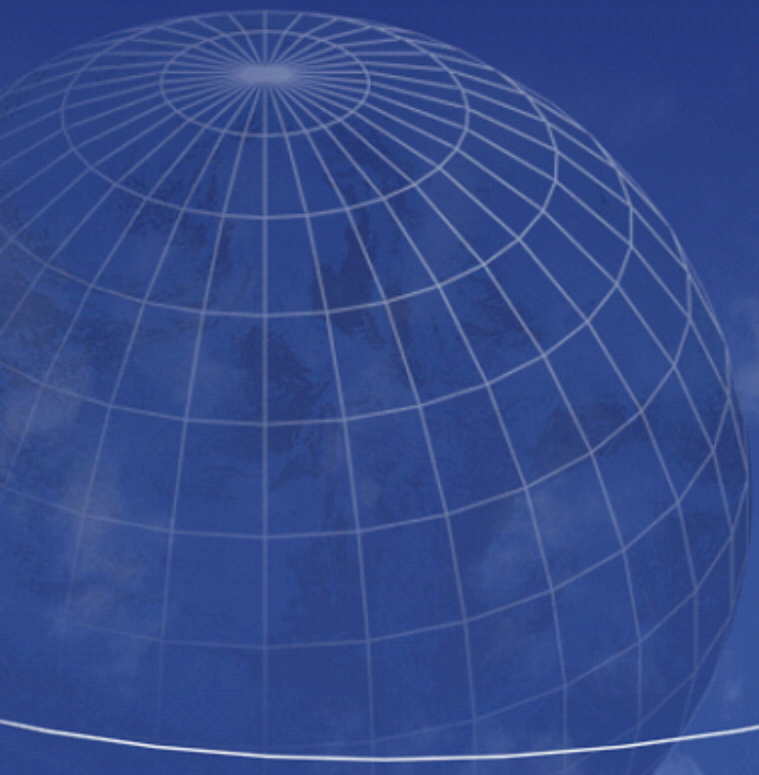
# Signal Handling Example

```
#include <signal.h>
#include <stdio.h>

void eventHandler(int);      //- forward declaration
int flag = 1;                //- evil global variable
void main()
{
    signal(SIGINT, eventHandler);
    while ( flag ) printf("Hello World");
}

void eventHandler (int sig)
{
    printf("aahhhh, you killed me !")
    flag = 0;
}
```

# Signal Handling Caveats

- You must be careful because when an interrupt occurs, your signal-handler function may get control when an operation is incomplete and state unknown
- Avoid using low-level or STDIO.H I/O routines (such as printf and fread).
- Do not call heap routines or any routine that uses the heap routines such as malloc etc.
- Avoid functions that generate system calls (e.g., _getcwd, time, execl(), etc).
- Signals are not very flexible and are not normally used for heavy duty interprocess communication.

# Win32 IPC



# File Locking

- With multiple users it is sometimes necessary to lock files so that only one user has access at any one time
- Windows can lock all or part of a files
- Lock can be read-only (shared) or read-write (exclusive)
- Locks belongs to the process and are NOT inherited
- Windows Locks are enforced
  - when using **`ReadFile`** or **`WriteFile`**
- You cannot create conflicting locks on a file
- You can specify whether you want to wait for a lock to become available, or return immediately, indicating whether it obtained the lock

# File Locks and Read/Write Access

- LockFile provides exclusive locks only

```
BOOL LockFile(HANDLE hFile,
        DWORD dwFileOffsetLow, DWORD dwFileOffsetHigh,
        DWORD nNumberOfBytesToLockLow,
        DWORD nNumberOfBytesToLockHigh );
```

- LockFileEx provides shared (RO) and exclusive locks

```
BOOL LockFileEx(HANDLE hFile,
                DWORD dwFlags, DWORD dwReserved,
                DWORD nNumberOfBytesToLockLow,
                DWORD nNumberOfBytesToLockHig,
                LPOVERLAPPED lpOverlapped);
```

# Unlocking File Locks

```
BOOL UnockFile(HANDLE hFile,
               DWORD dwFileOffsetLow,
               DWORD dwFileOffsetHigh,
               DWORD nNumberOfBytesToLockLow,
               DWORD nNumberOfBytesToLockHigh);
```

- Use the same parameters as used to lock the file

```
BOOL UnlockFileEx(HANDLE hFile,
                  DWORD dwReserved,
                  DWORD nNumberOfBytesToLockLow,
                  DWORD nNumberOfBytesToLockHig,
                  LPOVERLAPPED lpOverlapped);
```

- The unlock must use exactly the same range as a preceding lock – can not unlock a portion of the file

# File Locks Arguments

- ***hFile***
  - Handle of an open file which must have at least one of *GENERIC_READ* or *GENERIC_WRITE* access
- ***dwFlags*** – any combination of
  - LOCKFILE_EXCLUSIVE_LOCK, else requests a shared real-only block
  - LOCKFILE_FAIL_IMMEDIATELY, else waits for lock
- ***dwReserved*** must be zero
- ***dwFileOffsetHigh***, ***dwFileOffsetHigh***,
  - High and Low 32 bit words for starting address
- ***nNumberOfBytesToLockLow*** , ***nNumberOfBytesToLockHigh***
  - High & Low 32 bit words for range of bytes to lock
- ***lpOverlapped*** — Points to an `OVERLAPPED` data structure containing the start of the byte range

# File Locks Considerations

- Release locks as soon as possible
- Termination Handlers can be used to ensure the unlock is performed
- Locks cannot overlap existing locked regions
- You can lock beyond the range of a file's length – Useful when extending a file
- A failed read or write may take the form of a partially complete operation if only a portion of the read or write record is locked
- Need to call ***`GetLastError`*** to see what caused a read or write error

# File Locking Example

```
//- Open existing file.
HANDLE hFile = CreateFile(TEXT("one.txt"), ...);
if (hFile == INVALID_HANDLE_VALUE) ErrorExit();
HANDLE hAppend = CreateFile(TEXT("two.txt"), ...);
if (hAppend == INVALID_HANDLE_VALUE) ErrorExit();
//- Append the first file to the end of the second.
do {
      if (ReadFile(hFile, buff, sizeof(buff), &dwBytesRead, NULL))
      {
            dwPos = SetFilePointer(hAppend, 0, NULL, FILE_END);
            //- Lock the second file while writing.
            LockFile(hAppend, dwPos, 0, dwBytesRead, 0);
            WriteFile(hAppend, buff, dwBytesRead,
                                          &dwBytesWritten, NULL);
            UnlockFile(hAppend, dwPos, 0, dwBytesRead, 0);
      }
}
while (dwBytesRead == sizeof(buff));
```

# Win32 Posix Pipes

- **POSIX style pipes can be used in Win32:**

  **#include <io.h>**

  **int _pipe(int *pd*[2], unsigned int *psize*, int *mode*)**

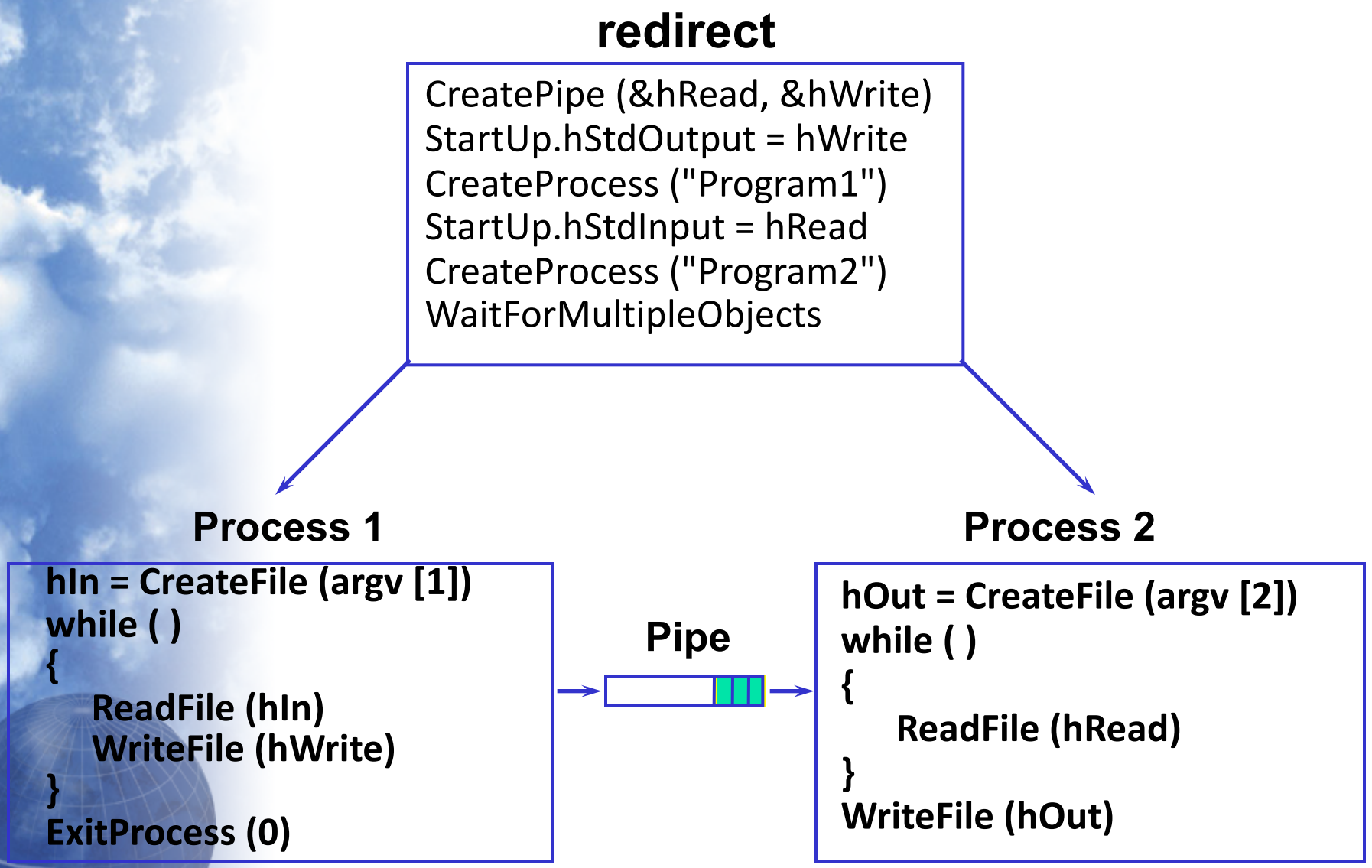
- Takes two extraparameters,
  - ***psize*** - the pipe capacity in bytes
  - ***mode*** – can be one of
    - _O_BINARY – used for binary data (ignore Ctrl-Z)
    - _O_TEXT – used for sending text data
- Use normal Win32 Posix.1 interface
  - read()
  - write()
  - Close()

# Win32 Pipes

**BOOL CreatePipe (PHANDLE phRead, PHANDLE phWrite, LPSECURITY_ATTRIBUTES lpsa, DWORD cbPipe)**

- ***phRead***
  - Address of variable where the reading HANDLE will be stored
- ***phWrite***
  - Address of variable where the reading HANDLE will be stored
- ***cbPipe***
  - The pipe size in bytes; use zero to get the default value
- Reading blocks if a pipe is empty will block; otherwise read will accept as many bytes as are in the pipe, up to the number specified in the **`ReadFile`** call
- Writing to a full pipe will block

# Pipe Example



# Pipe Example – 1

```
//- Create default size anonymous pipe, handles are inheritable.
if (!CreatePipe (&hReadPipe, &hWritePipe, &PipeSA, 0))
{
    fprintf(stderr, “Anonymous pipe create failed\n”); exit(1);
}
//- Set output handle to pipe handle, create first processes.
StartInfoCh1.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
StartInfoCh1.hStdError = GetStdHandle (STD_ERROR_HANDLE);
StartInfoCh1.hStdOutput = hWritePipe;
StartInfoCh1.dwFlags = STARTF_USESTDHANDLES;
if (!CreateProcess(NULL, (LPTSTR)Command1, NULL, NULL,
    TRUE, 0, NULL, NULL, &StartInfoCh1, &ProcInfo1))
{
    fprintf(stderr, “CreateProc1 failed\n”); exit(2);
}
CloseHandle (hWritePipe);
```

# Pipe Example – 2

```
//- Repeat (symmetrically) for the second process.
StartInfoCh2.hStdInput = hReadPipe;
StartInfoCh2.hStdError = GetStdHandle (STD_ERROR_HANDLE);
StartInfoCh2.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
StartInfoCh2.dwFlags = STARTF_USESTDHANDLES;
if (!CreateProcess(NULL, (LPTSTR)targv, NULL, NULL,TRUE,
    /*Inherit handles*/ 0, NULL, NULL, &StartInfoCh2, &ProcInfo2))
    { fprintf(stderr, "CreateProc2 failed\n"); exit(3); }
CloseHandle (hReadPipe);        //- Wait for both processes to end
WaitForSingleObject (ProcInfo1.hProcess, INFINITE);
WaitForSingleObject (ProcInfo2.hProcess, INFINITE);
CloseHandle (ProcInfo1.hThread);
CloseHandle (ProcInfo1.hProcess);
CloseHandle (ProcInfo2.hThread);
CloseHandle (ProcInfo2.hProcess);
```

# Named Pipes

- You can create multiple, independent instances of a named pipe
  - Several clients can communicate with a single server using the same pipe
  - Server can respond to a client using the same instance
- Pipe name can be accessed by systems on a network

# Named Pipes

HANDLE CreateNamedPipe(LPCTSTR *lpszPipeName*,
DWORD *fdwOpenMode*, DWORD *fdwPipeMode*,
DWORD *nMaxInstances*, DWORD *cbOutBuf*,
DWORD *cbInBuf*, DWORD *dwTimeOut*,
LPSECURITY_ATTRIBUTES *lpsa*)

- ***lpszPipeName*** - indicates the pipe name
  - Must be of the form `\\.\pipe\[path]pipename`
  - You cannot create a pipe on a remote machine
- ***fdwOpenMode*** – can be one of:
  - PIPE_ACCESS_DUPLEX == (GENERIC_READ | GENERIC_WRITE)
  - PIPE_ACCESS_INBOUND == GENERIC_READ
  - PIPE_ACCESS_OUTBOUND == GENERIC_WRITE
- The mode can also specify
  - FILE_FLAG_WRITE_THROUGH (not used with byte pipes)
  - FILE_FLAG_OVERLAPPED = use background IO

# Named Pipes

- ***fdwPipeMode*** has three mutually exclusive pairs of flags:
  - ***PIPE_TYPE_BYTE*** or ***PIPE_TYPE_MESSAGE***
    - Mutually exclusive
    - Use the same type value for all pipe instances
  - ***PIPE_READMODE_BYTE*** or ***PIPE_READMODE_MESSAGE***
    - Mutually exclusive
    - Reading stream of bytes or messages
    - ***PIPE_READMODE_MESSAGE*** requires ***PIPE_TYPE_MESSAGE***
  - ***PIPE_WAIT*** and ***PIPE_NOWAIT***
    - determine whether blocking mode is enabled
    - Generally use ***PIPE_WAIT***

# Named Pipes

- ***nMaxInstances*** — the number of pipe instances (simultaneous clients)
  - Specify this same value for every CreateNamedPipe call for a given pipe
  - PIPE_UNLIMITED_INSTANCES allows the OS to base the number on available system resources
- ***cbOutBuf*** and ***cbInBuf***
  - advise the OS on the required size of input and output buffers
- ***dwTimeOut***
  - default time-out period (in milliseconds) for the WaitNamedPipe function
- ***lpsa*** is as in all the other "*Create*" functions

# Clients And Server Using Named Pipes

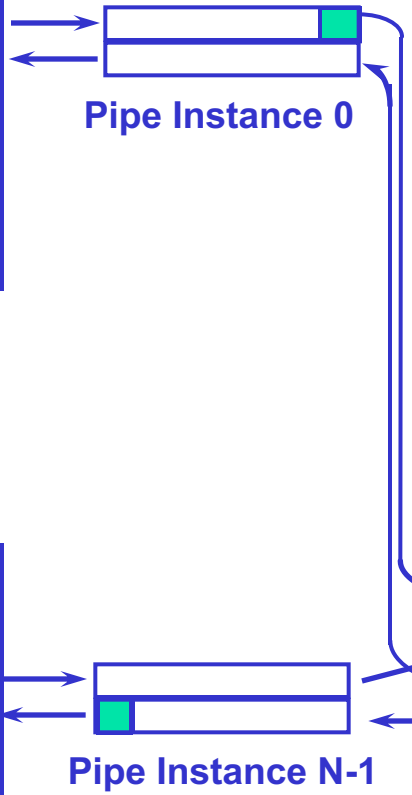
**Client 0**

```
h = CreateFile (PipeName);
while ( )
 {
    WriteFile (h, &Request);
    ReadFile (h, &Response)
     //-- Process Response
}
CloseHandle (h);
```

Up to N Clients

**Client (N-1)**

```
h = CreateFile (PipeName);
while ( )
{
    WriteFile (h, &Request);
    ReadFile (h, &Response)
     //-- Process Response
}
CloseHandle (h);
```




**Server**

```
//-- Create N instances --

for (i = 0; i < N, i++)
  h[i] = CreateNamedPipe (PipeName, N);

//-- Poll each pipe instance, get
//-- request, return response

i = 0;
while ( )
{
     if PeekNamedPipe (h [i])
     {
          ReadFile (h [i], &Request);
          //-- Create response --
          WriteFile (h [i], &Response);
     }
      i = i++ % N;
}
```

# Using Named Pipes

- ***CreateNamedPipe*(..)** creates the first instance of a named pipe and returns a handle
- The creating process is regarded as the server
- Client processes, possibly on other systems, open the pipe with ***CreateFile*(..)**
- Two problems with this approach:
  - It can only process one client request at a time. (use threads)
  - The server polls the pipe handles rather than waiting for a connection or a request. (inefficient)

# Using Named Pipes

- The program POLLs without blocking by using

  **BOOL PeekNamedPipe(HANDLE *hPipe*,**
  **LPVOID *lpvBuffer*, DWORD *cbBufSize*,**
  **LPDWORD *lpcbRead*, LPDWORD *lpcbAvail*,**
  **LPDWORD *lpcbRemains*);**

- Nondestructively reads any data in the pipe and returns immediately
- ***lpvBuffer*** – pointer where data will be stored or NULL
- ***cbBufSize*** – size of *lpvbuffer*
- ***lpcbRead*** – pointer to where the number of bytes read will be stored or NULL if no data is to be read
- ***lpcbRemains*** – pointer to where the number of bytes remaining in the message will be stored or NULL if no data is to be read

# Connecting to a Named Pipe

- A client can “connect” to a named pipe using `CreateFile` with the named pipe name
  - If the client and server are on the same machine, the name would be of the form:

    ```
    \\.\pipe\[path]pipename
    ```

  - If the client and server are on different machines, the name would be:

    ```
    \\servername\pipe\[path]pipename
    ```

  - It is faster to use the name “.” when the server is local, rather than the actual local machine name

# Named Pipe Status

- You can check the pipe status information
- **GetNamedPipeHandleState(…)**
  - returns information on whether the pipe is in blocking or non-blocking mode, whether it is message- or byte-oriented, the number of pipe instances, etc
- **SetNamedPipeHandleState(…)**
  - allows you to set the same state attributes
- **GetNamedPipeInfo(…)**
  - Returns whether the handle is the client or server end of a pipe, reading and writing buffer sizes, and the maximum number of pipe instances

# Named Pipe Functions – 1

- Combine the **WriteFile**, **ReadFile** client sequence into single operation (faster)
  **BOOL TransactNamedPipe(HANDLE** *hNamedPipe*,
  **LPVOID** *lpvWriteBuf*, **DWORD** *cbWriteBuf*,
  **LPVOID** *lpvReadBuf*, **DWORD** *cbReadBuf*,
  **LPDWORD** *lpcbRead*, **LPOVERLAPPED** *lpa*)
- Both output and input buffers are specified
- *lpvWriteBuf* – pointer to data to write to the pipe
- *lpvReadBuf* – pointer to data read from the pipe
- *cbWriteBuf* – size in bytes of the write buffer
- *cbReadBuf* – size in bytes of the read buffer
- *lpcbRead* – where the # bytes read will be stored
- *Lpa* – set to NULL if overlapped IO not used

# Named Pipe Functions – 2

- Combine the Create/Write/Read/Close client sequence into single operation (for short-lived comms)

  **BOOL CallNamedPipe(LPCTSTR** *lpszPipeName*,
  **LPVOID** *lpvWriteBuf*, **DWORD** *cbWriteBuf*,
  **LPVOID** *lpvReadBuf*, **DWORD** *cbReadBuf*,
  **LPDWORD** *lpcbRead*, **DWORD** *dwTimeOut*)

- *lpvWriteBuf* – pointer to data to write to the pipe
- *lpvReadBuf* – pointer to data read from the pipe
- *cbWriteBuf* – size in bytes of the write buffer
- *cbReadBuf* – size in bytes of the read buffer
- *lpcbRead* – where the # bytes read will be stored
- *dwTimeOut* – Time-out in msec for the connection or:
  - NMPWAIT_NOWAIT,
  - NMPWAIT_WAIT_FOREVER,
  - NMPWAIT_USE_DEFAULT_WAIT

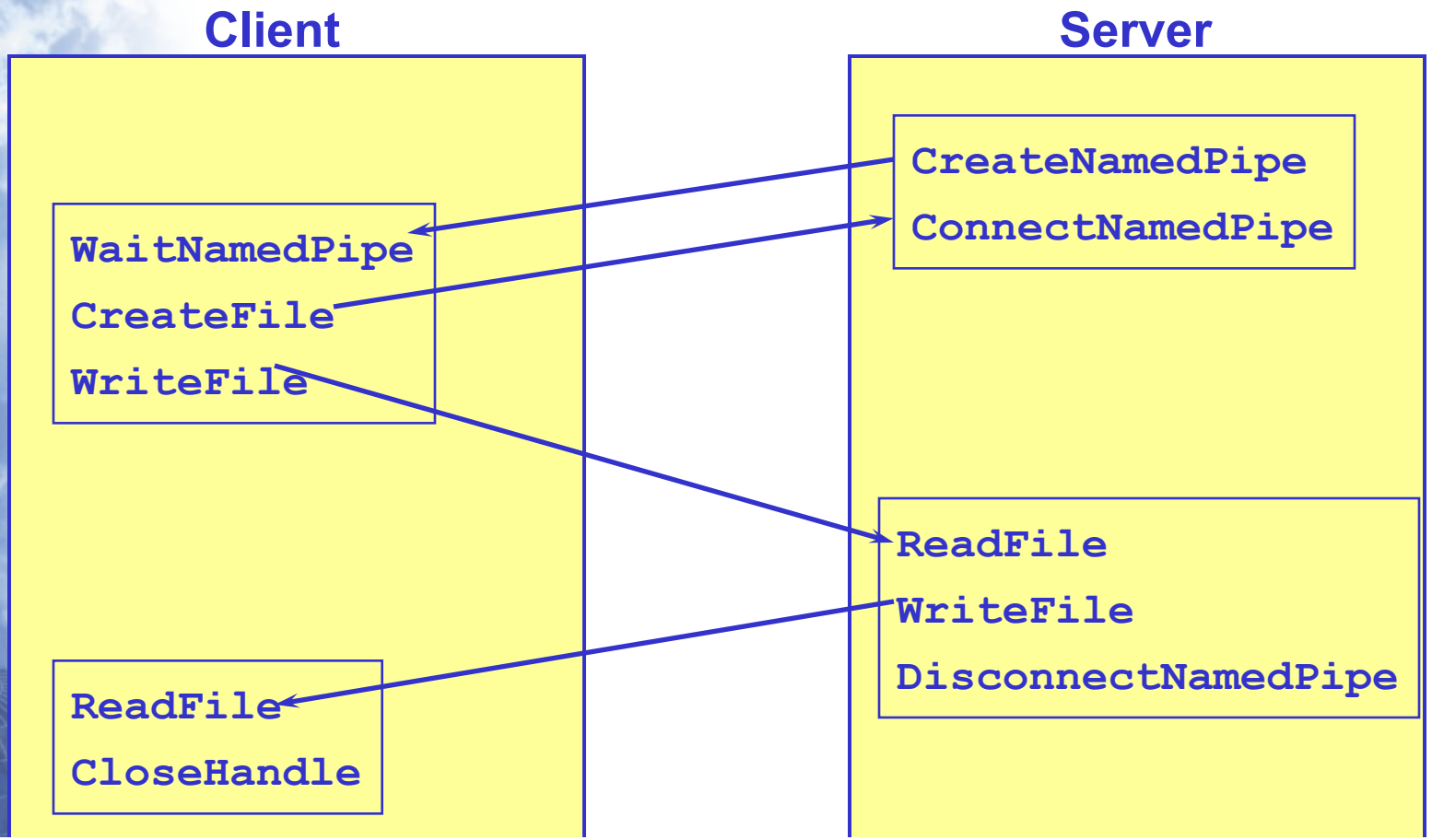
# Named Pipe Functions – 3

- For short-lived connections a named pipe server can wait for a client process to connect to its named pipe
  **BOOL ConnectNamedPipe(HANDLE *hNamedPipe*, LPOVERLAPPED *lpo*)**
- ***lpo*** – if set to `NULL`, will return as soon as the server has a connection from a client
- Returns –
  - TRUE if successful otherwise
  - FALSE if the client connected between the server's *CreateNamedPipe* call and the *ConnectNamedPipe* call and *GetLastError* returns ERROR_PIPE_CONNECTED
- The server end of a named pipe can be disconnected from a client process using
  **BOOL DisconnectNamedPipe(HANDLE *hNamedPipe*)**
- Perform **`ReadFile`**, **`WriteFile`** operations between connect and disconnect

# Named Pipe Functions – 4

- The client can wait for an instance of the named pipe to become available to connect to (ie server process has a pending *ConnectNamedPipeoperation* )
  **BOOL WaitNamedPipe (LPCTSTR** *lpszPipeName*, **DWORD** *dwTimeOut*)
- *Returns* – nonzero if a pipe is available before timeout
- *dwTimeOut* – Time-out in msec for the connection or:
  - NMPWAIT_NOWAIT,
  - NMPWAIT_WAIT_FOREVER,
  - NMPWAIT_USE_DEFAULT_WAIT

# Client-server Named Pipes Using A Short-lived Connection



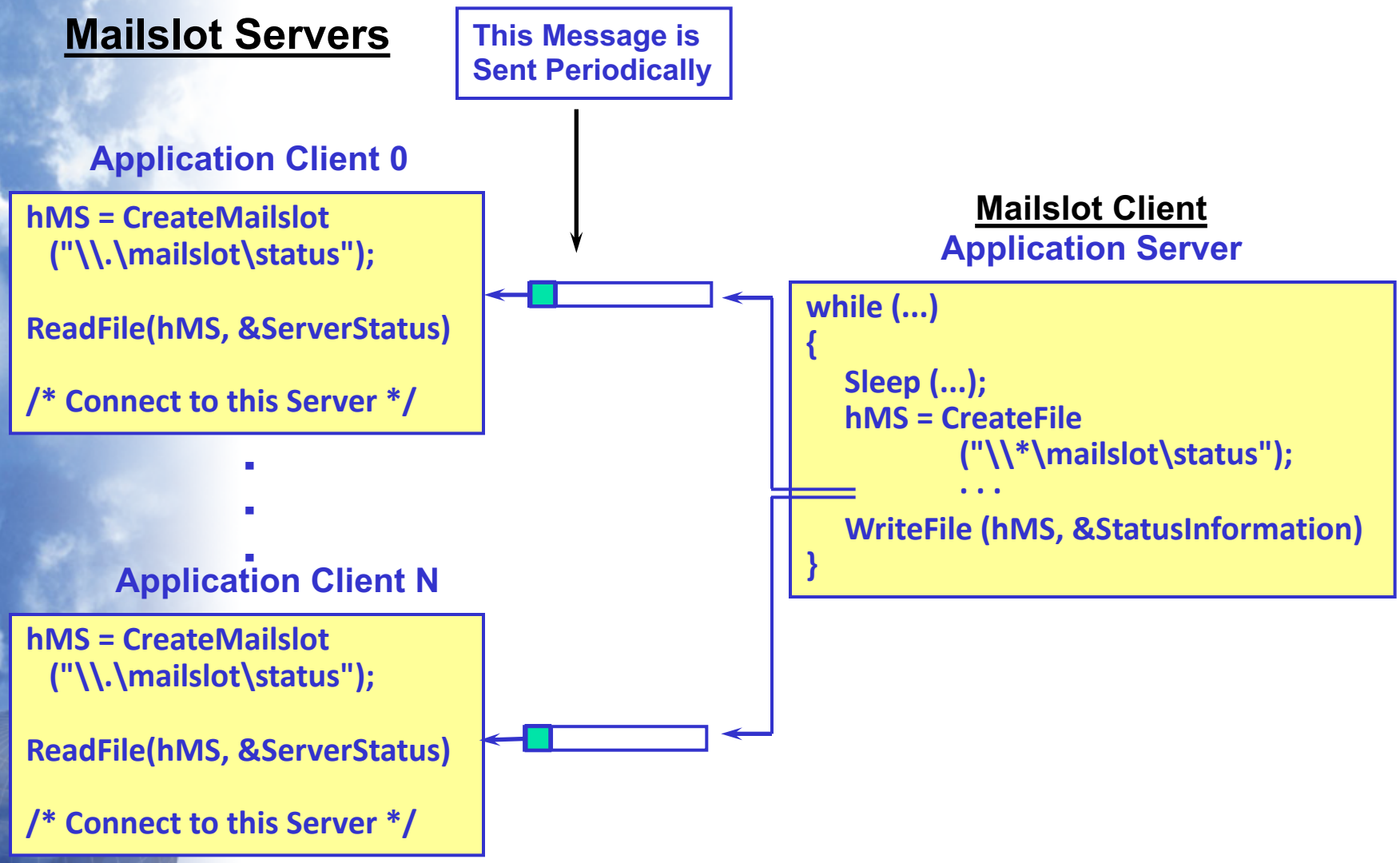
# MailSlots

- These are uni-directional broadcast mechanisms which behave differently named pipes
- Have names, so they can be used by unrelated processes for communication over a network domain
- Can have multiple readers and writers but often used in one-to-many configurations
- A writer ("client") does not know whether readers ("servers") actually received a message
- Message lengths are limited
- Usage Example:
  - An application server, acting as a mailslot client, periodically broadcasts its status (utilisation, database size, etc.) to the application clients
  - Any application client can receive the server's status by being a mailslot server (reader)

# MailSlots

- Usage Examples
  - A single mailslot client (writer) and multiple readers (servers)
  - With multiple application servers, you would have multiple mailslot readers and multiple writers
  - A bulletin board service might have a single mailslot reader and multiple writers
- A mailslot server creates a mailslot handle with **`CreateMailslot`**
- The server waits to receive a mailslot message with a **`ReadFile`** call
- A write-only client should open the mailslot with **`CreateFile`** and write messages with **`WriteFile`**
  - The open will fail (name not found) if no readers are waiting
- All servers receive the same message

# Using A Mailslot



# Creating MailSlots

**HANDLE CreateMailslot(LPCTSTR *lpszName*,**
**DWORD *cbMaxMsg*, DWORD *dwTimeout*,**
**LPSECURITY_ATTRIBUTES *lpsa*)**

- ***lpszName*** - a mailslot name of the forms:
  - \\.\mailslot\[path]name - for a local mailslot
  - \\hostname\mailslot\[path]name –on a remote machine
  - \\domainname\mailslot\[path]name - all mailslots in the domain
  - *\mailslot\[path]name - all mailslots in the current domain
- ***cbMaxMsg*** – Max size (in bytes) for messages a client can write, 0 means no limit
- ***dwTimeout*** — Number of milliseconds a read operation will wait
  - 0 causes an immediate return
  - MAILSLOT_WAIT_FOREVER is an infinite wait (no timeout)

# MailSlot Information

- Can get/set information about a mailslot using

  BOOL GetMailslotInfo(**HANDLE** *hMailslot*,
  **LPDWORD** *lpMaxMessageSize*,
  **LPDWORD** *lpNextSize*, //- the size of the next msg
  **LPDWORD** *lpMessageCount*, //- number of waiting msgs
  **LPDWORD** *lpReadTimeout*);

  **BOOL SetMailslotInfo( HANDLE** *hMailslot*, **DWORD** *lReadTimeout*);

- *lReadTimeout* – Time-out in msec for a read or:
  - MAILSLOT_WAIT_FOREVER

- The client must specify the FILE_SHARE_READ flag

# Win32 Shared Memory

- This is supported through ***File mapping***.
- File mapping is designed for
  - Sharing contents of files
  - Sharing memory blocks between two running processes.
- It works by mapping virtual memory space directly to normal files.
- The ‘copy’ of the file's contents is called the *file view*,
- The ‘copy’ is managed through a *file-mapping object*.
- Another process can create an identical file view in its own virtual address space by using the handle or name of the first process's file-mapping object.
- For multiple processes to share a memory block not associated with a file, create the file mapping object using the file handle: **(HANDLE)0xFFFFFFFF**

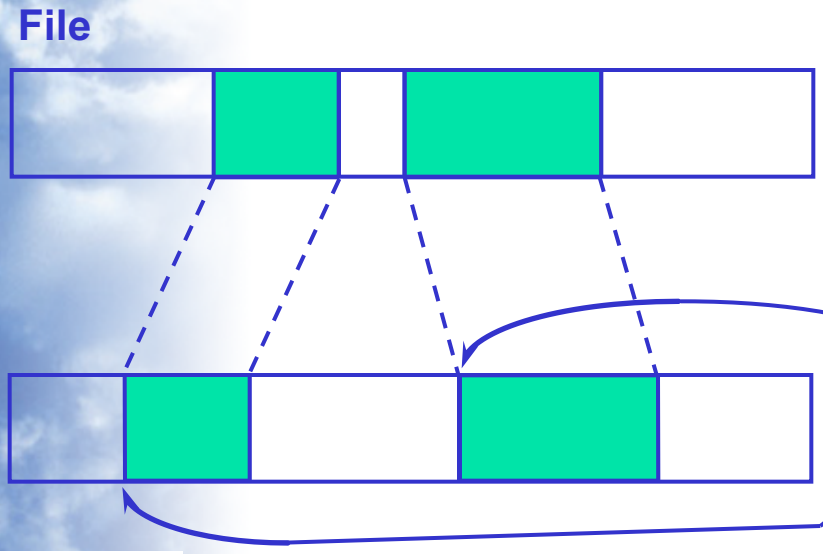
# Advantages of File Mapping

- In memory data structures
  - Data structures created in memory will be automatically saved in the file
  - No need to perform direct file I/O
- File processing
  - Faster
  - Can use in-memory algorithms to process files that are much larger than available physical memory
  - No need to manage buffers and the file data
- IPC
  - Multiple processes can share memory and the file views will be coherent
- There is no need to consume space in the paging file

# Using File Mapping

1. Open a file
2. If the file is new, set the file length either with CreateFileMapping or by using SetFilePoiner followed by SetEndOfFile
3. Map the file with CreateFileMapping or OpenFileMapping
4. Create one or more views with MapViewOfFile
5. Access the file through memory references
6. On completion, un-map the file and close handles for file mapping object as well as file
7. Normally only one process writes to the shared memory and the other reads the data, if both are going to write to the same memory location then they must synchronise their access using semaphores.

# Process Address Space Mapped To A File




**Program**

```
fH = CreateFile( );
mH = CreateFileMapping(fH);
    . . .
while ( )
{
   pRecA = MapViewOfFile(mH);
   pRecB = MapViewOfFile(mH);
   pRecB -> Data = pRecA -> Data;
            . . .
   UnmapViewOfFile(pRecA);
   UnmapViewOfFile(pRecB);
}
CloseHandle(mH);
CloseHandle(fH);
```

# Using File Mapping

**HANDLE CreateFileMapping(HANDLE** *hFile*,
**LPSECURITY_ATTRIBUTES** *lpsa*, **DWORD** *dwProtect*,
**DWORD** *dwMaxSizeHigh*, **DWORD** *dwMaxSizeLow*,
**LPCTSTR** *lpMapName*)

- Returns: A file mapping handle or NULL if unsuccessful
- ***hFile*** –Open file handle (access flags compatible with dwProtect)
- ***dwProtect*** — How you can access the mapped file:
  - `PAGE_READONLY` — Mapped region is read only
  - `PAGE_READWRITE` — Full access if `hFile` has both `GENERIC_READ` and `GENERIC_WRITE` access
  - `PAGE_WRITECOPY` — write changes to the paging file
- ***dwMaxSizeHigh*** and ***dwMaxSizeLow***
  - High and Low order words of the size of the mapping object;
  - 0 to keep the current file size. The file is extended if the current file size is smaller than the map size.
- ***lpMapName*** — Names the mapping object, so other processes can share the object. Normally `NULL`

# Using File Mapping

- You can open an existing file-mapping handle by specifying a mapping object name

  **HANDLE OpenFileMapping(DWORD** *dwDesiredAccess*, **BOOL** *bInheritHandle*, **LPCTSTR** *lpNameP*)

- Return: A file mapping handle or `NULL`
- ***dwDesiredAccess*** : one of
  - FILE_MAP_WRITE - Read-write access.
  - FILE_MAP_READ - Read-only access.
  - FILE_MAP_ALL_ACCESS - Read-write access
  - FILE_MAP_COPY - Read-write access with changes written to the paging file
- ***lpNameP*** : name of file mapping object
  - Collision Warning: The name space is shared
- Use CloseHandle() to destroy mapping handles

# Using File Mapping

- You can get a file view object using a handle
  **LPVOID MapViewOfFile(HANDLE** *hMapObject*,
  **DWORD** *dwAccess*, **DWORD** *dwOffsetHigh*,
  **DWORD** *dwOffsetLow*, **DWORD** *cbMapSize*)
- Returns:
  - The starting address of the mapped view or **`NULL`** if failed
  - Use MapViewOfFileEx to specify an existing address
- ***hMapObject***: handle to file mapping object
- ***dwDesiredAccess***: same as in **OpenFileMapping**
- ***dwOffsetHigh*** & ***dwOffsetLow***
  - High and Low order words of where the mapping is to begin rounded to the system word size
- ***cbMapSize***: how many bytes to map, or everything if 0
- Use **UnmapViewOfFile(LPVOID lpBaseAddress)** to release file views

# File Mapping Example

```
//- create 1k shmem with no security, read only access
HANDLE hMapFile = CreateFileMapping(0xFFFFFFFF,
                  NULL, PAGE_READWRITE, 0, 1024,
                  "MyFileMappingObject");
//- if we don't know handle we can get it using
HANDLE hMap = OpenFileMapping(FILE_MAP_READ,
              FALSE, "MyFileMappingObject");
//- Use the handle to access this shared memory
VOID* lpMapBase = MapViewOfFile(hMapFile,
                  FILE_MAP_READ, 0, 0, 0);
//- terminate the shared memory access :
UnmapViewOfFile(lpMapBase );
```

# File Mapping Limitations

- Note: The mapping view *does not* expand if the file size increases; you need to re-map
- 32-bit file-mapping limitations
  - With a large file (greater than 4GB) you cannot map everything into virtual memory space
  - Process data space is limited to 3GB but you cannot use all 3GB; available contiguous blocks will be smaller
  - When dealing with large files, you must create code that carefully maps and unmaps file regions as you need them
  - *No such limits in a 64-bit build – You can map very large files*

# Based Pointers

- If you use pointers in a mapped file region, they should be of type **`_based`**

  *type* **__based(** *base* **)** *declarator*

  - This address base will almost always be different the next time that file is mapped or a new view is created
- A conventional pointer refers to the virtual address
- A based pointer is an offset from a 32-bit pointer base.
  - The pointer should be based on the view address
- Example

```
int *pi = MapViewOfFile(...);
int __based(pi) *bpi;
*pi = 3;
bpi = pi;
int i = *bpi;
```


Synchronisation
Select / Poll, Semaphores and Locks

## Topics for Today:

1. Reading from multiple inputs

2. Multiple methods to transfer data

    * shared file
    * named pipe
    * shared memory

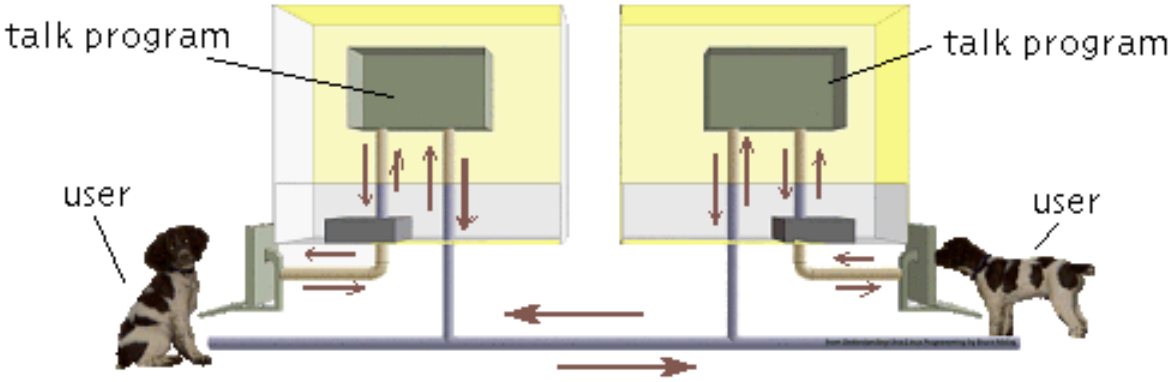
3. Multiple methods to share safely

    * file locks
    * semaphores

4. Discussion: Sharing a printer

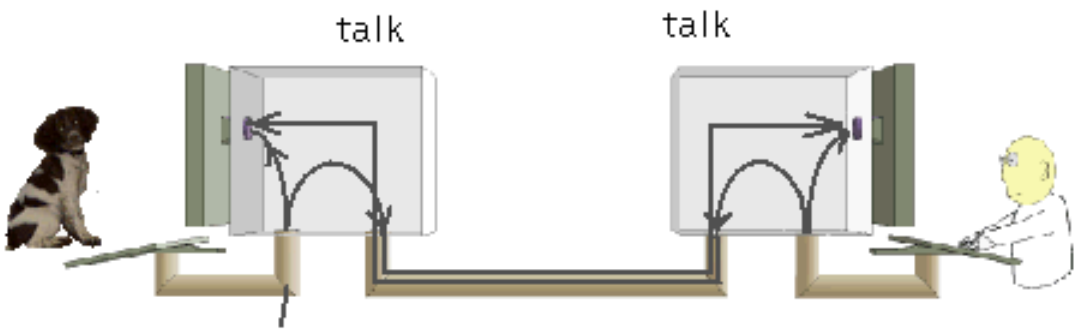
# Reading from Multiple Inputs: talk

## What talk does:




talk allows users to communicate by typing to each other. The users may be on different machines. Each keystroke is displayed on the user's screen and is sent to the screen of the other user.

## How talk works:



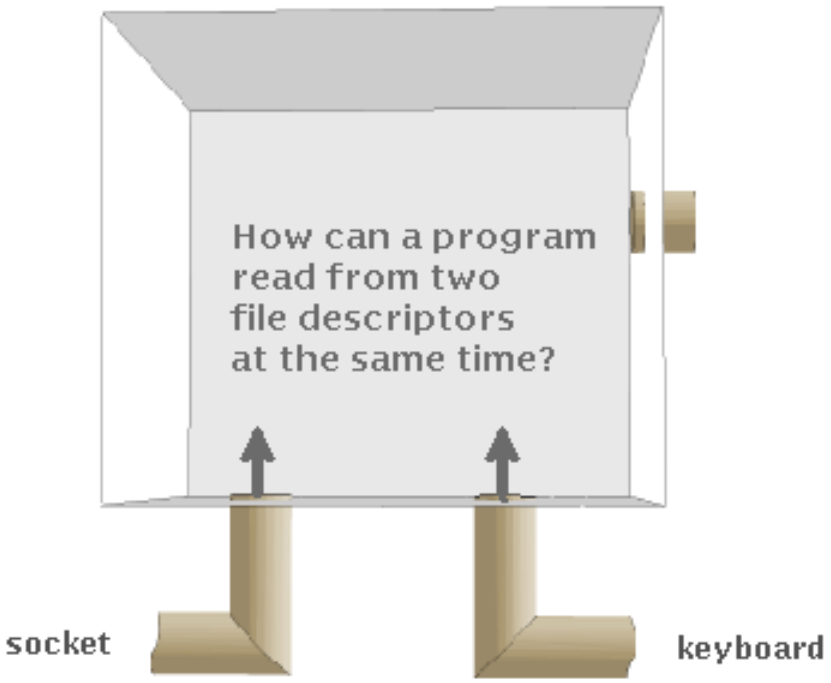

talk process reads input from keyboard and sends each char to the screen AND to the socket.

the process also reads input from the socket and sends each char to the screen

talk process reads input from keyboard and sends each char to the screen AND to the socket.

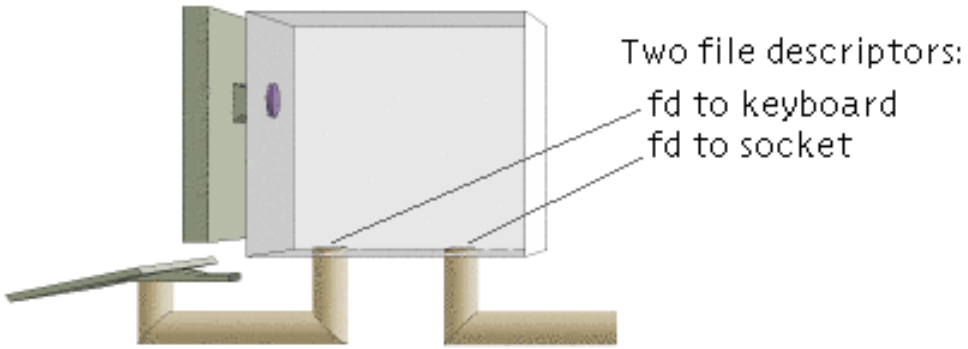
the process also reads input from the socket and sends each char to the screen

**talk receives characters on two file descriptors. Characters may show up at either at any time.**



## Watching multiple input sources: select()

1. Make a list of file descriptors you want to watch
2. Decide on a timeout value
3. Call select() with the list and the timeout value
4. select() returns when one (or more) of the file descriptors is ready OR the timeout period passes

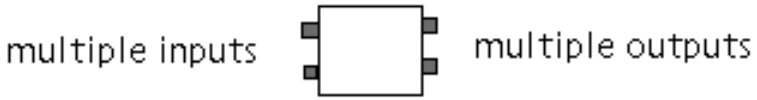


## using select() : sample code

```
fd_set          watch_us;      /* list of fds */
struct timeval  timeout;

FD_ZERO(&watch_us);           /* clear all bits */
FD_SET(kbd_fd, &watch_us);    /* add keyboard   */
FD_SET(sock_fd, &watch_us);   /* add socket     */

timeout.tv_sec = 10;          /* 10 sec timeout */
timeout.tv_usec = 0;          /* no microsecs   */

result = select(2,&watch_us,NULL,NULL,timeout);

if ( result == 0 )
    printf("no input in 10 seconds\n");

if ( FD_ISSET(kbd_fd, &watch_us) )
    printf("keyboard input\n");
if ( FD_ISSET(sock_fd, &watch_us) )
    printf("socket input\n");
```

## Concluding remarks about talk and select()

1. talk can use select to wait for input from the keyboard and from the socket. talk does not impose a timeout, so that value is set to 0.
2. talk is an example of a program that processes multiple (two in this case) sources of input at the same time.




**I/O multiplexer**

3. talk is more complicated than write because it communicates with a process; write sends data directly to the terminal of the other user. How do the two talk processes establish the connection?

## Choices in Unix Programming

### multiple system calls

select vs poll

poll is a different system call that does something similar to select. You could use poll to write talk. poll came from AT&T, select from BSD.

### multiple program designs

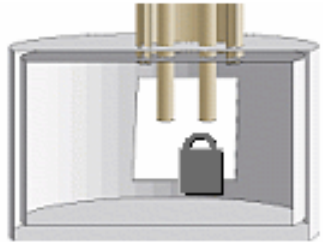
single thread vs multithread

You could write a multithreaded talk. One thread to read the keyboard, one to read the socket. Is this easier or harder? Why?

# Techniques for Locking Data

## File Locks

Unix provides system calls that allow processes to mark files as being read or as being written. Cooperating processes can check those locks and act accordingly.



## Semaphores

Unix provides system calls that allow processes to communicate through special shared variables called semaphores. Processes can use semaphores as locks. Processes use these variables to share safely any resource including shared memory segments.

## locking with semaphores

### A semaphore is like a key to the restroom.

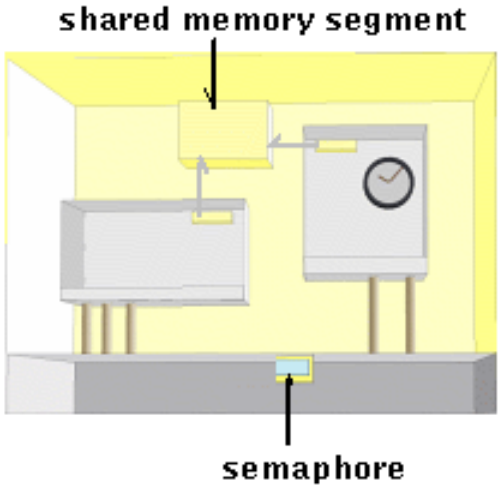
Some gas stations have a key to the restroom kept on a hook in the office. If you need to use the restroom, you take the key off the hook, use the restroom, and then return the key.

If the key is not on the hook, you wait until someone returns it, and then you take the key.

If several people want to use the rest room, only one can take the key when it is returned.

A semaphore is a special type of variable in the kernel that can hold some number of key-like values. A process can wait to take one (or more) of these things. When a process is done, it can return those things.

# Locking with Semaphores




Thus, one semaphore can be used by cooperating processes to make sure only one uses the segment at a time.

## A semaphore holds things

A semaphore is like a hook that holds a restroom key.

Multiple processes can have access to a semaphore.

## Processes wait to take one

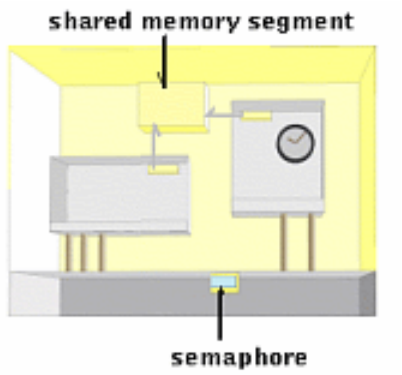
Just as customers at the gas station can wait to get the key, processes can wait to take the thing stored in the semaphore

Even if several are waiting, only one will get it.

## Processes can return them

When a process is done with the resource it returns the key.

# Locking with Semaphores




## Operations

| things | semaphores |
|---|---|
| get one | decrement semaphore |
| return one | increment semaphore |
| wait until gone | wait for zero |

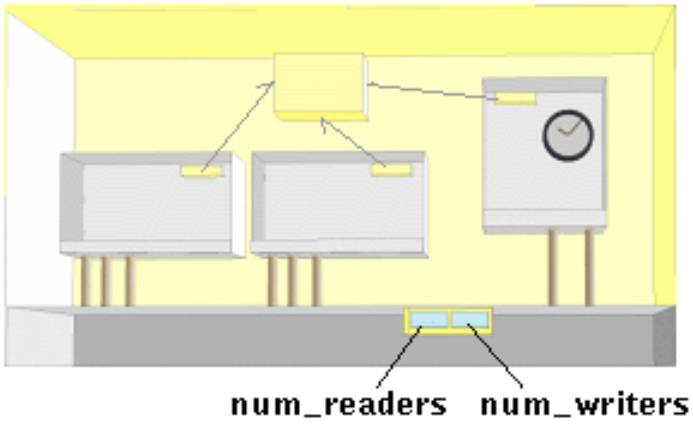
Operations on things (like keys) can be translated into corresponding operations on semaphores.

Semaphores can be decremented, incremented, and waited to equal zero. These operations correspond to taking things, returning things, and waiting until there are no things. Note: you can put new things in the storage container even if you did not 'take' them.

# Read and Write Locks with Semaphores

**Exclusive access may be too limiting**

With one semaphore, we can make sure only one process uses the resource at a time. But, it is OK for several readers to have access at once. By using two semaphores, we can create a system of read locks and write locks.

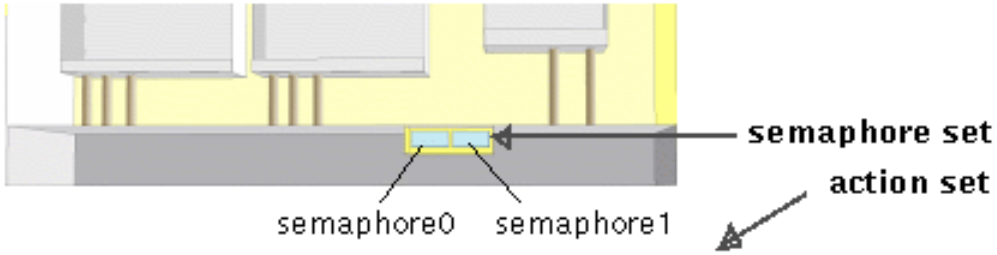



readers: wait for num_writers to be zero, then increment num_readers

writer: wait for num_readers to be zero, the increment num_writers

## Semaphore Sets and Action Sets

We translate the idea shown in the previous page into a semaphore set and two action sets.




readers: wait for sem1 to be zero, then increment sem0

writer: wait for sem0 to be zero, then increment sem1

readers:

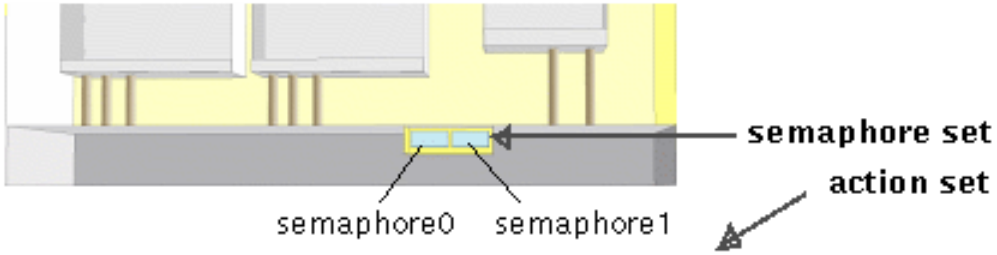
```
struct sembuf actset[2];
actset[0].sem_num=1;
actset[0].sem_op =0;
actset[1].sem_num=0;
actset[1].sem_num=+1;
```

writer:

```
struct sembuf actset[2];
actset[0].sem_num=0;
actset[0].sem_num=0;
actset[1].sem_num=1;
actset[1].sem_num=+1;
```

# Semaphore Sets and Action Sets : Releasing a Lock

We are using the number of things in a semaphore to stand for the number of processes using the resource. When a process is done, it just decrements the count.




readers: decrement semaphore 0

writer: decrement semaphore 1

readers:

```
struct sembuf actset[2];

actset[0].sem_num=0;
actset[0].sem_num=-1;
```

writer:

```
struct sembuf actset[2];

actset[0].sem_num=1;
actset[0].sem_num=-1;
```

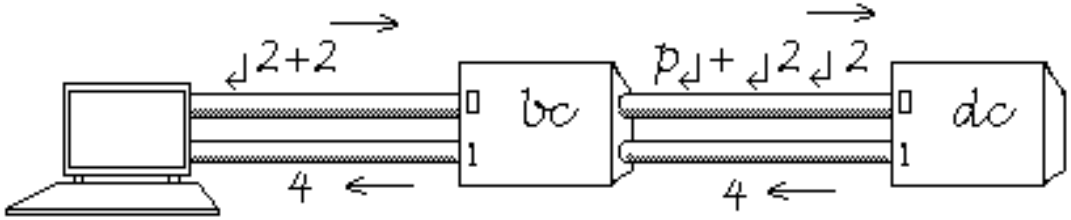
## (II) bc: A Unix Calculator (or is it?)

**what** bc is an arbitrary-precision calculator. It has variables, loops, functions, etc..

**demo** Supports normal order of operations

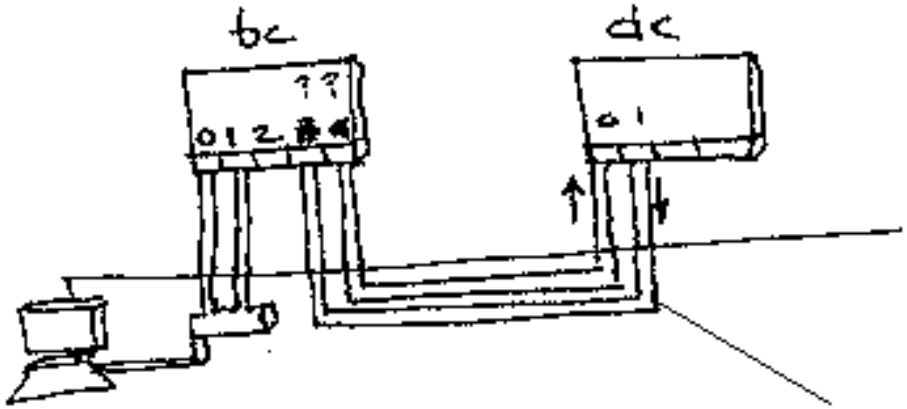
**but** use ps to see what's going on

**actually** bc does not do calculations, dc does calculations



# Coding bc: more pipe, fork, dup, exec

GOAL




```
int todc[2], fromdc[2];
pipe(todc); pipe(fromdc);
fork()
```

```
close(todc[0])
close(fromdc[1])
readline from user
convert
write(todc[1],cmd,len)
read(fromdc[0],reply)
printf to user
```