# Communication Between Threads and Processes

## 1. Memory, Shared Memory, and Memory-Mapped Files

- The easiest way for multiple threads to communicate is through memory. If two threads can access the same memory location, the cost of that access is little more than the memory latency of the system.
- Memory accesses still need to be controlled to ensure that only one thread writes to the same memory location at a time.
- A multithreaded application will share memory between the threads by default, so this can be a very low-cost approach.
- The only things that are not shared between threads are variables on the stack of each thread (local variables) .
- Sharing memory between multiple processes is more complicated. By default, all processes have independent address spaces, so it is necessary to pre configure regions of memory that can be shared between different processes.
- To set up shared memory between two processes, one process will make a library call to create a shared memory region. The call will use a unique descriptor for that shared memory. This descriptor is usually the name of a file in the file system.
- The create call returns a handle identifier that can then be used to map the shared memory region into the address space of the application. This mapping returns a pointer to the newly mappedmemory.
- This pointer is exactly like the pointer that would be returned by ***malloc()*** and can be used to access memory within the shared region.
- When each process exits, it detaches from the shared memory region, and then thelast process to exit can delete it. Fig.15 shows the rough process of creating anddeleting a region of shared memory.

```
ID = Open Shared Memory( Descriptor );
Memory = Map Shared Memory( ID );
...
Memory[100]++;
...
Close Shared Memory( ID );
Delete Shared Memory( Descriptor );
```

Fig.15.creating and deleting a shared memorysegment

## 2. Condition Variables

- Condition variables communicate readiness between threads by enabling a thread to be woken up when a condition becomes true. Without condition variables, the waiting thread would have to use some form of polling to check whether the condition hadbecome true.
- Condition variables work in conjunction with a mutex. The mutex is there to ensurethat only one thread at a time can access the variable.
- For example, the ***producerconsumer***model can be implemented using condition variables.
- Suppose an applicationhas one producer thread and one consumer thread. The producer adds data onto aqueue, and the consumer removes data from the queue. If there is no data on the queue,then the consumer needs to sleep until it is signaled that an item of data has been placedon the queue.
- Fig.16 shows the pseudocode for a ***producer*** thread adding an itemonto the queue.

```
Acquire Mutex();
Add Item to Queue();
If ( Only One Item on Queue )
{
        Signal Conditions Met();
}
Release Mutex();
```

Fig.16 Producer Thread Adding an Item to the Queue

- The producer thread needs to signal a waiting consumer thread only if the queue was empty and it has just added a new item into that queue.
- If there were multiple items already on the queue, then the consumer thread must be busy processing those items and cannot be sleeping.
- If there were no items in the queue, then it is possible that the consumer thread is sleeping and needs to be woken up.

Fig 17 shows the pseudocode for the consumer thread

```
Acquire Mutex();
Repeat
Item = 0;
```

```
If ( No Items on Queue() )
{
        Wait on Condition Variable();
}
If (Item on Queue())
{
        Item = remove from Queue();
}
Until ( Item != 0 );
Release Mutex();
```

Fig 17 Code for Consumer Thread Removing Items from Queue

- The consumer thread will wait on the condition variable if the queue is empty. When the producer thread signals it to wake up, it will first check to see whether there is anything on the queue.
- It is quite possible for the consumer thread to be woken only to find the queue empty; it is important to realize that the thread waking up does not imply that the condition is now true.
- If there is an item on the queue, then the consumer thread can handle that item; otherwise, it returns to sleep.

## 3. Signals and Events

- Signals are a UNIX mechanism where one process can send a signal to another process and have a handler in the receiving process perform some task upon the receipt of the message.
- Many features of UNIX are implemented using signals. Stopping a running application by pressing ^C causes a SIGKILL signal to be sent to the process.
- Windows has a similar mechanism for events. The handling of keyboard presses and mouse moves are performed through the event mechanism. Pressing one of the buttons on the mouse will cause a click event to be sent to the target window.
- Signals and events are really optimized for sending limited or no data along with the signal, and as such they are probably not the best mechanism for communication when compared to other options. Listing 4.20 shows how a signal handler is typically installed and how a signal can be sent to that handler. Once the signal handler is installed, sending a signal to that thread will cause the signal handler to be executed.

```
voidsignalHandler(void *signal)
{
...
}
int main()
{
        installHandler( SIGNAL, signalHandler );
        sendSignal( SIGNAL );
}
```

Fig 18 Installing and Using a Signal Handler

## 4. Message Queues

- A message queue is a structure that can be shared between multiple processes. Messages can be placed into the queue and will be removed in the same order in which they were added.
- Constructing a message queue looks rather like constructing a shared memory segment.
- The first thing needed is a descriptor, typically the location of a file in the file system. This descriptor can either be used to create the message queue or be used to attach to an existing message queue. Once the queue is configured, processes can place messages into it or remove messages from it. Once the queue is finished, it needs to be deleted.
- Fig 19 shows code for creating and placing messages into a queue. This code is also responsible for removing the queue after use.

```
ID = Open Message Queue Queue( Descriptor );
Put Message in Queue( ID, Message );
        ...
Close Message Queue( ID );
Delete Message Queue( Description );
```

Fig19. Creating and Placing Messages into a Queue

- Fig 20 shows the process for receiving messages for a queue. Using the descriptor for an existing message queue enables two processes to communicate by sending and receiving messages through the queue.

```
ID=Open Message Queue ID(Descriptor);
```

```
Message=Remove Message from Queue(ID);
    ...
Close Message Queue(ID);
```

Fig 20 Opening a Queue and Receiving Messages

## 5. Named Pipes

- UNIX uses pipes to pass data from one process to another.
- For example, the output fromthe command ***ls***, which lists all the files in a directory, could be piped into the ***wc*** command,which counts the number of lines, words, and characters in the input.
- The combinationof the two commands would be a count of the number of files in the directory.
- Named pipes provide a similar mechanism that can be controlled programmatically.
- Named pipes are file-like objects that are given a specific name that can be sharedbetween processes.
- Any process can write into the pipe or read from the pipe. There isno concept of a "message"; the data is treated as a stream of bytes. The method for usinga named pipe is much like the method for using a file: The pipe is opened, data is writteninto it or read from it, and then the pipe is closed.
- Fig 4.21 shows the steps necessary to set up and write data into a pipe, beforeclosing and deleting the pipe. One process needs to actually make the pipe, and once ithas been created, it can be opened and used for either reading or writing.
- Once theprocess has completed, the pipe can be closed, and one of the processes using it shouldalso be responsible for deleting it.

```
Make Pipe( Descriptor );
ID = Open Pipe( Descriptor );
Write Pipe( ID, Message, sizeof(Message) );
...
Close Pipe( ID );
Delete Pipe( Descriptor );
```

Fig 21 Setting Up and Writing into a Pipe

- Fig 22 shows the steps necessary to open an existing pipe and read messages fromit. Processes using the same descriptor can open and use the same pipe for communication.

```
ID=Open Pipe( Descriptor );
Read Pipe( ID, buffer, sizeof(buffer) );
...
Close Pipe( ID );
```

Fig 22. Opening an Existing Pipe to Receive Messages