G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Interprocess Communication

Dr Jason Atkin

This lecture

- Locks and deadlocks
- Inter-process communication
  - Windows messages
    - Synchronous
    - Asyncrononous
  - Sockets

Reminder: Test-and-Set instruction

The Test-and-Set (atomic) instruction effectively executes the function

```
bool TS(bool lock)
{
    bool v = lock;
    lock = true; // Set true
    return v; // Old lock value
}
```

Stops the other thread looking at it before we set it

See InterlockedExchange: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683590%28v=vs.85%29.aspx

Reminder: Test-And-Set spin-lock

```
// Process 1

init1;
while(true) {
    while(TS(lock)) ;
    crit1;
    lock = false;
    rem1;
}
```

```
// Process 2

init2;
while(true) {
    while(TS(lock)) ;
    crit2;
    lock = false;
    rem1;
}
```

Key point: this worked to avoid interference from other threads

Two-locks, two processes

```
// Process 1

init1;
while(true) {
   while(TS(lock1)) ;
   crit1a;

   while(TS(lock2)) ;
   crit1b;
   lock2 = false;

   lock1 = false;
   rem1;
}
```

```
// Process 2

init1;
while(true) {
   while(TS(lock1)) ;
   crit2a;

   while(TS(lock2)) ;
   crit2b;
   lock2 = false;

   lock1 = false;
   rem1;
}
```

- Two-processes Test and set spinlock – does it still work?

What about this case?

```
// Process 1

init1;
while(true) {
   while(TS(lock1)) ;
   crit1a;

   while(TS(lock2)) ;
   crit1b;
   lock2 = false;

   lock1 = false;
   rem1;
}
```

```
// Process 2

init1;
while(true) {
   while(TS(lock2)) ;
   crit2a;

   while(TS(lock1)) ;
   crit2b;
   lock1 = false;

   lock2 = false;
   rem1;
}
```

Any potential problems here?

Livelock! Perhaps 100% CPU used

```
// Process 1

init1;
while(true) {
   while(TS(lock1)) ;
   crit1a;

   while(TS(lock2)) ;
   crit1b;
   lock2 = false;

   lock1 = false;
   rem1;
}
```

```
// Process 2

init1;
while(true) {
   while(TS(lock2)) ;
   crit2a;

   while(TS(lock1)) ;
   crit2b;
   lock1 = false;

   lock2 = false;
   rem1;
}
```

Why did this happen? Can we avoid it?

Livelock: potential solution

```
// Process 1

init1;
while(true) {
   while(TS(lock1)) ;
   crit1a;

   while(TS(lock2)) ;
   crit1b;
   lock2 = false;

   lock1 = false;
   rem1;
}
```

```
// Process 2

init1;
while(true) {
   while(TS(lock1)) ;
   while(TS(lock2)) ;
   crit2a;
   while(TS(lock1)) ;
   crit2b;
   lock1 = false;

   lock2 = false;
   rem1;
}
```

Simple trick: always lock in the same order

Self-livelock example

```
init1;
while(true) {
    while(TS(lock1)) ;
    crit1a;
    func();
    lock = false;
    rem1;
}
```

```
int func()
{
    while(TS(lock1)) ;
    crit1b;
    lock = false;
}
```

- Some function gets a lock so that it can go into a critical section
- While in there it needs to do some other (minor?) thing, so calls a function to do so
- The function it calls needs to alter some resource which needs protecting from interference between threads
- So needs to be put into a critical section
- So the function attempts to get a lock before modifying the value
- The test-and-set lock will spin forever

Deadlock and Self-deadlock

- Exactly the same case can happen with deadlock as for livelock
  - Use a lock which blocks the thread rather than a spinlock
- Self-deadlock is often harder to achieve
  - Mutual exclusion objects often check the owner for this reason
- Two (or more) process deadlock is as easy to achieve accidentally as with livelock
- Consider using "WaitForMultipleObjects"

Self-deadlock

```
init1;
while(true) {
    LOCK
    crit1a;
    func();
    crit1c;
    UNLOCK
    rem1;
}
```

```
int func()
{
    LOCK
    crit1b;
    UNLOCK
}
```

- Lock mechanism tries to get a lock in main function
- And asks again in the sub-function

- **Locks that know about ownership:**
  - E.g. CRITICAL_SECTION or Mutex objects in windows
  - Lock automatically gained
  - Have to match locks and unlocks to avoid unlocking it too early
    - Before `crit1c` in this example

- **Locks with no concept of ownership:**
  - E.g. binary semaphore
  - Thread deadlocked at second p()

Inter-process communication

Interprocess Communication So Far

- We have so far seen various methods of interprocess communication

- Windows messages
  - PostMessage, SendMessage
- Shared data
  - Map the memory into both address spaces
  - Make it volatile
- Mutex locking
- Semaphore counts

Inter Process Communication

- Two conceptual types

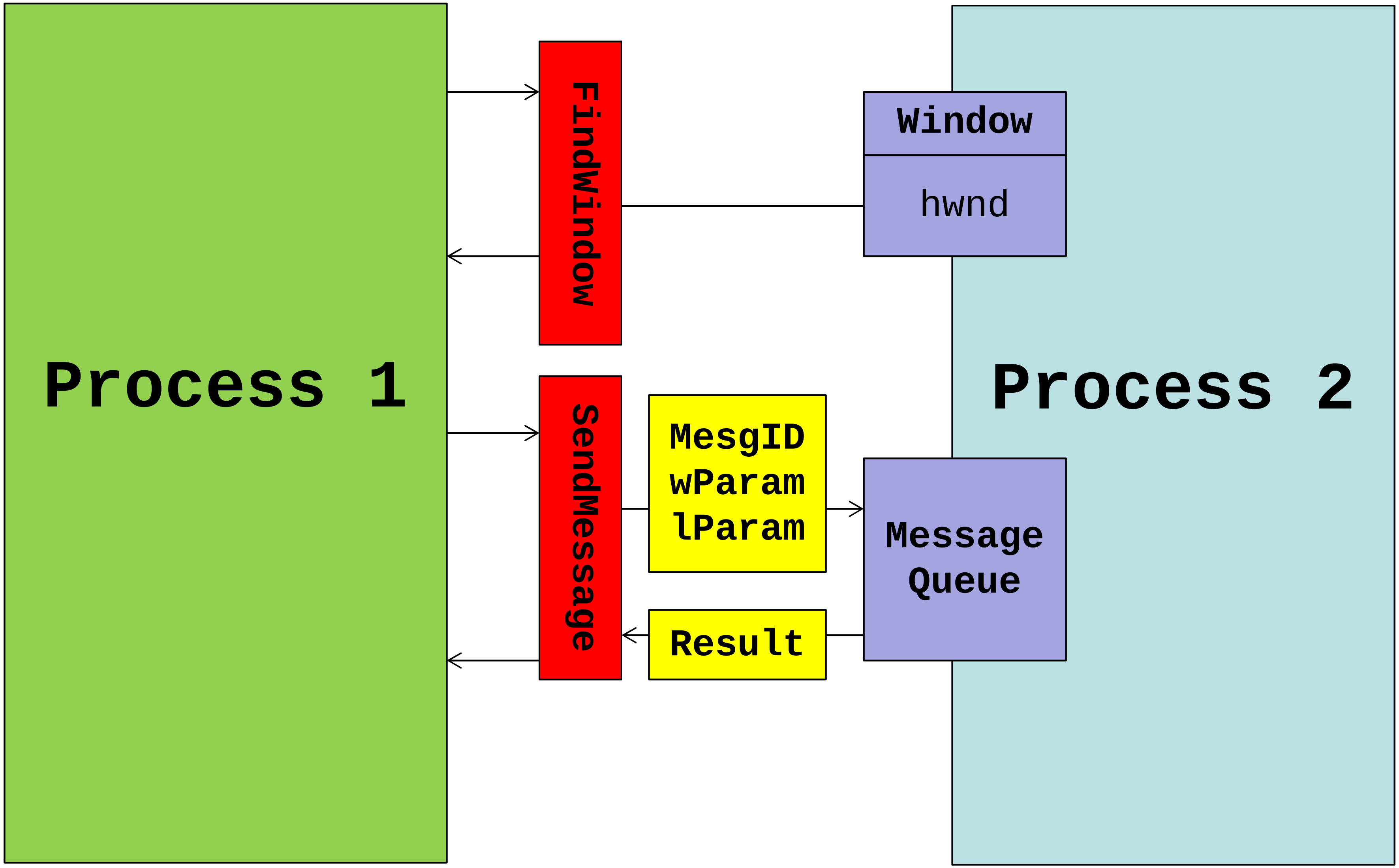
- Asynchronous communication
  - Leave a message
  - It may get back to you later (or not)

- Synchronous communication
  - Ask a question, get a response
  - Make a request to do something, wait for it to happen

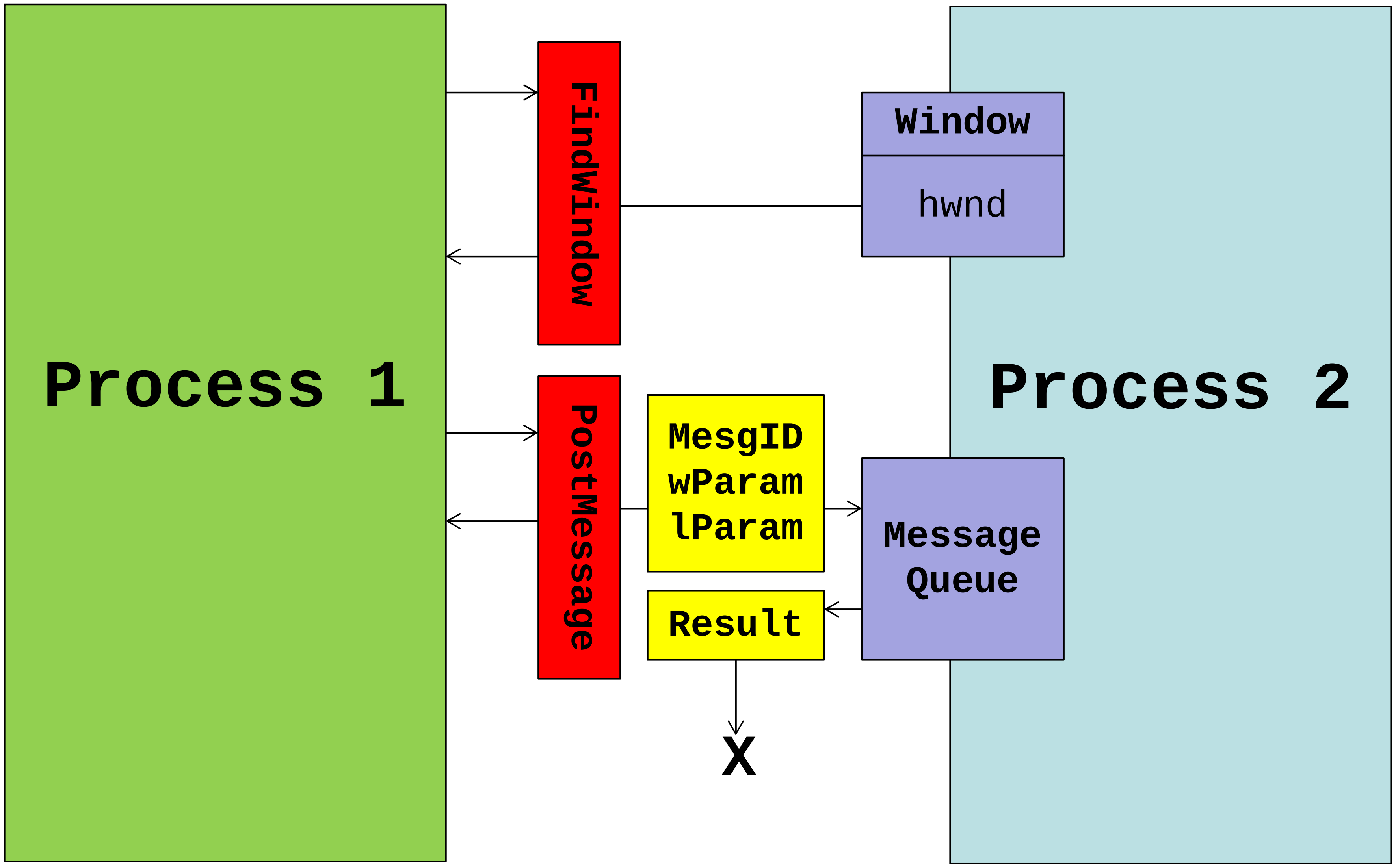
Windows Messages for communication

- SendMessage for synchronous communication
- Sends a message to a window and will wait for the thread which owns the window to handle the message
  - And to send a response back
- SendMessage blocks the thread which calls it until the message is handled
- If the caller thread owns the window, it will call the window procedure directly

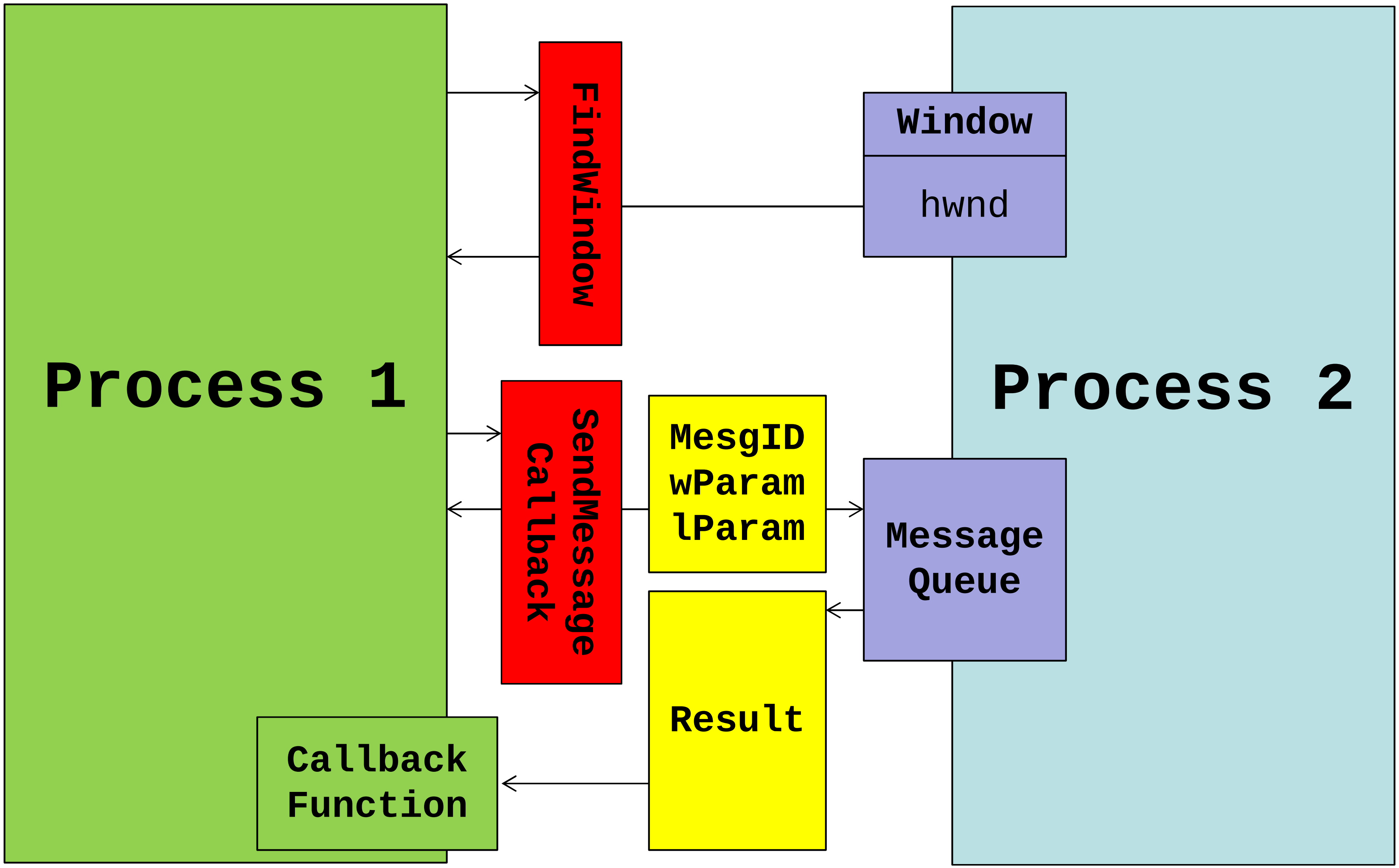
SendMessage



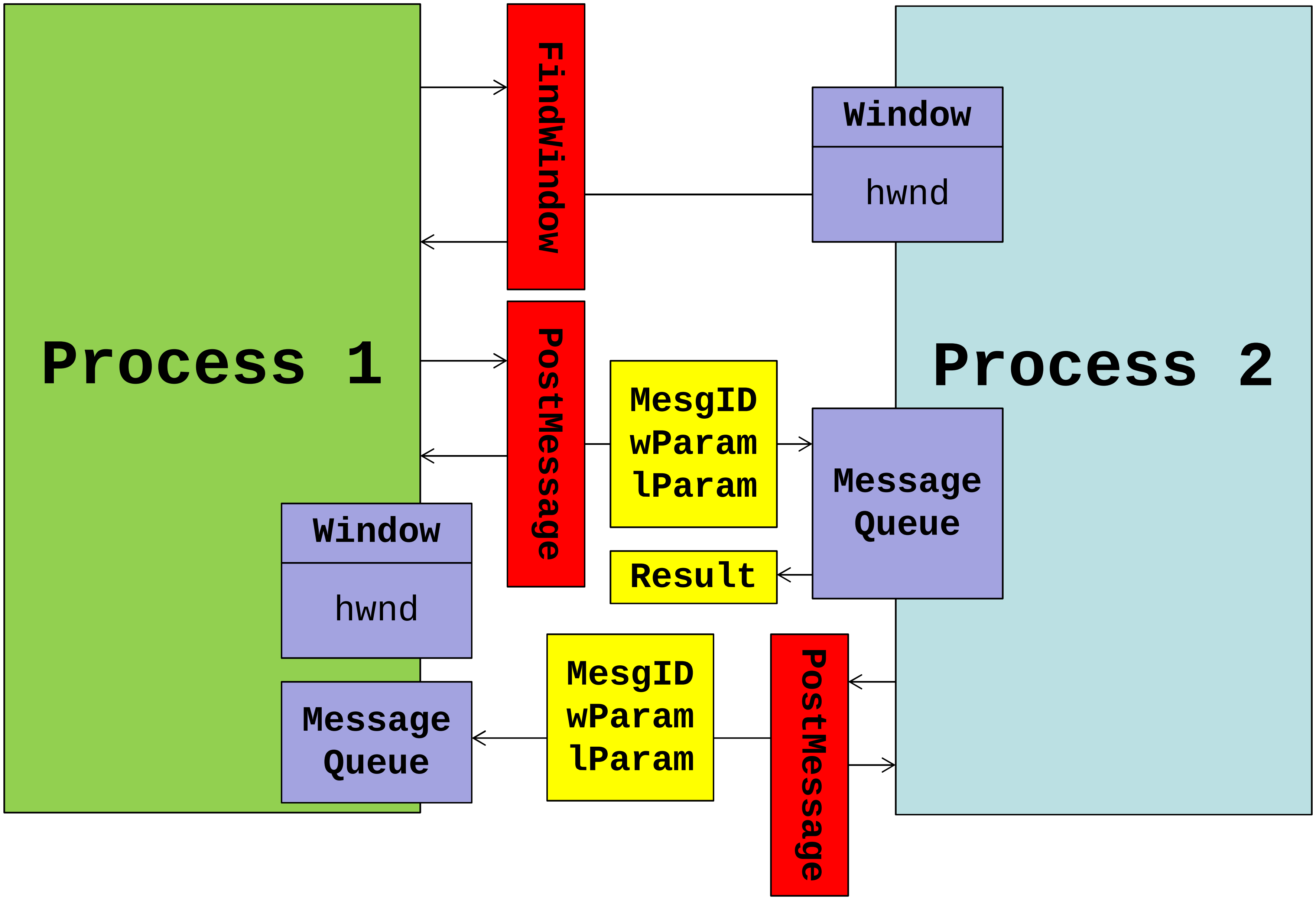
PostMessage



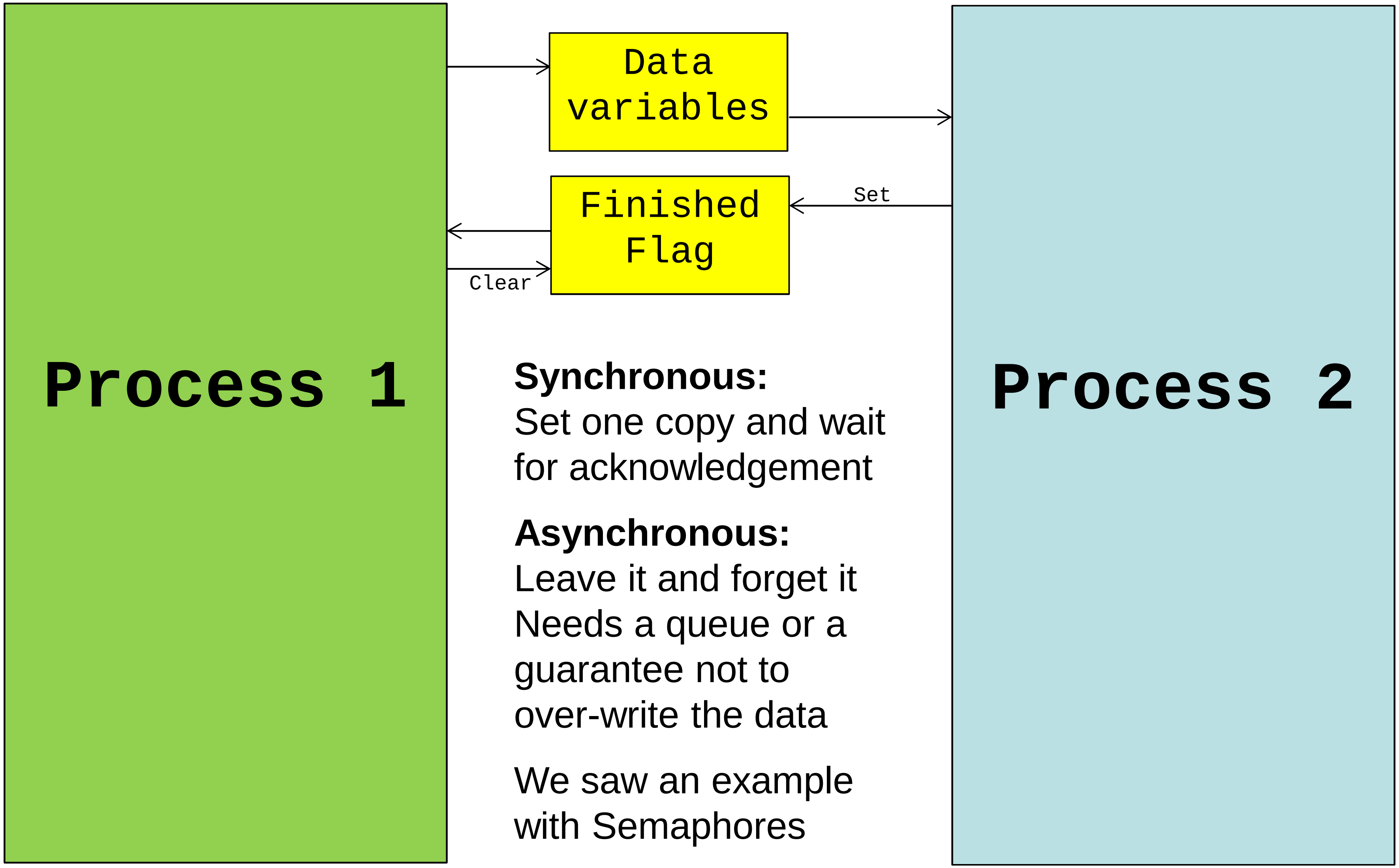
SendMessageCallback



Two-way PostMessage



Using shared memory to communicate



Synchronous:
Set one copy and wait for acknowledgement

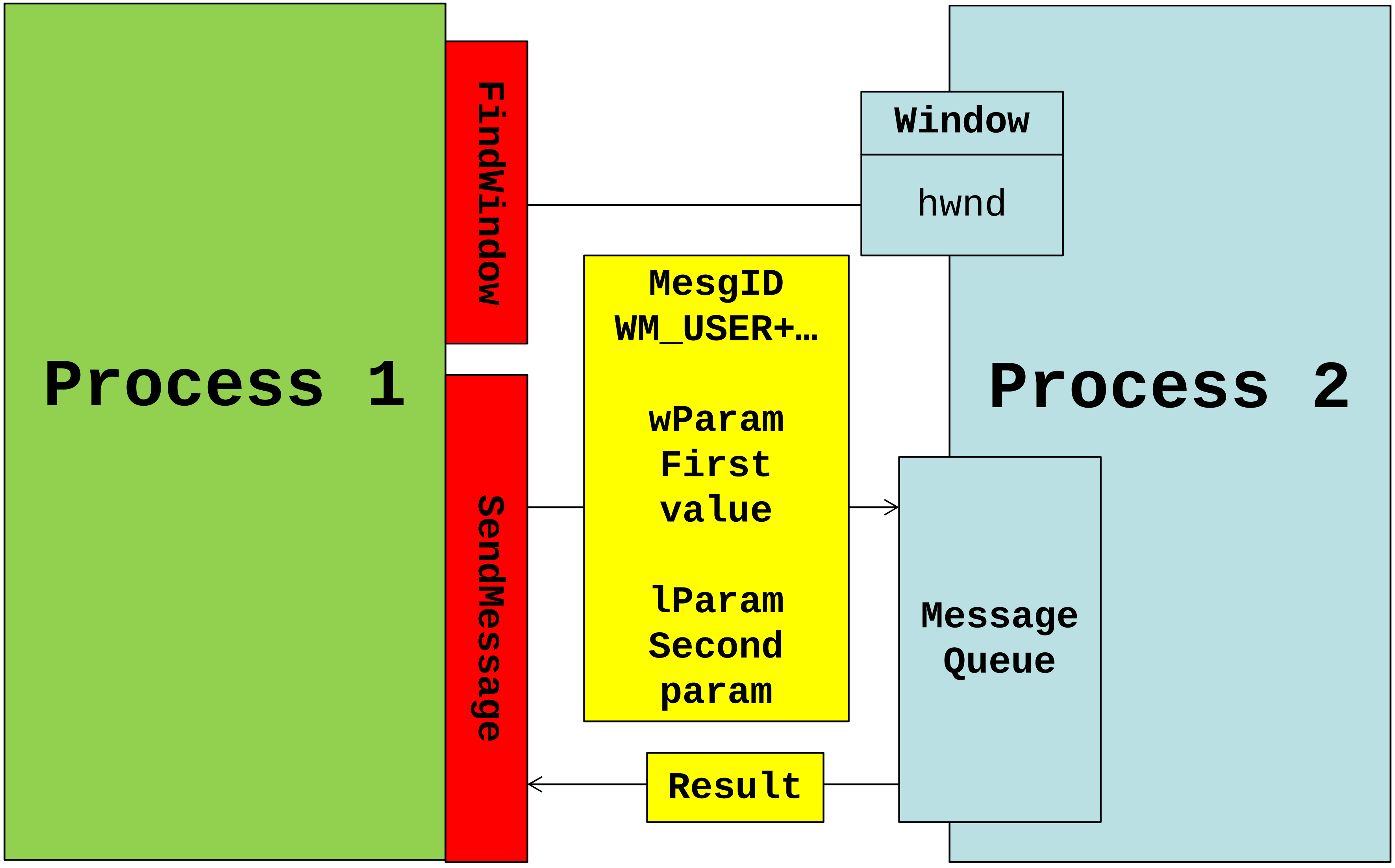
Asynchronous:
Leave it and forget it
Needs a queue or a guarantee not to over-write the data

We saw an example with Semaphores

Options for communications

- Synchronous
  - Easiest if you need a response, but caller has to wait (doing nothing/blocking)
- Asynchronous
  - Easy if you do **not** need a response
  - Need a sure-delivery method to be sure it is received
  - 'Message' must exist until the receiver handles it
  - Similarities to producer-consumer
    - Producer creates things which the consumer later uses
    - Things have to stay on the queue until picked up
  - Windows message queues meet this criterion
    - Limitations on payload though
  - MessageQueues exist on other OSes as well

Example



Process 1
FindWindow
SendMessage
Window
hwnd
MesgID
WM_USER+…
wParam
First
value
lParam
Second
param
Result
Process 2
Message
Queue

Example : Caller

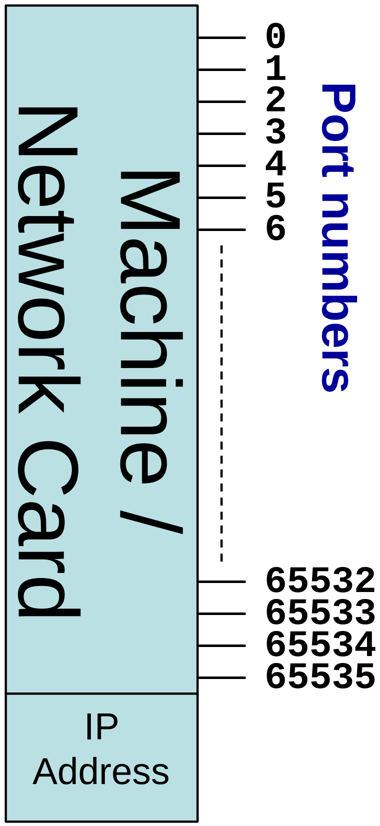
```
// Name matches what I used for a class name
HWND hwnd = FindWindow( "MyClassName",NULL );

while ( TRUE )
{
  printf( "Type two numbers with a space between:" );
  if ( scanf( "%d %d",&i,&j ) == 2 )
  {
    total = SendMessage( hwnd, WM_USER + 1, i, j );
    printf( "%d + %d = %d\n",i,j,total );
    PostMessage( hwnd,WM_USER + 2,total,0 );
  }
…
}
```

Example: Server/receiver

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg,
     WPARAM wParam, LPARAM lParam )
{
   switch ( msg )
   {
   …
   case WM_USER+1: // Do a plus
      printf( "%d plus %d = %d\n",wParam,lParam,wParam+lParam);
      …
      return wParam + lParam;
   case WM_USER + 2: // Set the caption
   {
      char buf[20]; sprintf( buf,"Calculated: %d",wParam );
      SetWindowText( hwnd,buf );
      return 0; // Not looking for this anyway
   }
```

Windows Sockets Introduction

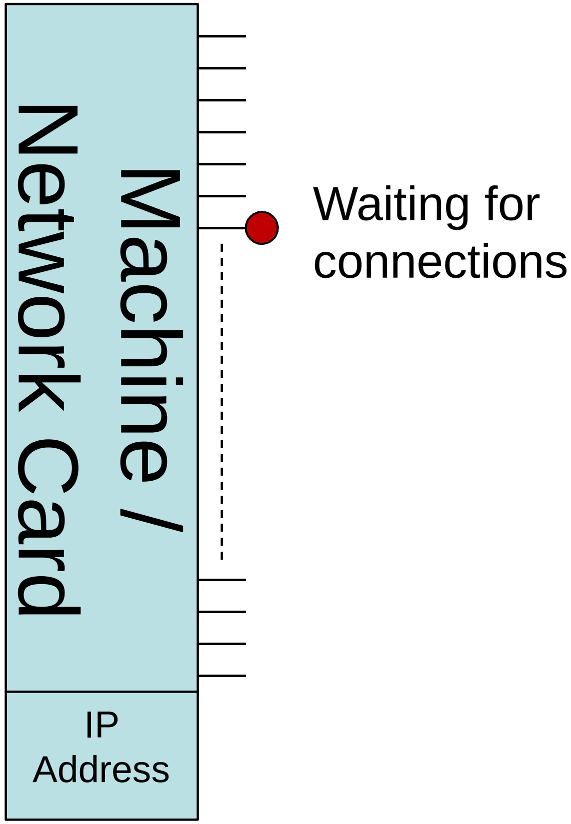
Sockets (Windows Sockets API)



- Stream-based
  - Even if underlying datagram-based
  - Sure-delivery – resends failed packets
- Numbered sockets (2 byte value)
- Server-side:
  - Listen for connections on a socket
  - Accept connections
    - Handle the connection (send/receive data)
  - Listen for next connection
- Client-side
  - Connect to socket on server (machine IP/socket #)
  - Send/receive data

Windows Sockets : server-side

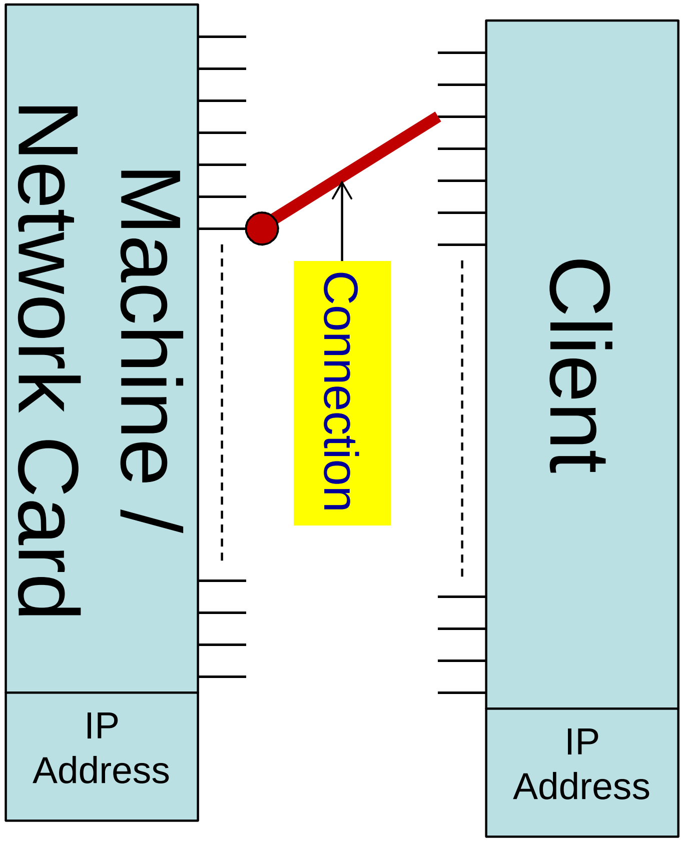
- **`socket()`**: allocates a socket handle to listen on
- **`bind()`**: grab a socket, associating a local address with it
- **`listen()`**: listen for incoming connections on the socket
- **`accept()`**: will block until a connection has occurred (or error)
  - **Start a new thread to handle connection?**
- **`send(), recv()`**: send and receive data using the open connection
- **`closesocket()`**: close the socket, release resources



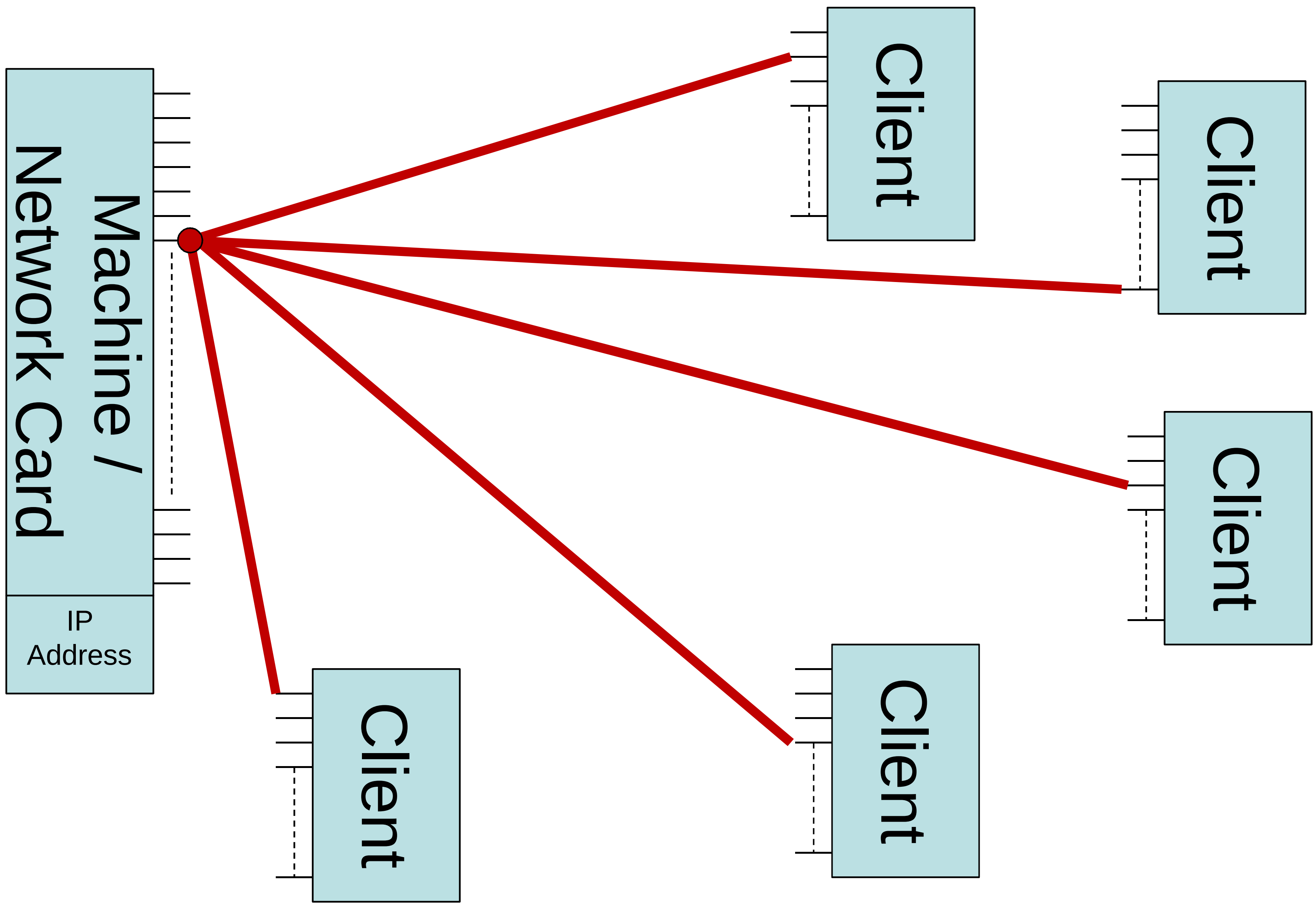
See the sample for an example of use, and for an overview, see:
https://msdn.microsoft.com/en-us/library/windows/desktop/ms740673%28v=vs.85%29.aspx
https://msdn.microsoft.com/en-us/library/windows/desktop/ms741394%28v=vs.85%29.aspx

Windows Sockets : Client

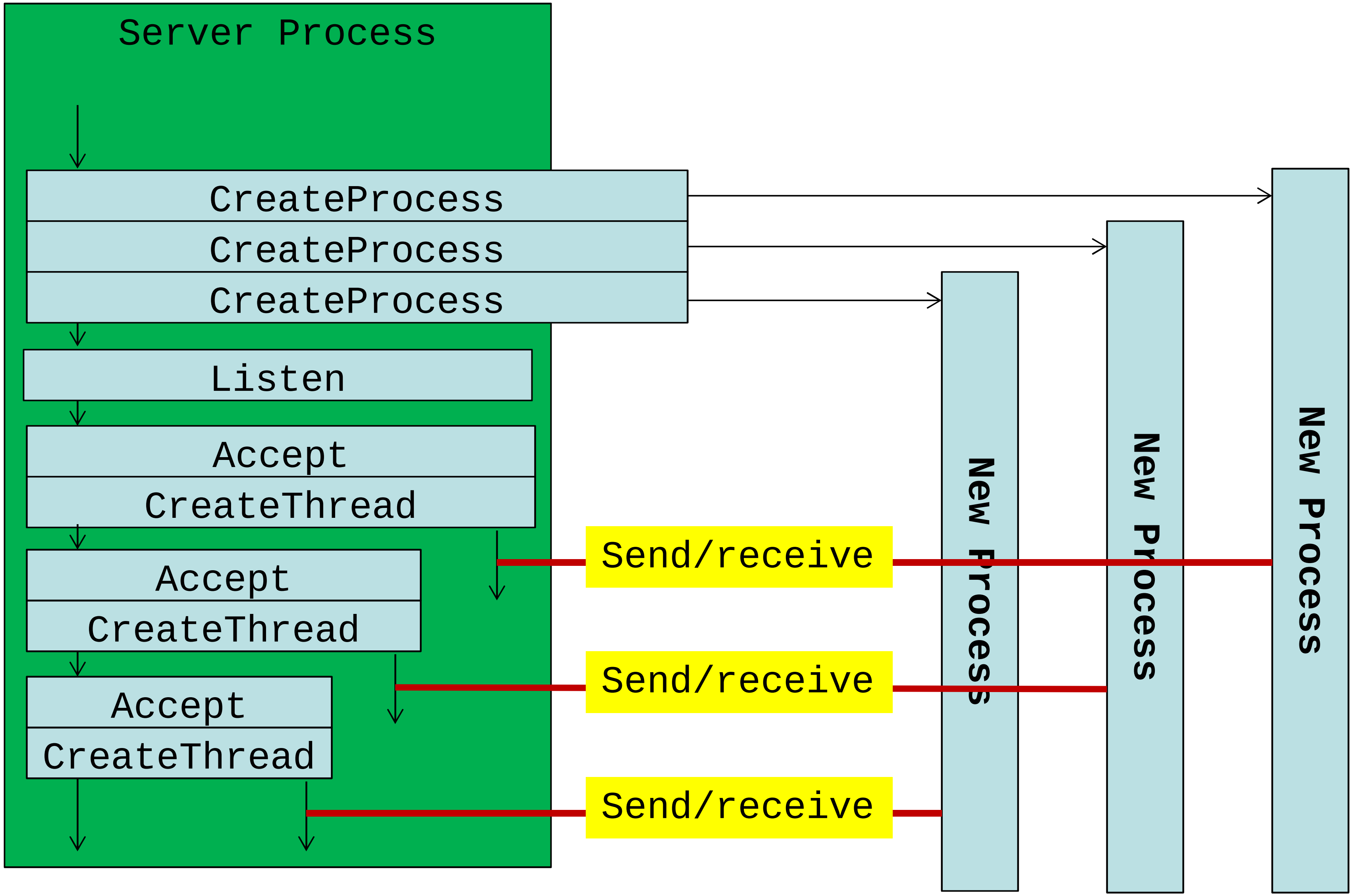
- **`socket()`**: Allocates a socket handle on local machine (protocol and address specified)
- **`connect()`**: Attempt to connect to server, supplying server address and port.
  Note: this will 'bind' an unbound local socket
- **`send(), recv()`**: send and receive data
- **`shutdown()`**: shutdown send or receive operations. Can shutdown send to let the server know that you have finished
- **`closesocket()`**: close the socket, release resources



Multiple connections possible



Example program



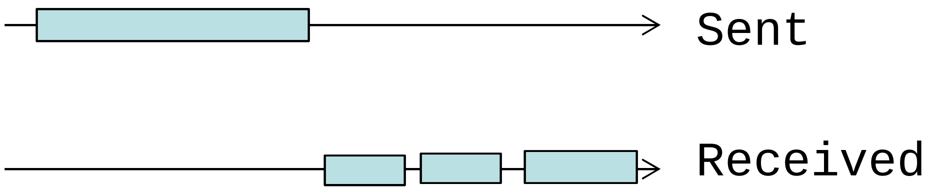
Sample code

1. Server opens socket and listens for connections
2. Client connects to server
3. Client sends and receives data
   – Client sends data
   – Server receives data from client
   – Server sends back whatever bytes it received
   – Client displays the messages
4. Eventually client has finished sending all data
   – Client calls shutdown to shutdown output
5. Server detects socket shutdown when it tries to read and closes the connection
6. Server thread dies

Structured messages

- Streams of bytes are sent down the connection
- You can send structured data if you wish
- E.g.

```
struct DataClientToServer
{
  char op;
  int first;
  int second;
};
```



- Even if you send the whole struct in one go, it may be split up when you receive it
  – E.g. you may receive the first two bytes, then 5 more, then 2 more, etc.
- You may need to reconstruct your data before you interpret it
- Example code in lab 4 exercise 4 does this for you (copy-paste-adapt)

Next lecture

- Concurrency in Java
- Monitors