Part III
Synchronization
Message Passing

The Concurrent Pascal compiler generated code for a simple machine tailored to the language.
I borrowed this idea from a portable Pascal compiler distributed by Wirth's group.
My Main concern was to simplify code generation.
The portability of Concurrent Pascal was just a useful by-product of this decision.
Twenty years later, the Java language would resurrect the idea of "platform-independent" concurrent programs.
Unfortunately, Java replaced the secure monitor concept of Concurrent Pascal with insecure shortcuts.

Per Brinch Hansen

Message Passing: 1/3

- When processes/threads are interested with one another, two fundamental requirements must be met: synchronization and communication.
- **Synchronization** enforces mutual exclusion.
- **Communication** allows information to be passed to other processes/threads.
- Message passing, a form of communication, can be implemented in shared-memory and distributed environment.

Message Passing: 2/3

- **Mutex locks, semaphores and monitors are shared-memory synchronization mechanisms.**
- **This means all processes and threads use a piece of shared memory to store and manage mutex locks, semaphores and monitors.**
- **In a distributed environment, processes and threads run on different computers without a global shared-memory.**
- **In this case, message passing becomes useful.**

Message Passing: 3/3

- **Communication links can be established between threads/processes. There are three important issues:**
  1. **Naming: How to refer to each other?**
  2. **Synchronization: Shall we wait when participating a message activity?**
  3. **Buffering: Can messages wait in a communication link?**

Naming: Direct Addressing Symmetric Scheme: 1/3

- **Direct Addressing: Each process that wants to communicate must explicitly name the other party:**
  - `Send(receiver, message);`
  - `Receive(sender, message);`
- **With this scheme:**
  - **Exactly one link exists between each pair of communicating processes.**
  - **These links may be established for processes that need to communicate before they run.**

Naming: Direct Addressing Asymmetric Scheme: 2/3

- **In this scheme, we have**
  - `Send(receiver, message);`
  - `Receive(id, message);`
- **The `Receive()` primitive receives the ID of the sender. Thus, in this scheme, a receiver can receive messages from any process.**

Naming: Direct Addressing Disadvantages: 3/3

- **There are disadvantages in the symmetric and asymmetric schemes:**
  - **Changing the name/ID of a process may require examining all other process definitions.**
  - **Processes must know the IDs of the other parties to start a communication.**

Naming: Indirect Addressing Mailbox: 1/4

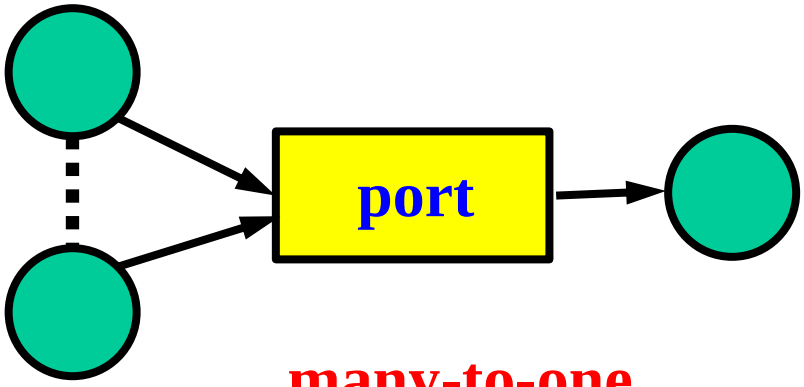
- With indirect addressing, messages are sent to and received from **mailboxes**.
- Each mailbox has a unique ID.
- The primitives are
  - `Send(mailbox-name, message);`
  - `Receive(mailbox-name,message);`

Naming: Indirect Addressing Communication Links: 2/4

- There is a link between two processes only if they share a mailbox.
- A link may be shared by multiple processes.
- Multiple links may exist between each pair of processes, and each link corresponds to a mailbox.
- By decoupling the sender and receiver, indirect addressing provides a greater flexibility in the use of messages.

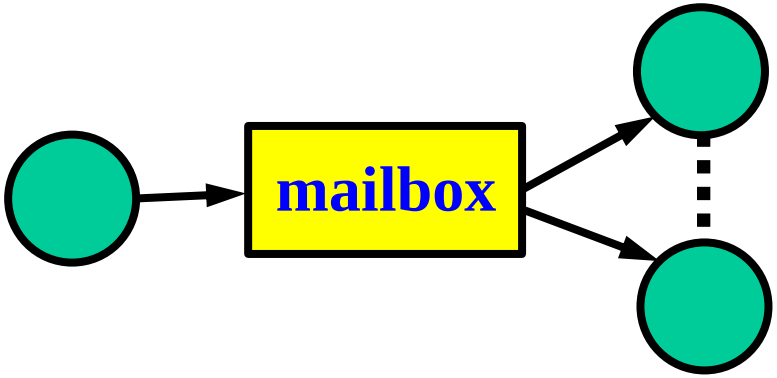
Naming: Indirect Addressing
Communication Links: 3/4

one-to-one

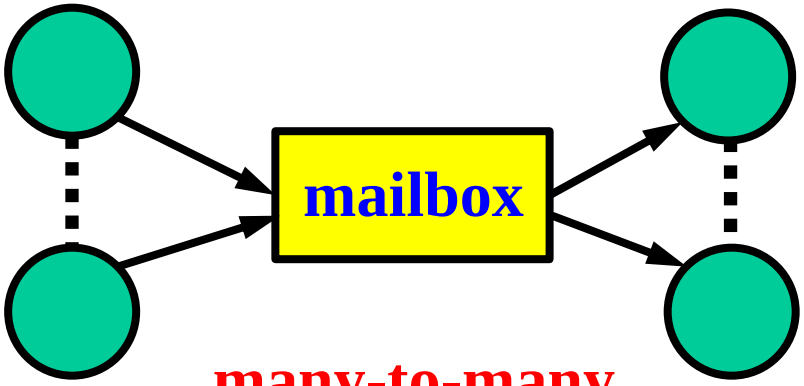


many-to-one

mailbox → port



one-to-many



many-to-many

Naming: Indirect Addressing Communication Links: 4/4

- **What if there is only one message in a mailbox and several processes execute `Receive()`? It depends on the following:**
  - **If there is only one link between at most two processes, this situation will not happen.**
  - **Allow at most one process to receive at a time.**
  - **Allow the system to select an arbitrary order.**

Synchronization

- **The sender and/or receiver may be blocked:**
  - ❖ **Blocking Send: the sender blocks until its message is received**
  - ❖ **Nonblocking Send: the sender sends and resumes its execution immediately**
  - ❖ **Blocking Receive: the receiver blocks until a message is available**
  - ❖ **Nonblocking Receive: the receive receives a message or a null.**
- **When both send and receive are blocking, we have a rendezvous between the sender and receiver.**

Synchronous vs. Asynchronous

- **Blocking** and **non-blocking** are known as synchronous and asynchronous, respectively.
  - If the sender and receiver must synchronize their activities, use synchronous communication.
  - Because of the uncertainty in the order of events, asynchronous communication is more difficult to program.
  - On the other hand, asynchronous algorithms are general and portable, because they are guaranteed to run correctly on networks with arbitrary timing behavior.

Capacity

- **The capacity of a link is its buffer size:**
  - **Zero Capacity: Since no message can be waiting in the link, it is synchronous. Sender blocks.**
  - **Unbounded Capacity: Messages can wait in the link. Sender never blocks and the link is asynchronous. The order of messages being received does not have to be FIFO.**
  - **Bounded Capacity: Buffered Message Passing. Sender blocks if the buffer is full, and the link is asynchronous. Isn't it a bounded buffer if the order is FIFO?**

Message Passing in Unix

- **Unix systems provide at least two message passing mechanisms: pipes and message queues.**
- **A pipe is a generalization of the pipe in `A | B`.**
- **Function `pipe(int pfd[2])` creates a communication link and returns two file descriptors. Writing to `pfd[1]` puts data into the pipe; reading from `pfd[0]` gets the data out. Data items in a pipe are FIFO just like `A | B`.**
- **Message queues are mailboxes. Use `msgget()` to obtain a message queue, and use `msgsnd()` and `msgrcv()` to send and receive messages.**

Message Passing with ThreadMentor

Channels

- A **channel** is a *bi-directional* communication link between two specific threads.
- A channel can be synchronous or asynchronous.
- A thread can be assigned a user-defined thread ID. A user-defined thread ID is a unique non-negative integer selected by the user.
- Channels use user-defined threads IDs for identification purpose
- The user-defined thread ID must be set in the thread constructor:

```
UserDefinedThreadID = 10;
```

Declaring a Channel

- **Two classes `SynOneToOneChannel` (blocking send and receive) and `AsynOneToOneChannel` (non-blocking send and receive) are available:**

```
SynOneToOneChannel  X("chan-2-3", 15, 3);
AsynOneToOneChannel Y("chan-4-5", 15, 3);
```

channel names

user-defined thread IDs
channels `X` and `Y` are built between threads 15 and 3

Sending and Receiving: 1/2

- **Use methods `Send()` and `Receive()` to send and receive a message to and from a channel:**

```
X.Send(*pointer-to-message,size);
Y.Receive(*pointer-to-message,size);
```

- **Send an `int` message to channel `X`:**

```
AsynOneToOneChannel  X;
int  Msg;
X.Send(&Msg, sizeof(int));
```

Sending and Receiving: 2/2

- **Receive a message of four doubles from channel `Y`:**

```
AsynOneToOneChannel  Y;
double  Msg[4];
Y.Receive(Msg, 4*sizeof(double));
```

- **Send a message of `struct Data` to channel `Z`;**

```
struct Data
   {double x; int y; char z[100];}
      Msg;
SynOneToOneChannel  Z;
Z.Send(&Msg, sizeof(Data));
```

Linear Array Sorting: 1/10

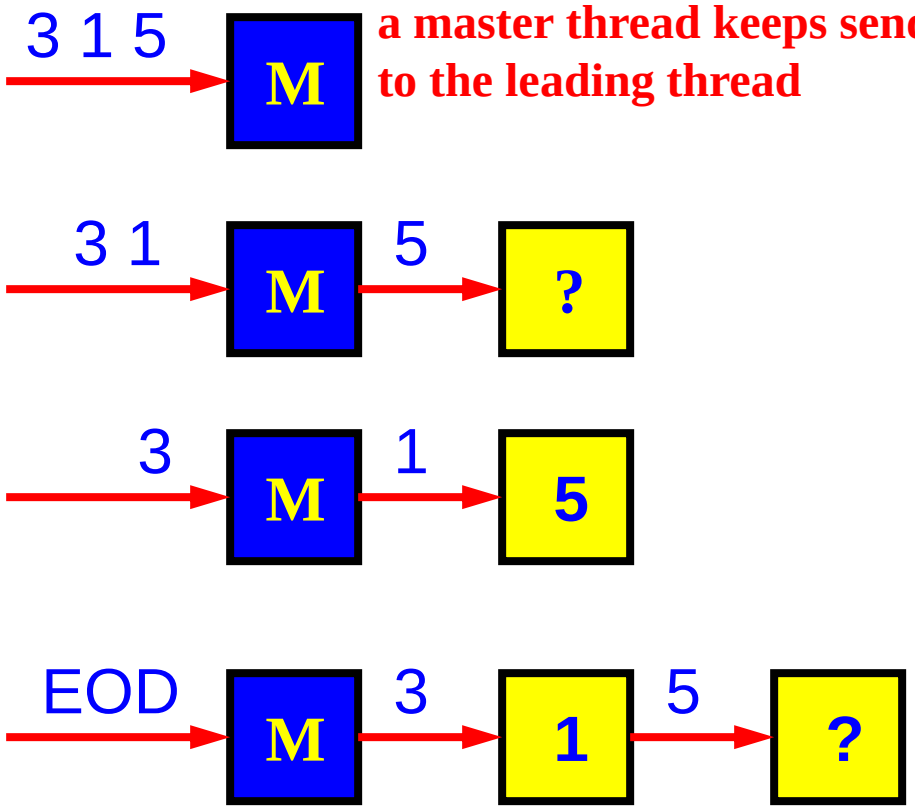
- **Each thread has an input channel from its predecessor and an output channel to its successor.**
- **The first time a thread receives a positive integer, it is memorized as *N*.**
- **For a subsequent received positive integer *X*:**
  - **If $X < N$, this thread sends *N* to its successor and memorizes *X* as *N*.**
  - **Otherwise, *X* is sent to its successor.**
  - **If there is no successor, this thread creates a thread, builds a channel to it, and sends the number.**

Linear Array Sorting: 2/10

- **Why does this method sort the input?**
- **For each thread, the number it memorizes is less than all numbers memorized by its successors.**
- **Therefore, the first thread has the smallest number, the second thread has the second smallest number, etc. and the last thread has the largest number.**

Linear Array Sorting: 3/10

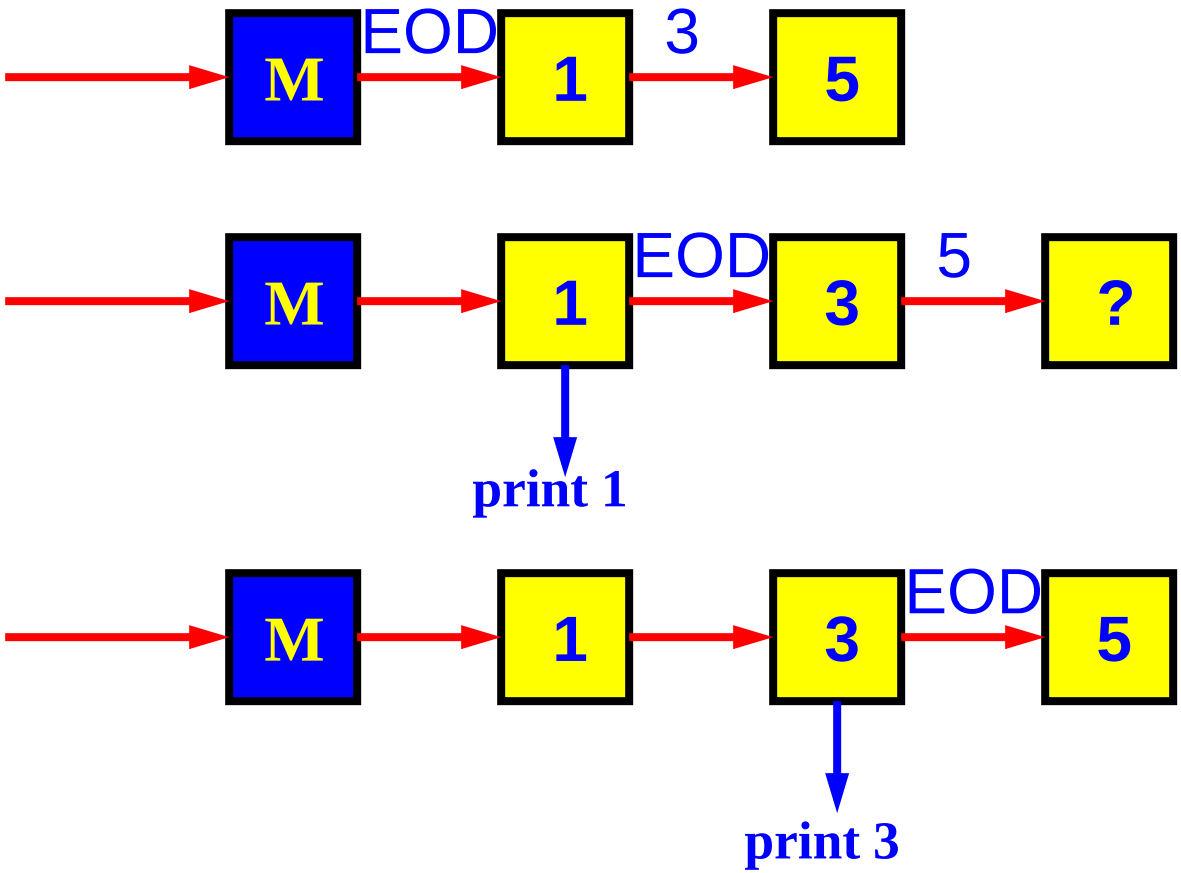
input: 5 1 3



a master thread keeps sending numbers to the leading thread

sort positive integers with `-1` as `end-of-data`

Linear Array Sorting: 4/10



What type of channels (i.e., sync or async) should be used?

Linear Array Sorting: 5/10

```
const int NOT_DEFINED = -2;
const int END_OF_DATA = -1; // end of input flag
class SortThread : public Thread
{
   public:
      SortThread(int index, int threadID);
      ~SortThread(); // destructor
      SynOneToOneChannel *channel;
   private:
      void ThreadFunc();
      int Index;     // index of the sort thread
      int Number;    // number memorized
      SortThread *neighbor; // next sort thread
};
class MasterThread : public Thread
{
   public: MasterThread(int threadID);
   private: void ThreadFunc();
};
```

used to construct thread name

class definition

Linear Array Sorting: 6/10

can an async. channel be used here?

```
SortThread::SortThread(int index, int threadID)
{
    Index = index;
    UserDefinedThreadID = threadID;
    neighbor = NULL; // initially no neighbor
    Number = NOT_DEFINED; // no memorized number
    ChannelName = … // give this channel a name
    channel = new SynOneToOneChannel(ChannelName,
                        threadID-1, threadID);
}

SortThread::~SortThread() // this is a destructor
{ delete channel; }
```

a sync channel between them

constructor and destructor

SortThread body 1/2

```
void SortThread::ThreadFunc()
{
   Thread::ThreadFunc();
   int number, tmpNum;
   Thread_t self = GetID();
   while(true) {
      channel->Receive(&number, sizeof(int)); // receive a number
      if (number == END_OF_DATA)
         break;
      if (Number == NOT_DEFINED)
         Number = number;                  // first number.  Memorize it
      else {                               // other numbers
         if (number >= Number)             // larger than mine
            tmpNum = number;               //    save it in temporarily
         else {
            tmpNum = Number;               // no. save mine in temporarily
            Number = number;               //    but, also memorize it
         }
         if (neighbor == NULL)              // no neighbor? create one
            neighbor = new SortThread(Index+1,UserDefinedThreadID+1);
            neighbor->Begin();              // run it!
         }
         neighbor->channel->Send(&tmpNum, sizeof(int)); // send number
      }
   }  // end of data reached.  see next slide
```

GetID() returns the ID of a thread

Linear Array Sorting: 8/10

SortThread body 2/2

```
void SortThread::ThreadFunc()
{
   while (true) {
     // other stuffs on the previous slide
     // end of data received
   }
   if (neighbor != NULL) { // if I am not the last one
                           // I should pass the EOD
      neighbor->channel->Send(&number, sizeof(int));
      neighbor->Join();    // wait for neighbor to complete
   }
   Exit();
}
```

MasterThread constructor and body

```
MasterThread::MasterThread(int threadID)
{
    UserDefinedThreadID = threadID;
    ThreadName = …; // a thread name
}

void MasterThread::ThreadFunc()
{
    Thread::ThreadFunc();
    int input;
    do {
        cin >> input;            // read an integer or END_OF_DATA
        if (input == END_OF_DATA)
            break;
        else
            firstSortThread->channel->Send(&input,sizeof(int));
    } while (input != END_OF_DATA);
    firstSortThread->channel->Send(&input, sizeof(int));
    Exit();
}
```

send END_OF_DATA to the first thread

Linear Array Sorting: 10/10

```
SortThread *firstSortThread; // first sorting thread
void main(void)
{
   MasterThread *masterThread;
   firstSortThread = new SortThread(1,2);
   firstSortThread->Begin();
   masterThread = new MasterThread(1);
   masterThread->Begin();
   masterThread->Join();
   firstSortThread->Join();
   Exit();
}
```

sorting thread #1
threadID 2

threadID 1

main program

The End