

CSCI3180 Principles of Programming Languages



The Administrator and Worker Paradigm and SIMPL Library

Tutorial 5



Introduction

- Concurrent programming
 - Several sequential processes are executing in parallel, and cooperating with other processes.
 - They must communicate and synchronize.
 - Low-level concurrent constructs include: parbegin-parend, semaphores, mutexes, message passing, and remote procedure calls.



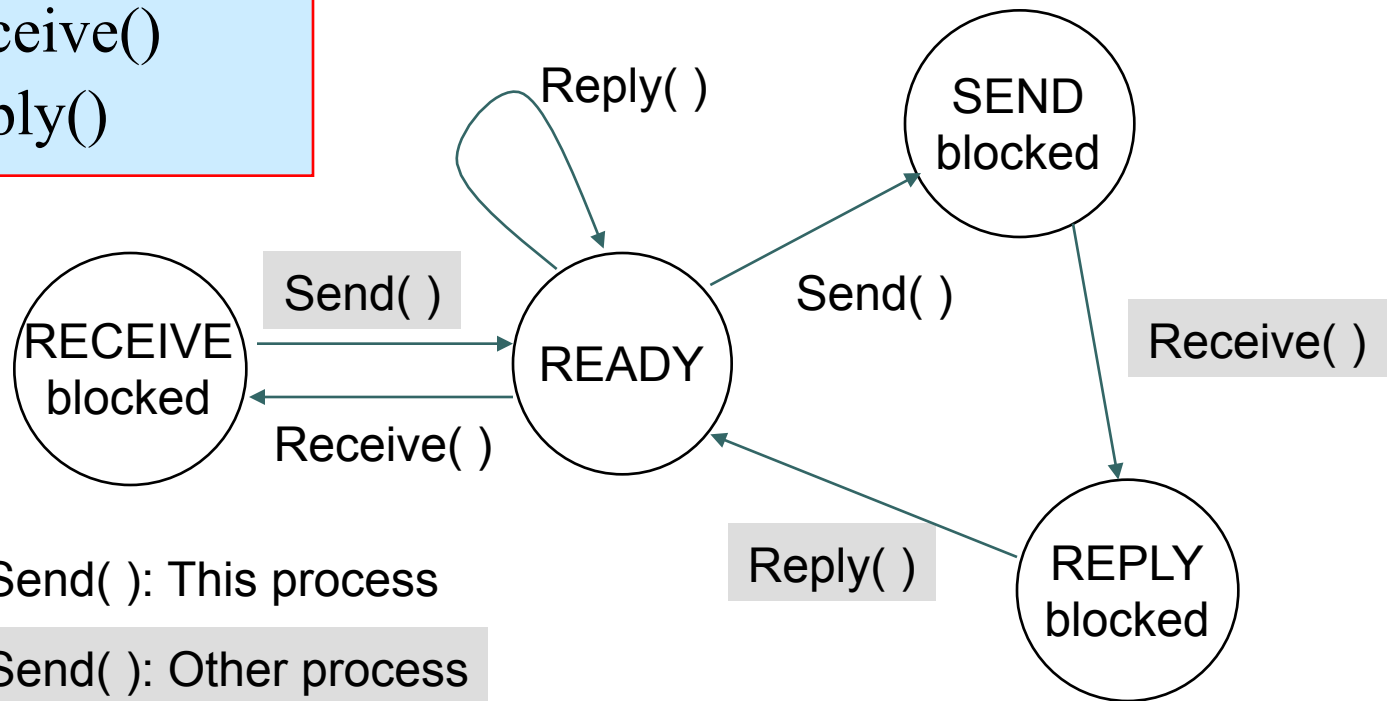
The Message Passing System

- With carefully designed semantics, message passing can be used for process **synchronization**.
- The message passing system consists of a blocking Send(), a blocking Receive(), and a non-blocking Reply().

Send/Receive/Reply

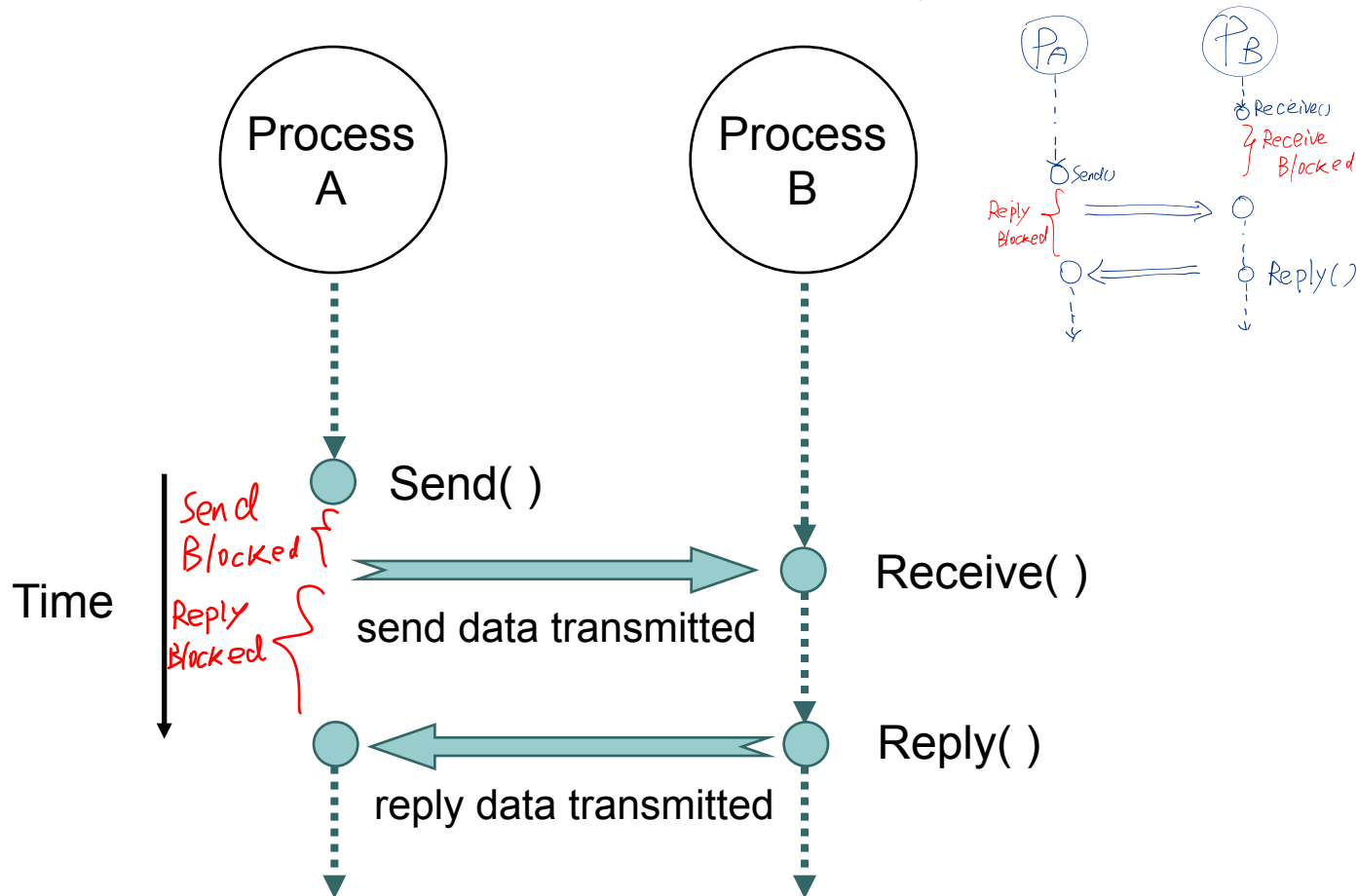
○ Messaging:

- Send()
- Receive()
- Reply()



A Simple Sequence of Events

△△ Final!!!画Process B提前Receive的图





More about Message Passing

- Note how message passing not only allows processes to pass data to each other, but also provides a means of **synchronizing** the execution of several cooperating processes.



Administrator and Worker Paradigm

- The message passing model just described supports a new paradigm for concurrent programming: the Administrator-and-Worker paradigm.
- Two types of processes:
administrators and **workers**.



Administrator

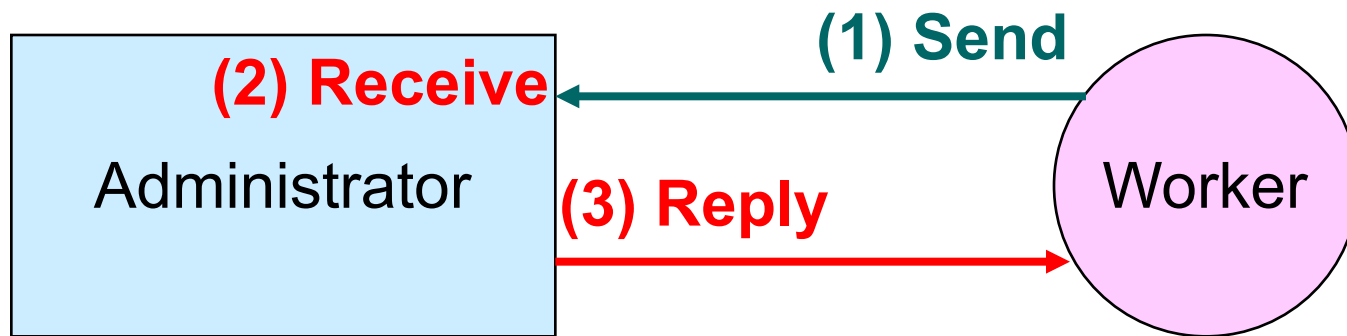
- An administrator owns one or more worker processes.
- Usually, an administrator is used to maintain a critical resource
 - e.g. memory, display, keyboard, etc.
- It is mainly responsible for
 - Receiving requests from clients
 - Queuing up the requests
 - Delegating jobs to its worker processes.



Worker

- A **worker** performs the actual computation that its administrator assigned.
- It is **dedicated for a special purpose**
 - e.g. painting the screen
- It has to report availability to its administrator.

Administrator and Worker Cooperation



Administrator's action

Worker's action

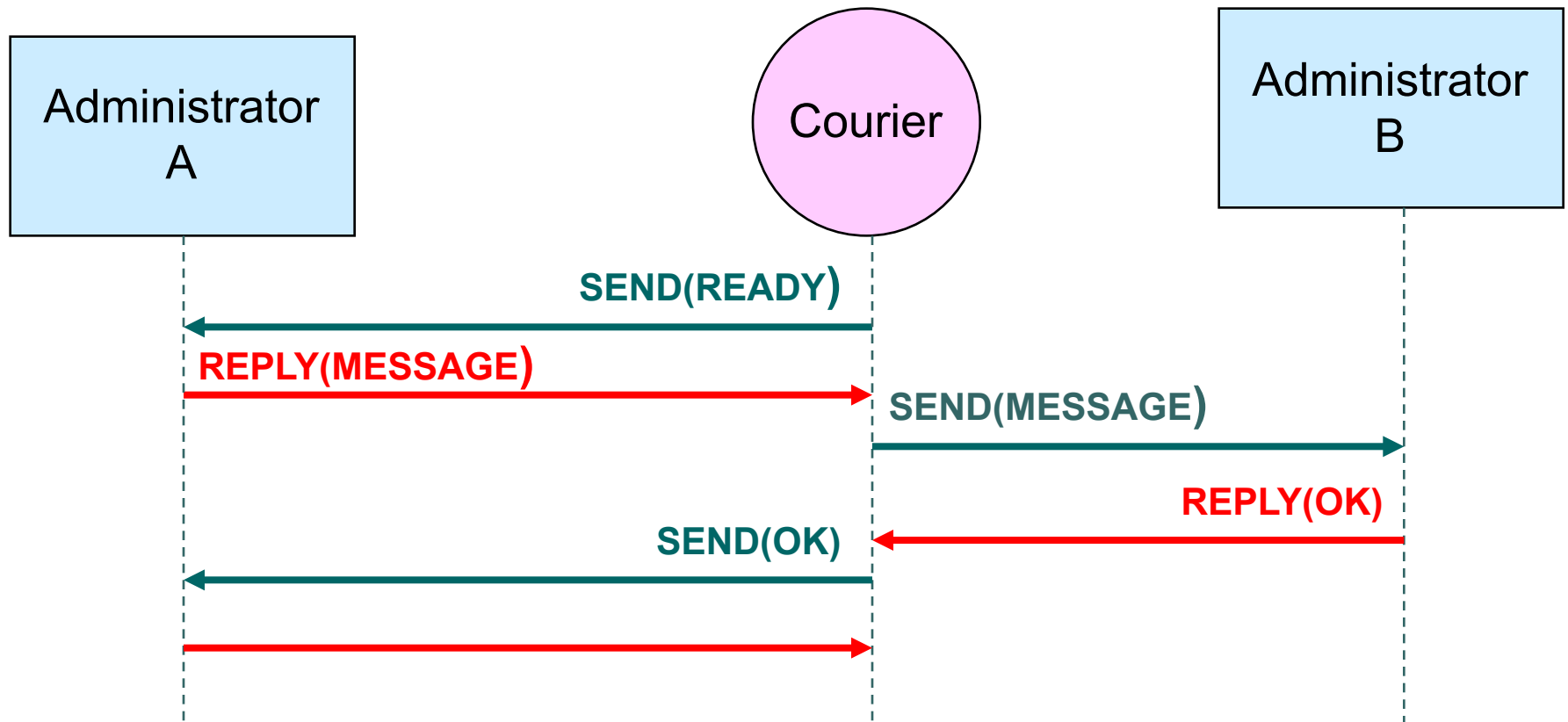


Courier

- In a program, there can be *more than one* administrators.
- Two administrators cannot communicate each other directly!
- A courier is a special type of worker processes that sends messages on behalf of its administrator to another administrator.

Cooperation Between Two Administrators

- Relay messages from one Administrator to another





Implementation

- C Language
- GNU/LINUX with GCC compiler
- Concurrent Programming
 - Synchronous Interprocess Messaging Project for LINUX (**SIMPL**)
- **SIMPL provides Send/Receive/Reply messaging**
 - first popularized in commercial Real-Time Operating System (RTOS), such as QNX



SIMPL Setup

- LINUX Only!
 - We have [linux6](#) to [linux9](#)
 - Connection method is the same as for sparc machines
- Create directory [fifo/](#)
 - e.g. in your home directory
- Decompress SIMPL library to [simpl/](#)
 - can be downloaded from course web page



SIMPL Setup

- Set paths in your shell environment
 - e.g. in your `.cshrc` file
 - `setenv FIFO_PATH $HOME/fifo`
 - `setenv SIMPL_HOME $HOME/simpl`
- Include header file "`simpl.h`" in your source files
- Use the `makefile` provided to compile

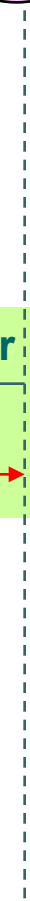
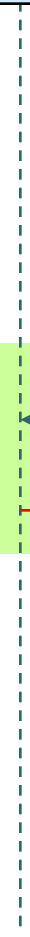
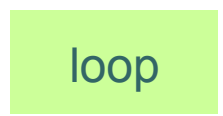
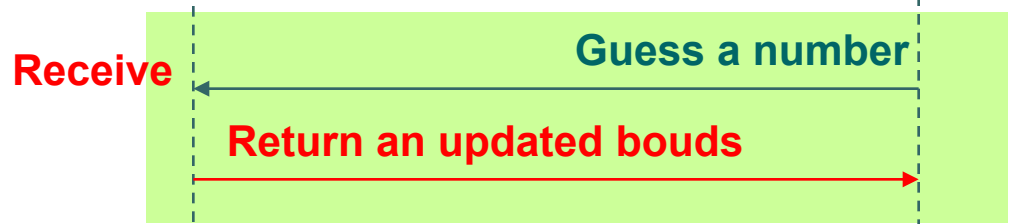
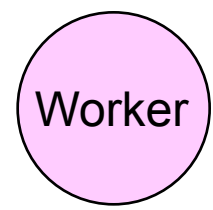
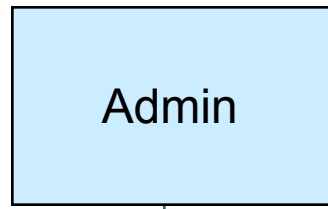


Sample Code: Number Guessing

- A *very simple* example
- Administrator randomly picks a number from 0 to 99
- Every worker continuously guessing the number by randomly picking a number from the bounds
 - the bounds are updated for each guess



Admin and Worker





Attaching Process

```
int name_attach(char* name,  
                void(*exitFunc)())
```

- Register a process's name in the namespace
- Required by every process that expects to send/receive messages from any other process
- It returns 0 on success or -1 on failure



Locating a Process

```
int name_locate(char* name)
```

- Identify the receiver before sending a message
- Returns
 - Success: the unique ID of the receiver
 - Failure: -1

NN大佬



Detaching process

```
int name_detach(void)
```

- Remove a process's (previously registered) name from the namespace:
- It returns 0 on success or -1 on failure

Blocking Message Send

File Descriptor of the receiver

```
int Send(int fd, void *out, void *in,  
          unsigned outSize, unsigned inSize)
```

- Send a message to **fd** pointed to by **out** with size **outSize**.
- The **fd** is the receiver ID obtained from **name_locate()**
- It expects a reply message from **fd** to be placed in memory pointed to by **in** with size no larger than **inSize**.
- Returns
 - Success: n, size of the reply message
 - Failure: -1



Blocking Message Receive

```
int Receive(char **ptr, void *inArea,  
            unsigned maxBytes)
```

- Receive a message in a memory area pointed to by `inArea` and no larger than `maxBytes`.
- The `*ptr` is a **record** to uniquely identifies the sender and is used for `Reply()`
- The record `*ptr` is destroyed after `Reply()`
- Returns
 - Success: `n`, size of the received message
 - Failure: `-1`



Non-blocking Message Reply

```
int Reply(char *ptr, void *outArea,  
          unsigned outSize)
```

- Reply a message to a blocked sender (designated by `ptr`) pointed to by `outArea` with size `outSize`.
- It returns 0 on success or -1 on failure

Administrator Template

Predefined message structure

Blocked when waiting for incoming message

Remember to detach name

```
int main(...) {
    char* fromWhom = NULL; MESSAGE msg, reply; // ...
    if (name_attach("Admin", NULL) == -1)
        die(ATTACH_ERR); // ...
    while (...) {
        // ...
        if (Receive(&fromWhom, &msg, sizeof(msg)) == -1)
            die(RECEIVE_ERR);
        // ...
        if (Reply(fromWhom, &reply, sizeof(reply)) == -1)
            die(REPLY_ERR);
        // ...
    } // ...
    if (name_detach() == -1) die(DETACH_ERR);
    return 0;
}
```


Worker Template

```
int main(...) {
    int fd; MESSAGE msg, reply; // ...
    if (name_attach("Worker", NULL) == -1)
        die(ATTACH_ERR); // ...
    if ((fd = name_locate("Admin")) == -1)
        die(LOCATE_ERR); // ...
    while (...) {
        // ...
        if (Send(fd, &msg, &reply, sizeof(msg),
                sizeof(reply)) == -1) die(SEND_ERR);
        // ...
    } // ...
    if (name_detach() == -1) die(DETACH_ERR);
    return 0;
}
```

Predefined message structure

Blocked when waiting for reply

Remember to detach name



Shortcomings of the Sample

- When one of the worker **wins**, the administrator **quits** immediately
- Other workers **fail** to send message
- Solutions
 - Fixed number of workers
 - ➔ Always **wait** for all
 - Variable number of workers
 - ➔ **Register** each worker and inform them when the game is over

Modify Message Structure

- Add additional field in MESSAGE to indicate the type

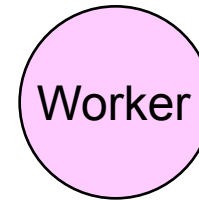
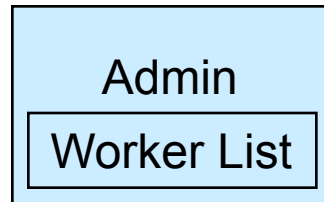
```
typedef struct {  
    int lowerBound;  
    int upperBound;  
    int guess;  
} MESSAGE;
```



```
typedef struct {  
    int type;  
    int lowerBound;  
    int upperBound;  
    int guess;  
} MESSAGE;
```

- e.g. 1 → Join, 2 → Guess

Register Workers



I want to join the game!

OK

Guess a number

Return an updated bounds

loop

Guess a number

The game is over!

Send()

Reply()

Leave as an exercise