Communication with Messages

ECE595
Feb 1

Y. Charlie Hu

Roadmap

- Interprocess communication *with shared data*
  - Synchronization with locks, semaphores, condition var
  - Classic sync. problem 1: producer-consumer
  - Semaphore implementations (uniprocessor, multiprocessor)
  - Classic sync. problems 2 & 3
  - Wait-free synchronization

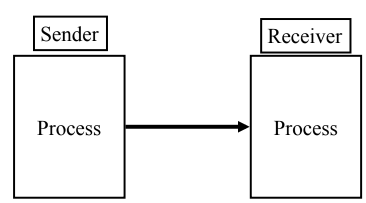
Today:

- Interprocess communication *with messages*
- Project 2

Inter-process Communication with Messages

- Messages provide for communication without shared data
  - One process or the other owns the data, (guaranteed) never two at the same time
  - Think about usmail

Big Picture



Why use messages?

- Many types of applications fit into the model of processing a sequential flow of information
- Communication across address spaces – no side effects
  - Less error-prone
  - They might have been written by diff programmers who aren't familiar with code
  - They might not trust each other
  - They may not be running on different machines!
  - Examples?

Message Passing API

- Generic API
  - `send( mailbox, msg )`
  - `recv( mailbox, msg )`
- What is a mailbox?
  - A *buffer* where messages are stored between the time they are sent and the time when they are received
- What should "msg" be?
  - Fixed size msgs
  - Variable sized msgs: need to specify sizes

Buffering leads to design options

- When should send() return?
- When should recv() return?

Send

- *Fully Synchronous*
  - Will not return until data is received by the receiving process
- *Synchronous*
  - Will not return until data is received by the mailbox
  - Block on mailbox full
- *Asynchronous*
  - Return immediately
  - Completion
    - Require the application to check status (appl polls)
    - Notify the application (OS sends interrupt)
  - Block on mailbox full

Receive

- *Synchronous*
  - Return data if there is a message
  - Block on empty buffer
- *Asynchronous*
  - Return data if there is a message
  - Return status if there is no message (probe)

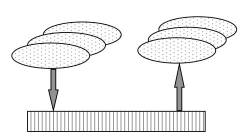
OS Implementation

- What is the conceptual problem for OS implementation here?
  - Assume sender and receiver are on the same machine

No Buffering

- Sender must wait until the receiver receives the message
- Rendezvous on each message

Mailbox - Bounded Buffer



- Buffer
  - Has fixed size
  - Is a FIFO
  - Variable size message
- Multiple producers
  - Put data into the buffer
- Multiple consumers
  - Remove data from the buffer
- Blocking operations
  - Sender waits if not enough space
  - Receiver waits if no message
- Synchronization
  - Using lock/condition variable (or semaphore)

Direct Communication

- Each process must name the sending or receiving process
- A communication link
  - is set up between the pair of processes
  - is associated with exactly two processes
  - exactly one link between each pair of processes

```
P: send( process Q, msg )
Q: recv( process P, msg )
```


Producer-Consumer Problem with Message Passing

```
Producer(){
  while (1) {
    …
    produce item
    …
    send( consumer, item);
  }
}
```

```
Consumer(){
  while (1) {
    recv( producer, item );
    …
    consume item
    …
  }
}
```


Indirect Communication

- Use a "mailbox" or "ports" to allow many-to-many communication
  - Mailbox typically owned by the OS
    - Requires open/close a mailbox before allowed to use it
- A "link"
  - is set up among processes only if they have a shared mailbox
  - Can be associated with more than two processes

```
P: open (mailbox); send( mailbox, msg);
   close(mailbox)
Q: open (mailbox); recv( mailbox, msg );
   close(mailbox)
```


The big debate in parallel computing: Messaging vs. Sharing Data

- Two programming models are equally powerful
- But result in very different-looking programming styles

- Most people find shared-data programming easier to work with
  - Debugging?

- What about machines that do not share memory?
  - Can be simulated in software [SDSM – hot topic in 80-90's]
  - But often not as efficient as message passing