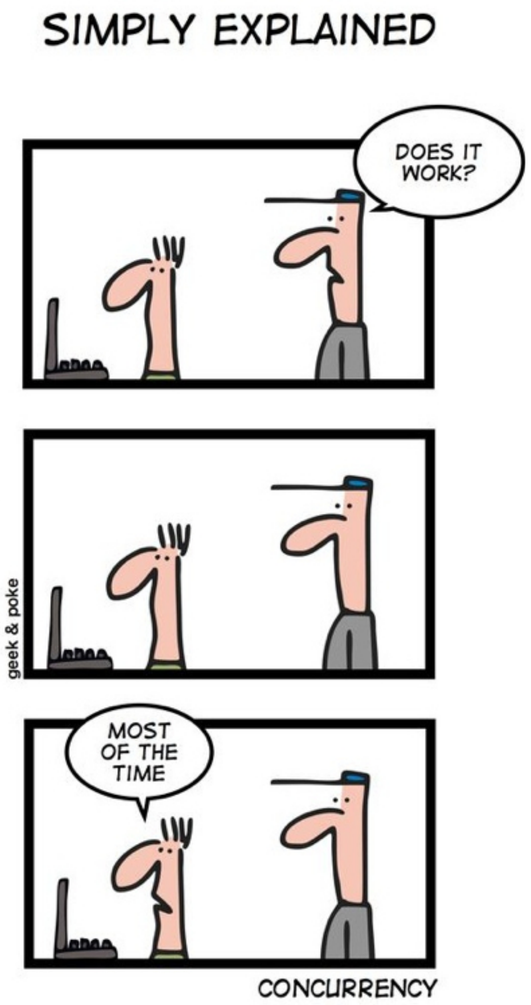
# *Concurrency (Part IV):* Mutual Exclusion, **Synchronization (Finish)**, **Deadlock, and Starvation**

Professor Travis Peters
CSCI 460 Operating Systems
Fall 2019

*Some slides & figures adapted from Stallings instructor resources.*

*Some slides adapted from Adam Bates's F'18 CS423 course @ UIUC*
*https://courses.engr.illinois.edu/cs423/sp2018/schedule.html*

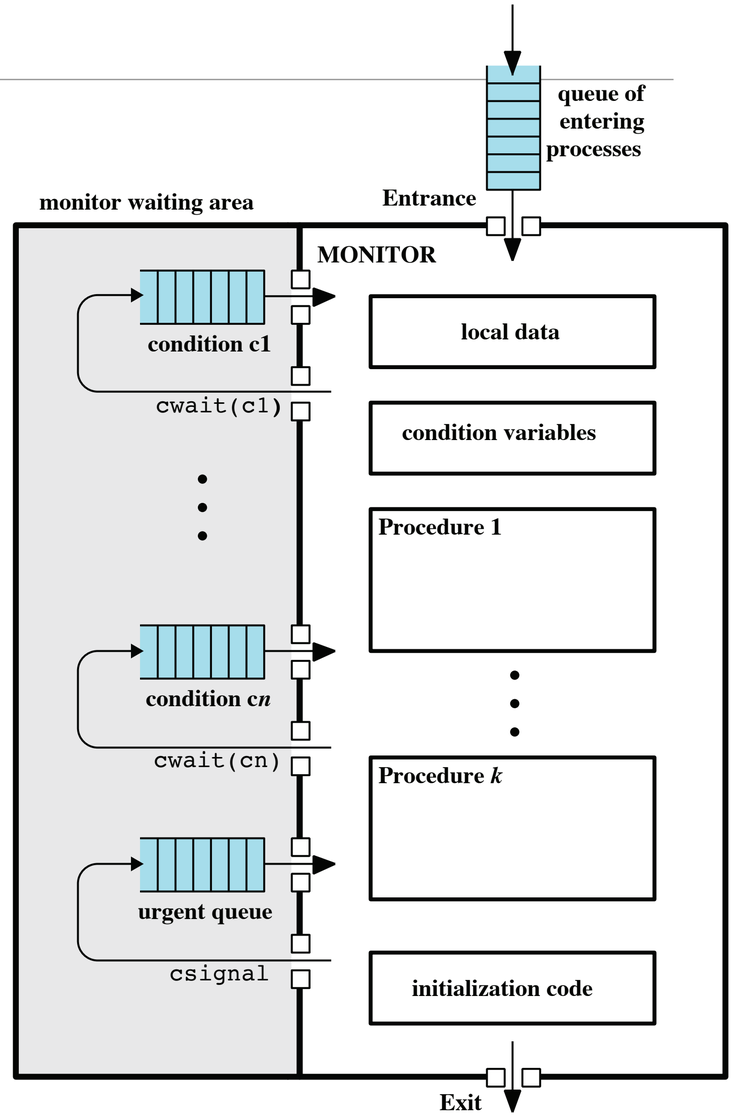



–http://www.datamation.com/news/tech-comics-quantum-physics-2.html

# Monitors

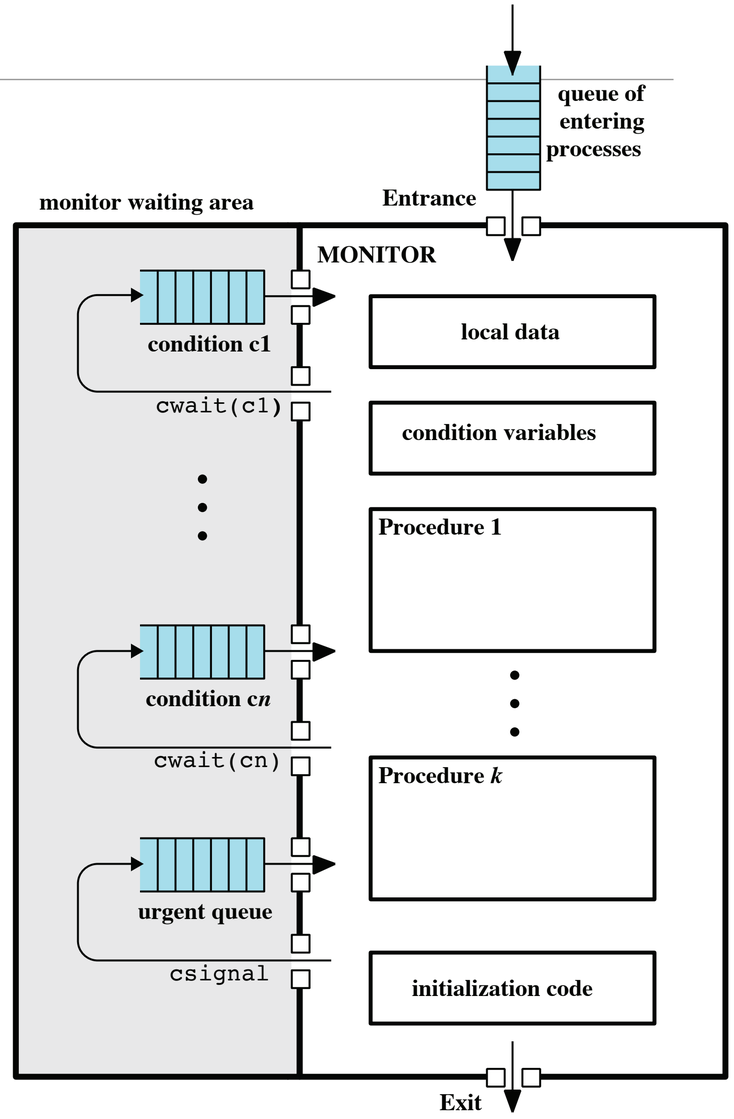
- A SW module consisting of…
  - an initialization sequence
  - 1+ procedures *—the only way for a process to enter the monitor*
  - local data *—accessible only by monitor's procedures; similar to objects in OOP*

- Eqivalent to semaphores, but easier!
  - Only one process may execute within the monitor at a time;
    all other processes are blocked until it becomes available again
    *=> Mutual Exclusion by design!*
  - Synchronization achieved via **condition variables**.
    - Used to represent a condition that needs to be waited on until the condition is True
    - No "value"
    - Think of it as a waiting queue (initial "non-value" = Empty)



# Monitors *(cont.)*

- Hoare-Style
  - Block caller (signaller) *immediately* and run the next waiting proc
  - Operations for condition variables (cvar):
    - `cwait( cvar )` *//suspend caller on condition `cvar`*
    - `csignal ( cvar )` *//resume some process waiting on condition `cvar`*
  - **Questions:** Advantages? Limitations? Drawbacks? Potential improvements?
- Mesa-Style
  - Called (signaller) keeps running and retains access to the monitor
  - Waiter placed on ready queue
  - *On resume, need to re-check conition!*
  - Operations for condition variables (cvar):
    - `notify( cvar )` *//resume the next waiting process at some convenient time (later)*
    - `broadcast ( cvar )` *//all procs waiting on cvar get moved to a ready state/queue*
  - **Questions:** Advantages? Limitations? Drawbacks? Potential improvements?



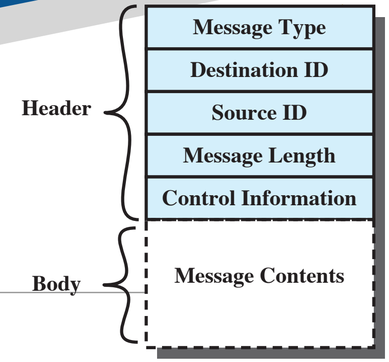
# Message Passing

- Operations
  - `send( dest, msg )`
  - `receive ( src, msg )`
- Operations come in diffferent flavors…
  - blocking send, blocking receive *(a.k.a. "rendezvous")*
  - nonblocking send, blocking receive *(most common)*
  - nonblocking send, nonblocking receive
- Addressing
  - direct addressing (e.g., specific process ID known)

    *vs.*

    indirect addresses (msgs sent to shared mailbox)

# Message Passing




- Operations
  - `send( dest, msg )`
  - `receive ( src, msg )`
- Operations come in diffferent flavors…
  - blocking send, blocking receive *(a.k.a. "rendezvous")*
  - nonblocking send, blocking receive *(most common)*
  - nonblocking send, nonblocking receive
- Addressing
  - direct addressing (e.g., specific process ID known)

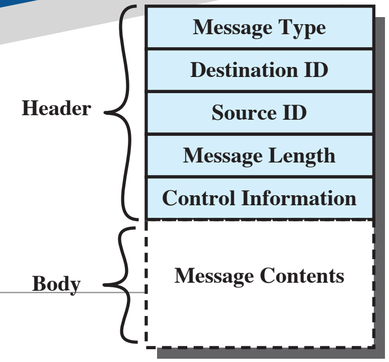
    *vs.*

    indirect addresses (msgs sent to shared mailbox)


(a) One to one

***Examples?***

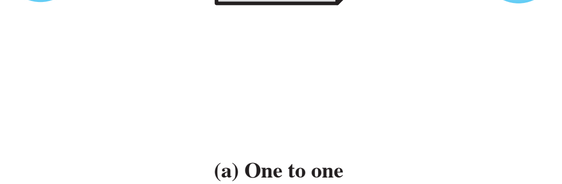
# Message Passing



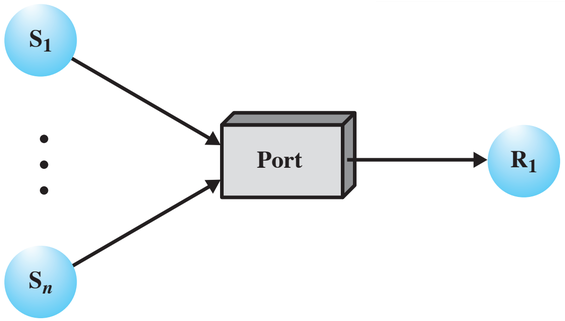

- Operations
  - `send( dest, msg )`
  - `receive ( src, msg )`
- Operations come in diffferent flavors…
  - blocking send, blocking receive *(a.k.a. "rendezvous")*
  - nonblocking send, blocking receive *(most common)*
  - nonblocking send, nonblocking receive
- Addressing
  - direct addressing (e.g., specific process ID known)

    *vs.*

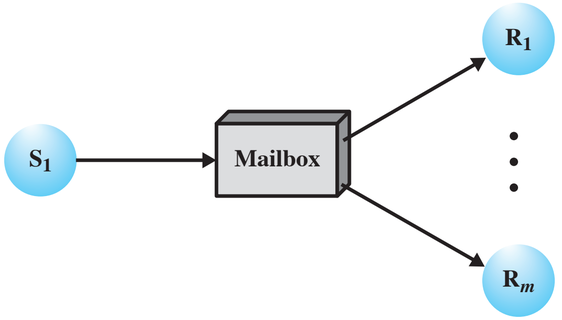
    indirect addresses (msgs sent to shared mailbox)




(a) One to one



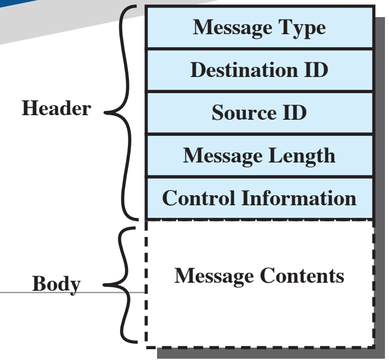

(b) Many to one




(c) One to many

***Examples?***

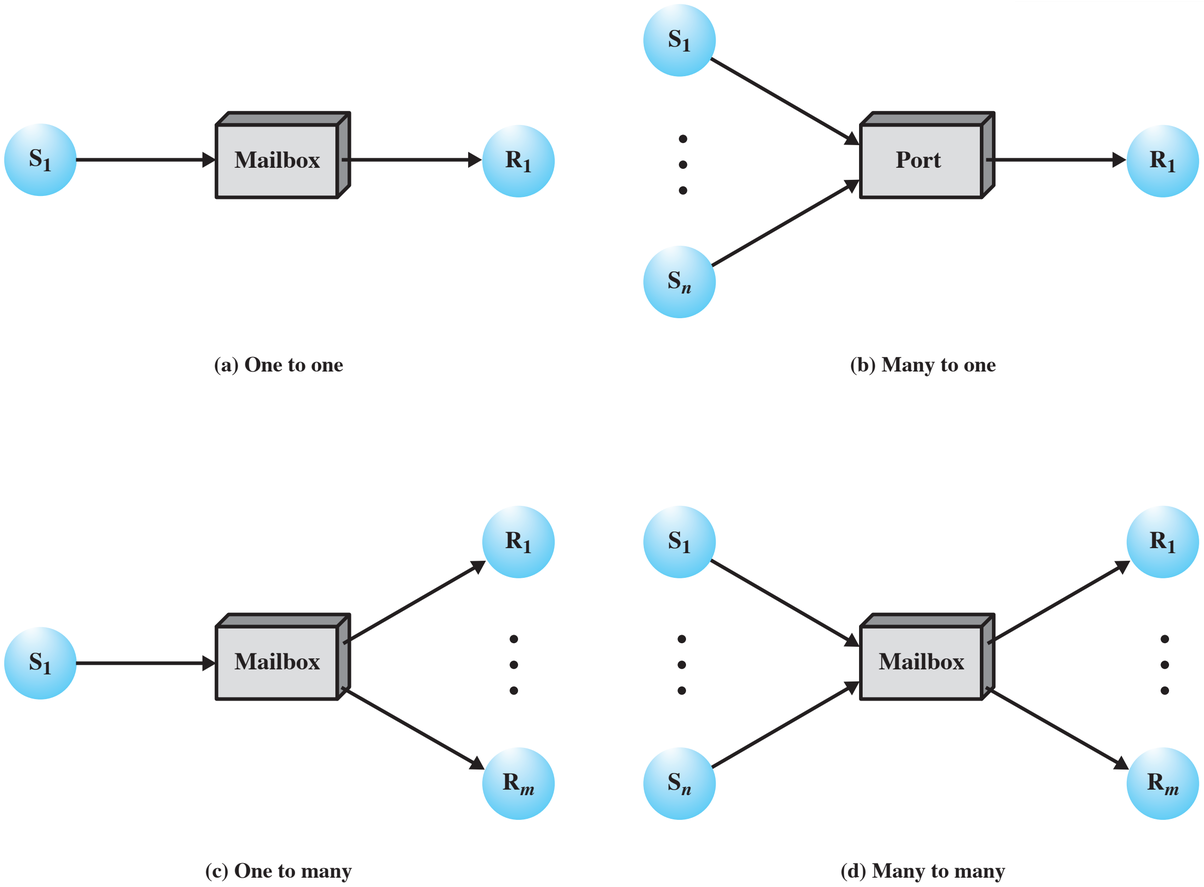
# Message Passing




- Operations
  - `send( dest, msg )`
  - `receive ( src, msg )`
- Operations come in diffferent flavors…
  - blocking send, blocking receive *(a.k.a. "rendezvous")*
  - nonblocking send, blocking receive *(most common)*
  - nonblocking send, nonblocking receive
- Addressing
  - direct addressing (e.g., specific process ID known)

    *vs.*

    indirect addresses (msgs sent to shared mailbox)




***Examples?***

*Be sure to review solutions for, e.g., Producer/Consumer with different styles of monitors, message passing schemes, etc.*

# Readers/Writers Problem

- Each process is either a ***reader*** or a ***writer***
- Both readers and writers share access to a data object
  *(e.g., file, database)*
- Multiple readers can access the data object simultaneously
- Each writer must have exclusive access
  *(i.e., cannot share w/ readers OR any other writer)*

# Readers/Writers Problem

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
     semWait (x);
     readcount++;
     if (readcount == 1) semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true) {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

***Priority goes to readers…***

```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

***Priority goes to writers…***