Wait-Free Synchronization, Communication with Messages, Dining Philosopher

ECE469

Jan 31

Yiying Zhang

Roadmap

- Interprocess communication *with shared data*
  - Synchronization with locks, semaphores, condition var
  - Classic sync. problems 1, 2
  - Semaphore implementations (uniprocessor, multiprocessor)

Today:

- Wait-free synchronization
- Interprocess communication *with messages*
- Classic sync. Problem 3

[lec6] Uniprocessor solution: disable interrupts!

```
void  wait(semaphore s)
{
  disable interrupts;
  if (s->count > 0) {
      s->count --;
      enable interrupts;
      return;
  }
  add(s->q, current_process);
  enable interrupts;
  sleep(); /* re-dispatch */
}
```

```
void  signal(semaphore s)
{
   disable interrupts;
   if (isEmpty(s->q)) {
     s->count ++;
   } else {
     process = removeFirst(s->q);
     wakeup(process);
     /* put process on Ready Q */
   }
   enable interrupts;
}
```

[lec6] Use TAS to implement semaphores on multiprocessor?

```
void  wait(semaphore s)
{
   disable interrupts;
  while (1 == tas(&lock,1));
   if (s->count > 0) {
       s->count --;
       lock = 0;
       enable interrupts;
       return;
   }
   add(s->q, current_process);
   lock=0;
   sleep(); /* re-dispatch */
   enable interrupts;
}
```

```
void  signal(semaphore s)
{
    disable interrupts;
    while (1 == tas(&lock,1));
    if (isEmpty(s->q)) {
       s->count ++;
    } else {
       thread = removeFirst(s->q);
       wakeup(process);
       /* put process on Ready Q */
     }
    lock = 0;
     enable interrupts;
}
```

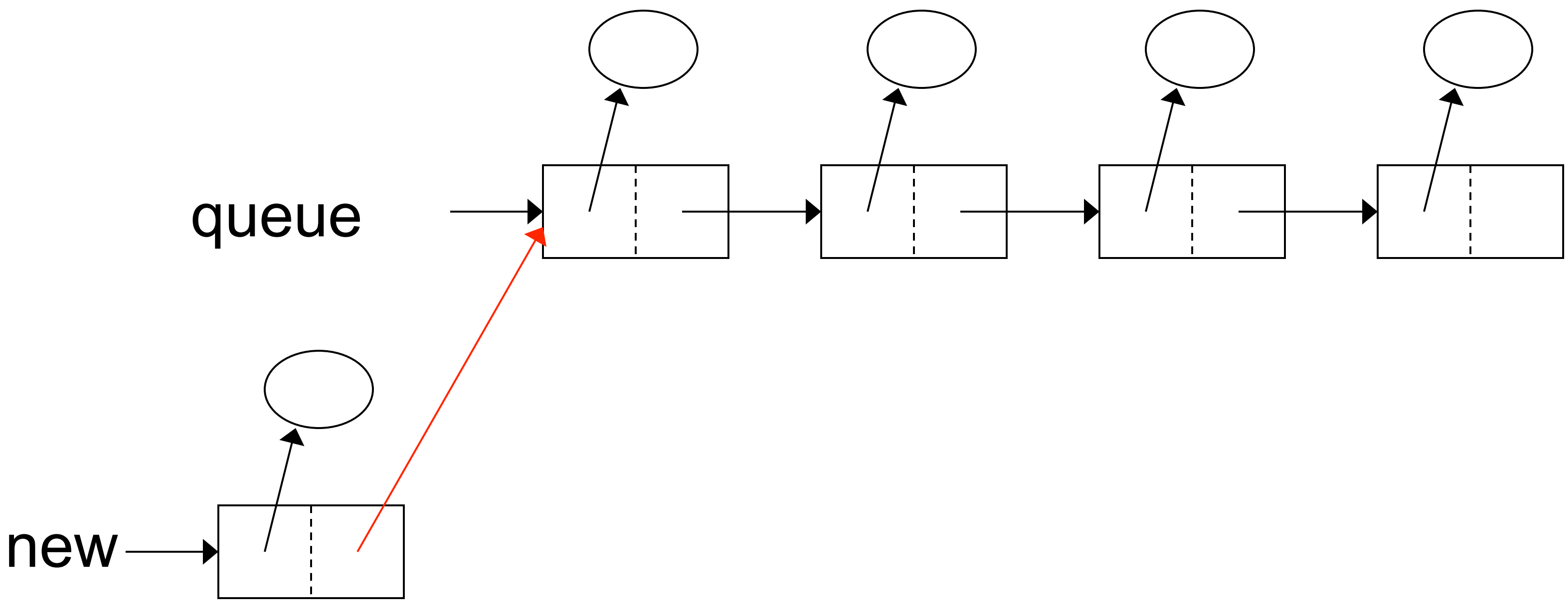
Wait-free Synchronization

- Finally we need tsa or ldl&stc anyway to implement sync. primitives (on multiprocessors)

- Can we design data structures in a way that allows safe concurrent accesses?
  - no mutual exclusion necessary
  - no possibility of deadlock
  - only using tsa / ldl^stc
  - no busy waiting

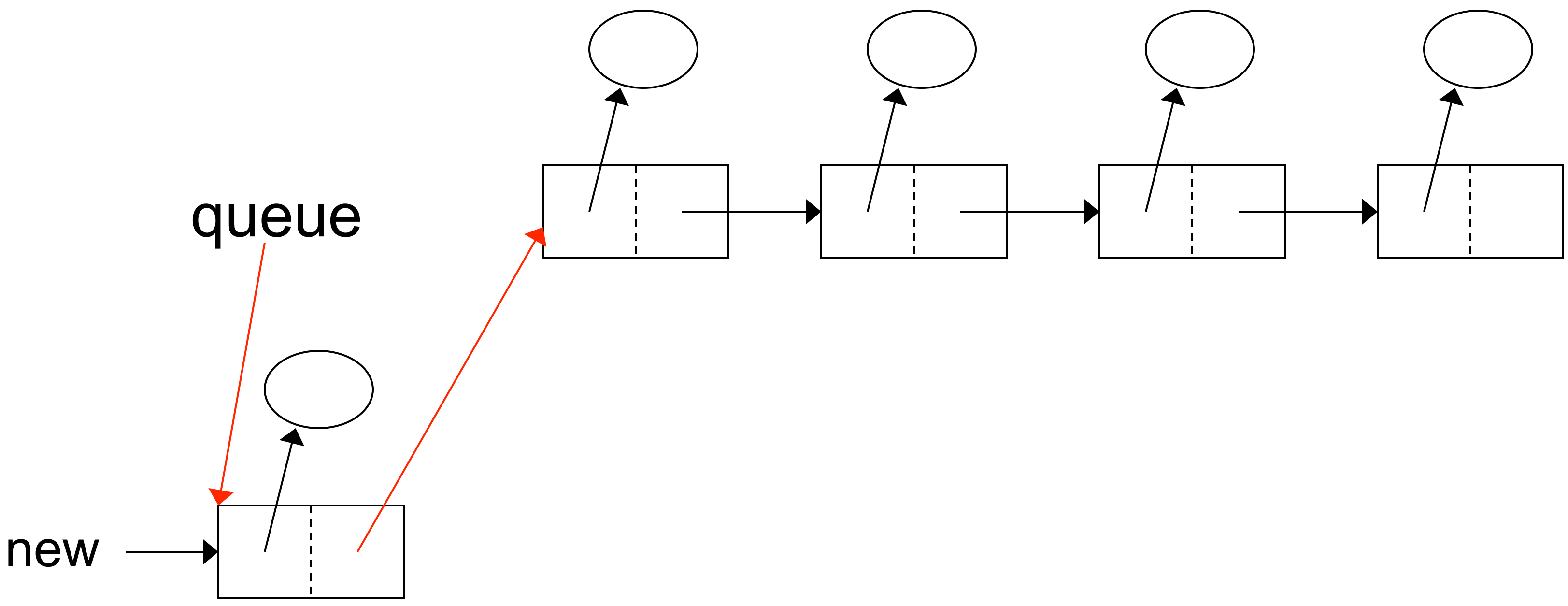
Simple example – Queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



Simple example – Queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



Singly-linked Queue Insertion

```
QElem *queue;


void Insert(item) {
   QElem *new = malloc(sizeof(QElem));
   new->item = item;
   new->next = queue;
   queue = new;
}
```

Wait-free Synchronization

- Design data structures in a way that allows safe concurrent accesses
  - no mutual exclusion (lock acquire & release) necessary
  - no possibility of deadlock

- Approach: use a single atomic operation to
  - commit all changes
    - move the shared data structure from one consistent state to another

[lec6] Read-modify-write on CISC

- Most CISC machines provide atomic *read-modify-write* instruction
  - read existing value
  - store back a new value
  - Example: *test-and-set* by IBM and others

```
int TAS(int *old_ptr, int new {
 int old = *old_ptr; // fetch old value at old_ptr
 *old_ptr = new; // store 'new' into old_ptr
 return old; // return the old value
}
```

  - Using TAS to implement lock (mutex)

Singly-linked Queue Insertion

```
QElem *queue;


void Insert(item) {
   QElem *new = malloc(sizeof(QElem));
   new->item = item;
   do {
          new->next = queue;
   } while (tas(&queue, new) != new->next);
}
```

Is this busy waiting a problem?

Limitation

- Example only works for simple data structures where changes can be committed with *one store instruction*

- What about more complex data structures?

More General Approach

- Maintain a pointer to the “master copy” of the data structure
- To modify,
  1. remember current value of the master pointer
  2. copy shared data structure to a scratch location
  3. modify copy
  4. *atomically*:
     - verify that master pointer has not changed
     - write pointer to refer to new master
  5. if verification fails (another process interfered), start over at step 1

- Downside?
  - When does it work reasonably well?

[lec6] Load-Linked and Store-Conditional on RISC [MIPS R4000 series]

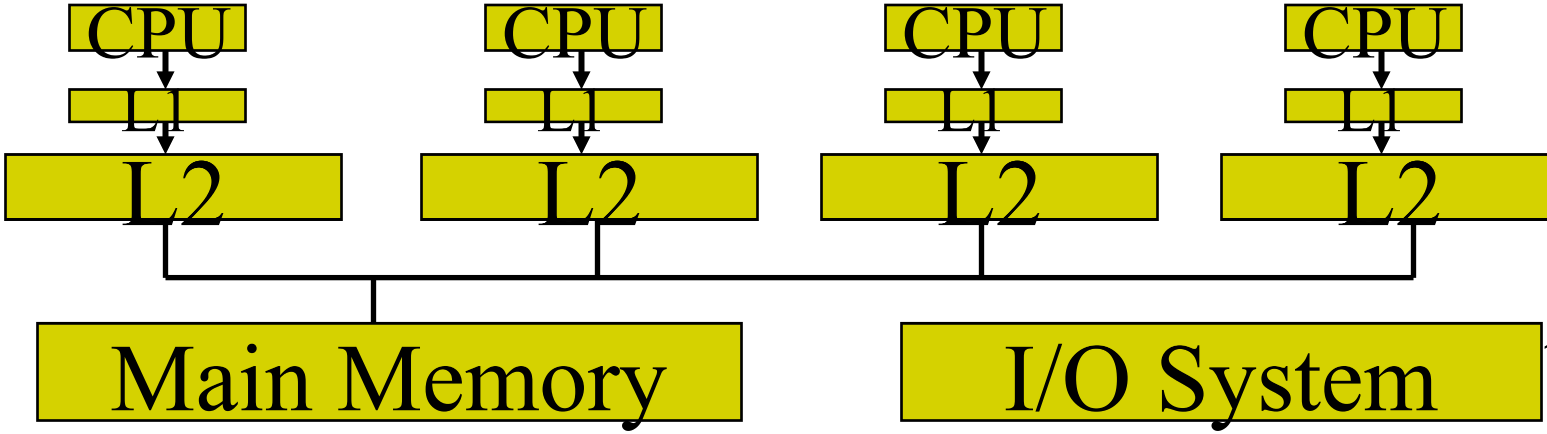
Load-linked instruction: LDL Rx,y

- loads Rx with a word from mem addr y
- holds y in per-processor lock register

Store operation to addr y (by any processor) resets all other processors' lock registers if containing addr y

Store-conditional instruction: STC Rx, y

- stores a word iff y matches the processor's lock register
- indicates success (1) or failure (0)



Atomically Move to New Copy Approach

```
Retry:
Sav = master;

Copy master content to master_tmp
Do the work use master_tmp..

if (sav != ldl(& master) ||  stc(&master, master_tmp)!= success)
        goto retry;
```

Roadmap

- Interprocess communication *with shared data*
  - Synchronization with locks, semaphores, condition var
  - Classic sync. problems 1, 2
  - Semaphore implementations (uniprocessor, multiprocessor)

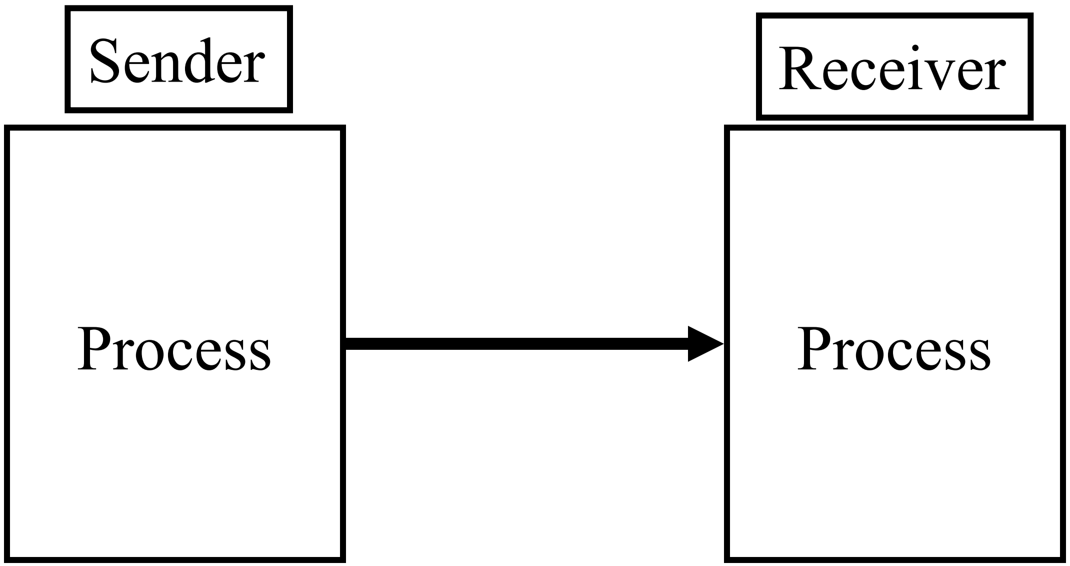
Today:

- Wait-free synchronization
- Interprocess communication *with messages*
- Classic sync. Problem 3

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 different programmers who aren’t familiar with code
  - They might not trust each other
  - They may 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 full buffer

- *Asynchronous*
  - Return immediately
  - Completion
    - Require the application to check status (appl polls)
    - Notify the application (OS sends interrupt)
  - Block on full buffer

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

Buffering

- No buffering
  - Sender must wait until the receiver receives the message
  - Rendezvous on each message

- Bounded buffer
  - Finite size
  - Sender blocks when the buffer is full
  - Receiver blocks when the buffer is empty
  - 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
    …
  }
}
```

Break

- You have 32 numbers. What is the least number of comparison needed to find the 2nd smallest out of them? (in general, finding the 2nd smallest out of N numbers)

Indirect Communication

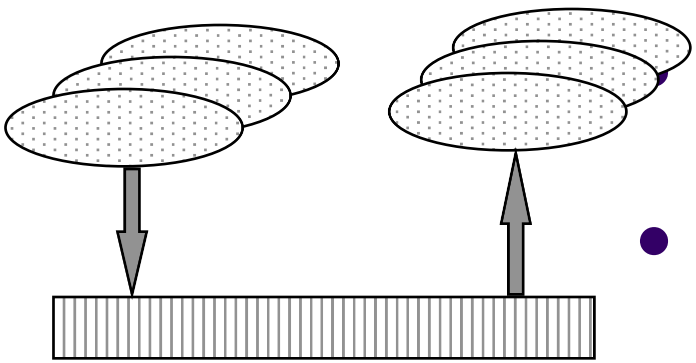
- Use a “mailbox” or “ports” to allow many-to-many communication
  - Mailbox typically owned by the OS
    - Requires open/close a mailbox before allowed to use it
- A “link”
  - is set up among processes only if they have a shared mailbox
  - Can be associated with more than two processes

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

Indirect Communication (cont)

- Where should the buffer be?
  - A buffer and its mutex/conditions should be at the mailbox

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

Mailbox - Implementation

- For each Mailbox structure (msgQ_t)
  - Circular buffer
    - A counter (used) to track number of processes that have opened the mailbox
    - Implemented as fixed size array (buffer[BUFFER_SIZE])
    - Indices (head and tail) indicate next free byte and first byte of top message in the circular buffer
    - A counter (count) to track number of variable length, queued messages
    - SPACE_AVAILABLE((msgQ_t)q) computes buf space
  - Synchronization
    - A lock (l) and two condition variables (moreSpace and moreData) for producers and consumers to wait on
    - If a producer's message doesn't fit it waits on moreSpace
    - If a consumer finds no messages it waits on more data

Mailbox – Implementation cont’d

- mbox_init()
  - Initialize all the mailboxes (Q[MAX_Q])
    - Reset usage count, synchronization structures
- mbox_open(q)+
  - Acquires access to the q^{th} mailbox
    - Clear out the buffer if unused (adjust head, tail)
    - Increment the usage count
    - Returns q if q is valid box number, else –1
- mbox_close(q)+
  - Relinquishes access to the q^{th} mailbox
    - Decrement the usage count
- mbox_stat(key,*count,*space)+
  - Queries the q^{th} mailbox for its status
    - Fill in the message count and free space

+ Be sure to use the mailbox lock appropriately

Mailbox – Implementation cont’d

- Messages structure
  - Struct msg_t {int size,char body[0]}
    - Msg size in bytes: sizeof(int) + (msg_t)m->size
    - MSG_SIZE(m) macro computes this
- mbox_send(q,m)
  - Put message m into the q^{th} mailbox
    - Use lock and monitor to wait until there’s space
    - Copy m into the buffer (note that buffer wraps)
    - Update head pointer and broadcast moreData
- mbox_recv(q,*m)
  - Get the top message from the q^{th} mailbox and copy into m
  - Broadcast moreSpace

[t] No CRITICAL_SECTIONs used/assumed

Issues related to Mailboxes

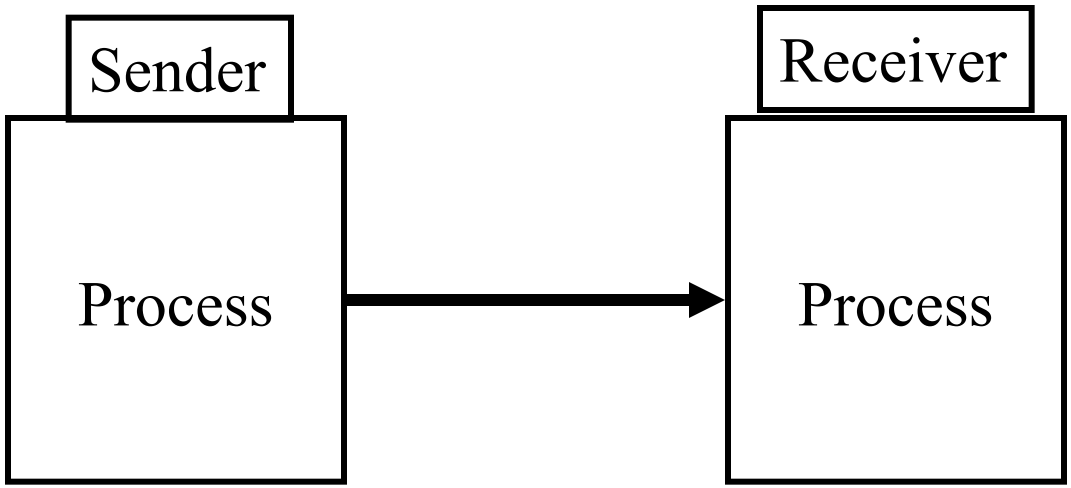
- Asynchronous or synchronous?
- How are links established?
- Mailbox between more than two processes?
- How many links can there be between any pair?
- Directional or bidirectional?
- Direct or indirect?
- Exceptions: process termination, message loss?

Exception: Messages can get lost over the network

- Detection
  - Acknowledge each message sent
  - Timeout on sender (OS)

- Retransmission
  - Retransmit on timeout
  - Sequence number for each message
  - Remove duplication messages on the receiver side
  - Multiple outstanding messages → Retransmit on out-of-sequence acknowledgement

Exception: Process Termination



- S has terminated
  - Problem: R may be blocked forever
  - Solution: R pings S once a while
- R has terminated
  - Problem: S runs out buffer and will be blocked forever
  - Solution: S checks on R occasionally

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

Roadmap

- Interprocess communication *with shared data*
  - Synchronization with locks, semaphores, condition var
  - Classic sync. problems 1, 2
  - Semaphore implementations (uniprocessor, multiprocessor)

Today:

- Wait-free synchronization
- Interprocess communication *with messages*
- Classic sync. Problem 3

Classic Synchronization Problems

1. Producer-consumer problem (bounded buffer problem)

2. Readers-writers problem

3. Dining philosophers problem

Dining Philosopher' s Problem

- Dijkstra 1971

- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time

Dining philosophers problem

Abstraction of concurrency-control problems

The need to allocate several resources among several processes while being deadlock-free and starvation-free

Rules of the Game

- The philosophers are very logical
  - They want to settle on a shared policy that all can apply concurrently
  - They are hungry: the policy should let everyone eat (eventually)
  - They are utterly dedicated to the proposition of equality: the policy should be totally fair

Basic Operation of Each Philosopher

```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

Helper functions:

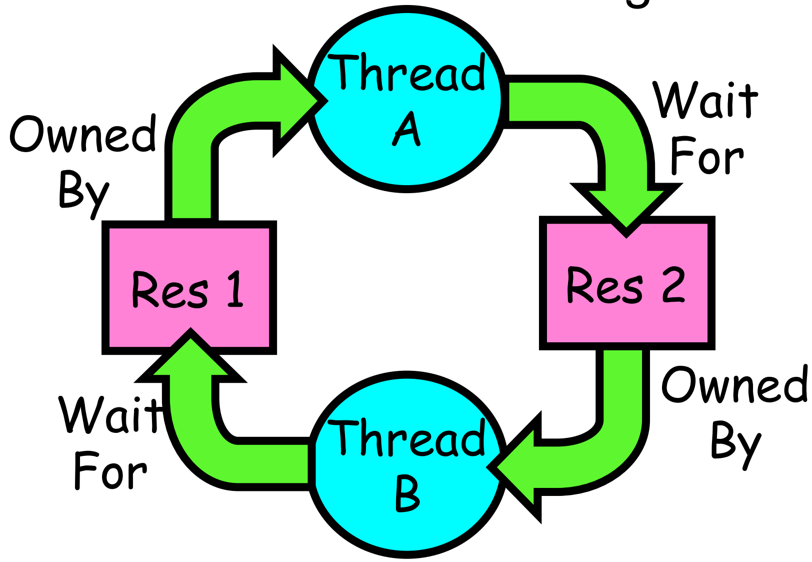
```
int left(int p) { return p; }
int right(int p) { return (p + 1) % 5; } // Assuming 5 philosophers
sem forks[5]; // semaphore for the 5 forks
```

What can go wrong?

- Primarily, we worry about:
  - Starvation: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)
  - Deadlock: A policy that leaves all the philosophers "stuck", so that nobody can do anything at all
  - Livelock: A policy that makes them all do something endlessly without ever eating!

Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock ⇒ Starvation but not vice versa
    - Starvation can end (but doesn't have to)
    - Deadlock can't end without external intervention

A flawed conceptual solution

```
void getforks() {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
}

void putforks() {
        sem_post(forks[left(p)]);
        sem_post(forks[right(p)]);
}
```

Oops! Subject to deadlock if they all pick up their “right” fork simultaneously!

Dijstra's Solution

```
void getforks() {
        if (p == 4) {
                sem_wait(forks[right(p)]);
                sem_wait(forks[left(p)]);
        } else {
                sem_wait(forks[left(p)]);
                sem_wait(forks[right(p)]);
        }
}
```

Other Dining Philosophers Solutions

- Allow only 4 philosophers to sit simultaneously
- Asymmetric solution
  - Odd philosopher picks left fork followed by right
  - Even philosopher does vice versa
- Pass a token
- Allow philosopher to pick fork only if both available

Solutions are less interesting than the problem itself!

- In fact the problem statement is why people like to talk about this problem!
- Rather than solving Dining Philosophers, we should use it to understand properties of solutions that work and of solutions that can fail!