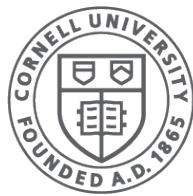


Actors, Barriers, Interrupts

CS 4410
Operating Systems



Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[Robbert van Renesse]

Havender's Scheme (OS/360)

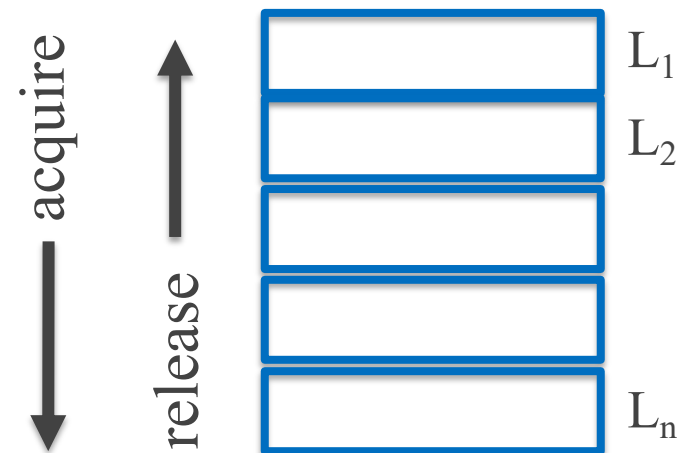
Hierarchical Resource Allocation

Every resource is associated with a level.

- **Rule H1:** All resources from a given level must be acquired using a single request.
- **Rule H2:** After acquiring from level L_j must not acquire from L_i where $i < j$
- **Rule H3:** May not acquire from L_i unless already released from L_j where $j > i$.

Example of allowed sequence:

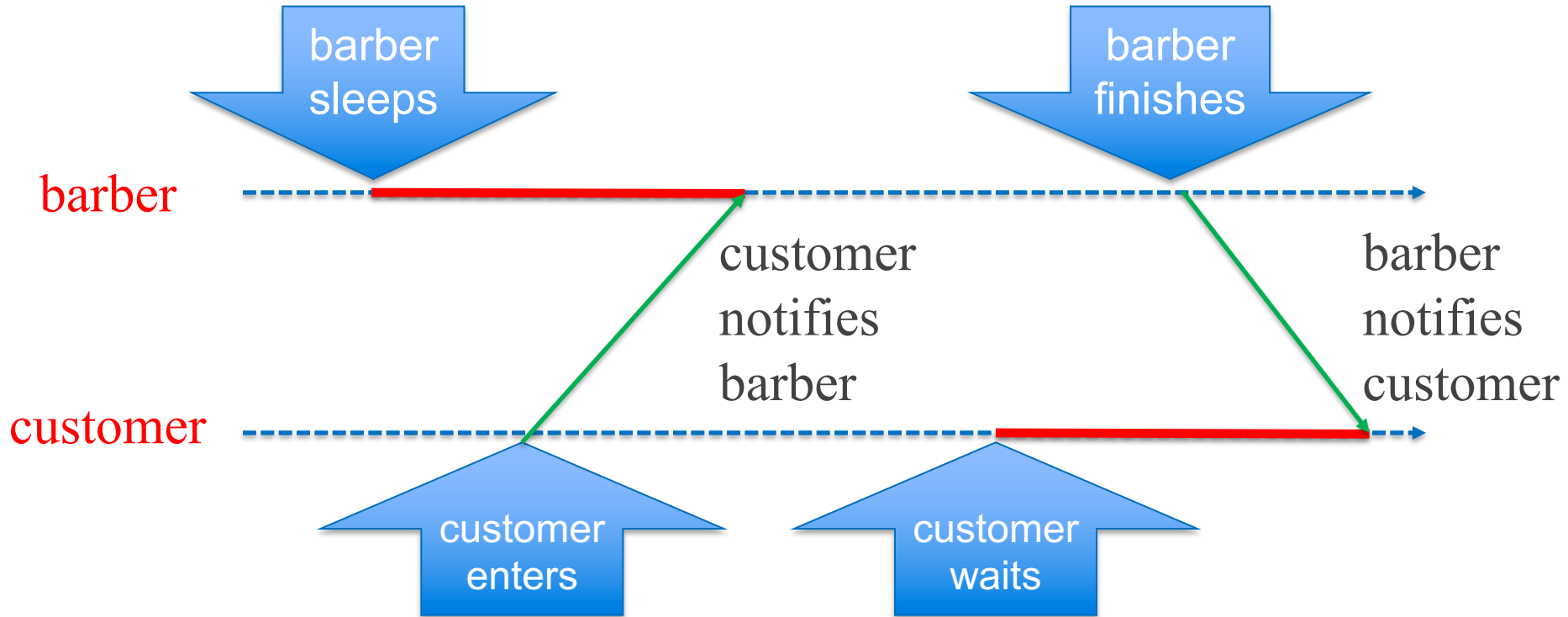
1. `acquire(W@L1, X@L1)`
2. `acquire(Y@L3)`
3. `release(Y@L3)`
4. `acquire(Z@L2)`



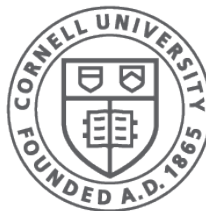
Sleeping Barber Problem



Sleeping Barber Problem



Actor Synchronization



Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Actor Model

- An *actor* is a process
- Each actor has an incoming *message queue*
- **No other shared state**
- Actors communicate by “message passing”
 - placing messages on message queues
- Supports modular development of concurrent programs

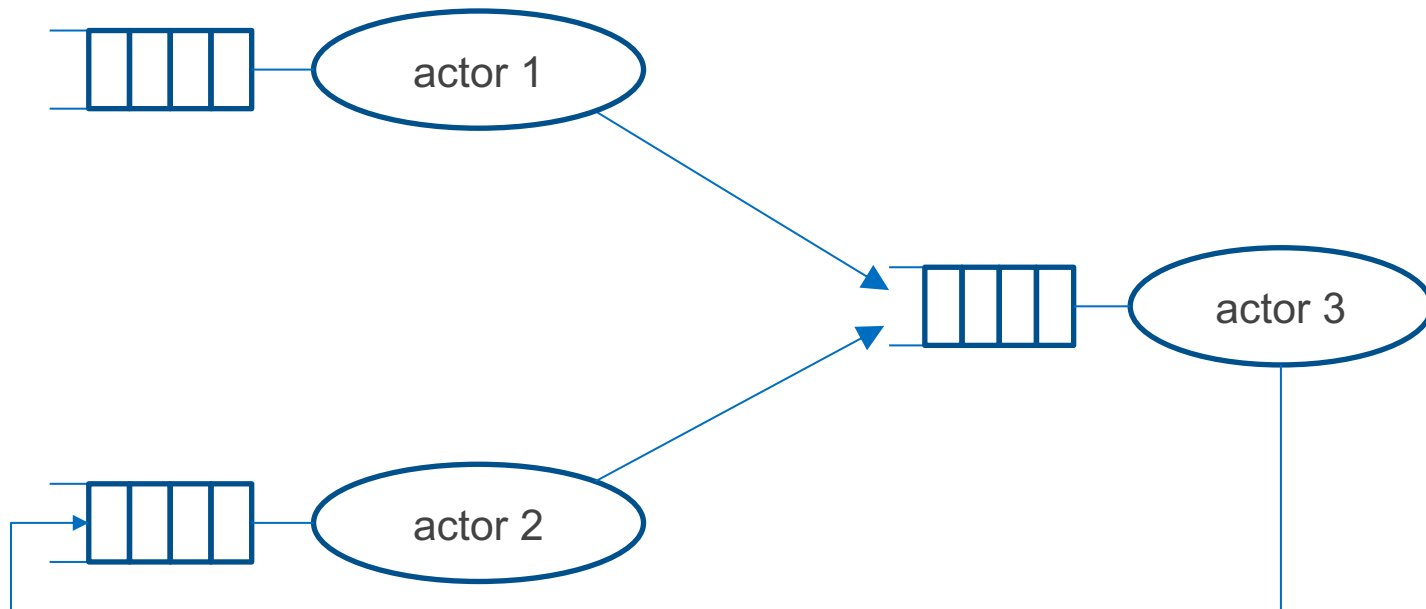
Reminiscent of event-based programming, but each actor has local state

Mutual Exclusion with Actors

- Data structure owned by a “server actor”
- Client actors can send request messages to the server and receive response messages if necessary
- Server actor awaits requests on its queue and executes one request at a time



- Mutual Exclusion (one request at a time)
- Progress (requests eventually get to the head of the queue)
- Fairness (requests are handled in FCFS order)

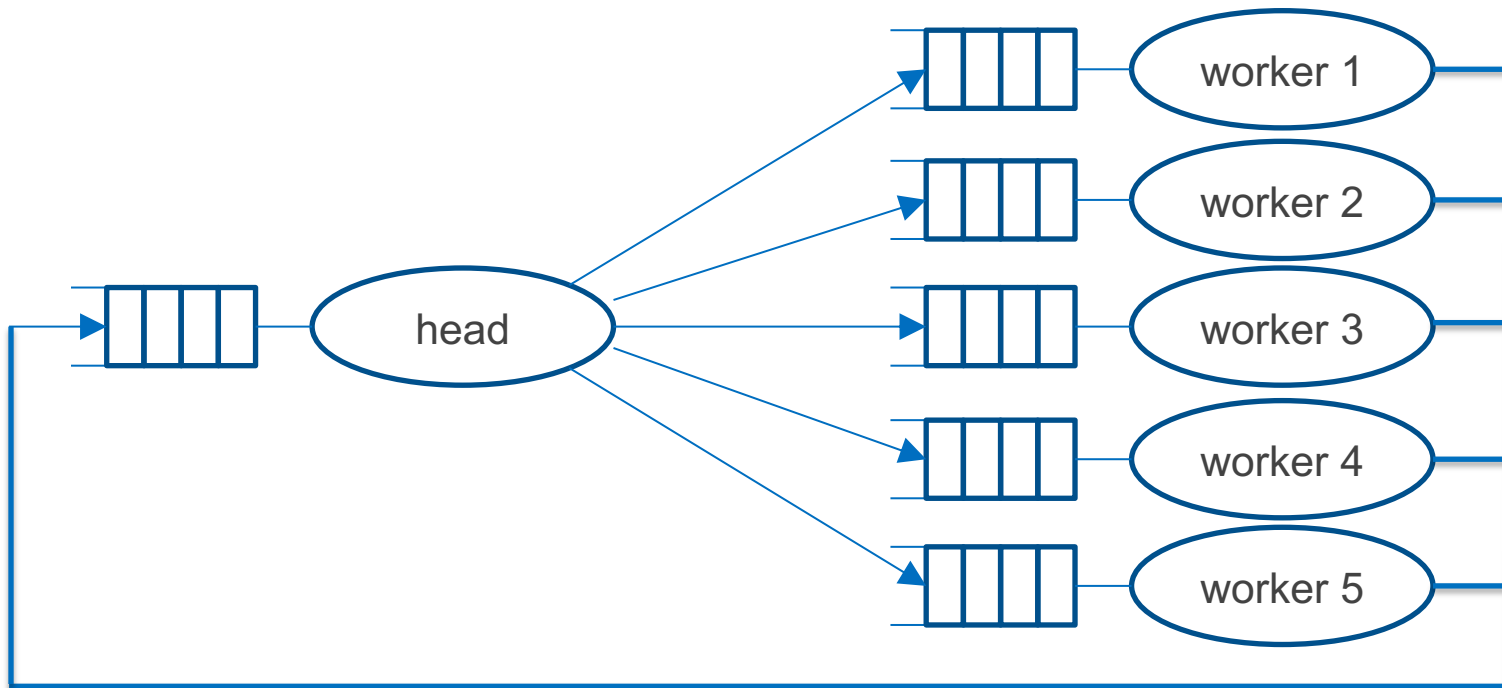


Conditional Critical Sections with Actors

- An actor can “wait” for a condition by waiting for a specific message
- An actor can “signal/notify” another actor by sending it a message

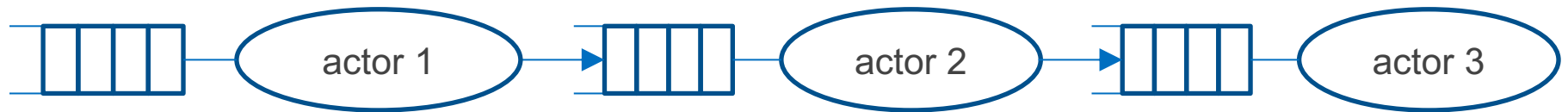
Parallel processing with Actors

- Organize program with a Master Actor and a collection of Worker Actors
- Master Actor sends work requests to the Worker Actors
- Worker Actors send completion requests to the Master Actor



Pipeline Parallelism with Actors

- Organize program as a chain of actors
- For example, REST/HTTP server
 - Network receive actor → HTTP parser actor → REST request actor → Application actor → REST response actor → HTTP response actor → Network send actor



Support for actors in programming languages

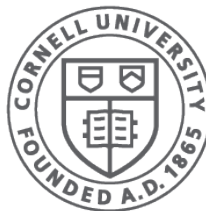
- Native support in languages such as Scala and Erlang
- "blocking queues" in Python, Harmony, Java
- Actor support libraries for Java, C, ...

Actors also nicely generalize to distributed systems!

Actor disadvantages?

- Doesn't work well for “fine-grained” synchronization
 - overhead of message passing much higher than lock/unlock
- Marshaling/unmarshaling messages just to access a data structure leads to significant extra code

Barrier Synchronization



Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Barrier Synchronization: the opposite of mutual exclusion...

- Set of processes run in rounds
- Must all complete a round before starting the next
- Popular in simulation, HPC, graph processing, ...
- During review we saw the “Tea House” example

Barrier Synchronization in Harmony

```
import synch;
```

```
const NROUNDS = 3;
```

```
const NPROC = 3;
```

```
# check that all non-None values in round are the same
```

```
def check():
```

```
    result = True;
```

```
    let x = None:
```

```
        for i in {0..NPROC-1}:
```

```
            if result and (round[i] != None):
```

```
                if x != None:
```

```
                    result = round[i] == x;
```

```
                ;
```

```
                x = round[i];
```

```
            ;
```

```
        ;
```

```
    ;
```

```
;
```

```
def process(self):
```

```
    for r in {0..NROUNDS-1}:
```

```
        barrier_enter(self);
```

```
        round[self] = r;
```

```
        assert check();
```

```
        round[self] = None;
```

```
        barrier_exit(self);
```

```
    ;
```

```
    done[self] = True;
```

```
    ;
```

```
def main():
```

```
    await all(done);
```

```
    ;
```

```
    round = [None,] * NPROC;
```

```
    done = [False,] * NPROC;
```

```
    for i in {0..NPROC-1}:
```

```
        spawn process(i);
```

```
    ;
```

```
    spawn main();
```

Barrier Synchronization in Harmony

```
import synch;
```

```
const NROUNDS = 3;  
const NPROC = 3;
```

check that all non-None values in round are the same

```
def check():
```

```
    result = True;
```

```
    let x = None:
```

```
        for i in {0..NPROC-1}:
```

```
            if result and (
```

```
                if x != None
```

```
                    result = round[i] == x;
```

```
            ;
```

```
            x = round[i];
```

```
        ;
```

```
    ;
```

```
    ;
```

```
;
```

no processes in
different rounds

all processes finish
all rounds

```
def process(self):
```

```
    for r in {0..NROUNDS-1}:
```

```
        barrier_enter(self);
```

```
        round[self] = r;
```

```
        assert check();
```

```
        round[self] = None;
```

```
        barrier_exit(self);
```

```
    ;
```

```
    done[self] = True;
```

```
;
```

```
def main():
```

```
    await all(done);
```

```
;
```

```
round = [None,] * NPROC;
```

```
done = [False,] * NPROC;
```

```
for i in {0..NPROC-1}:
```

```
    spawn process(i);
```

```
;
```

```
spawn main();
```

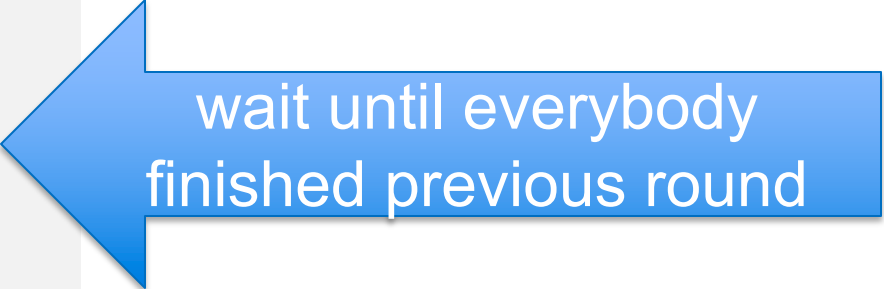
Actor-style Barriers

```
def barrier_enter(self):  
    for i in {0..NPROC-1} such that i != self:  
        enqueue(?queues[i], i);  
    ;  
    for i in {1..NPROC-1}:  
        dequeue(?queues[self]);  
    ;  
;  
def barrier_exit(self):  
    pass;  
;  
queues = [ Queue() for i in {0..NPROC} ];
```

*Each actor sends a message
to one another and then
waits for peers' messages*

RVR-style Barriers

```
def barrier_enter(self):
    lock(?mutex);
    while bstate >= NPROC:
        wait(?start);
    ;
    bstate += 1;
    if bstate == NPROC:
        notifyAll(?finish);
    ;
    unlock(?mutex);
;
def barrier_exit(self):
    lock(?mutex);
    while bstate < NPROC:
        wait(?finish);
    ;
    bstate = (bstate + 1) % (2 * NPROC);
    if bstate == 0:
        notifyAll(?start);
    ;
    unlock(?mutex);
;
mutex = Lock();
start = Condition(?mutex);
finish = Condition(?mutex);
bstate = 0;
```



wait until everybody
finished previous round



wait until everybody
started current round

bstate = 0..*NPROC* - 1:

processes are entering

bstate = *NPROC*..*2*NPROC* - 1:

processes are leaving

”Double Turnstile” barriers



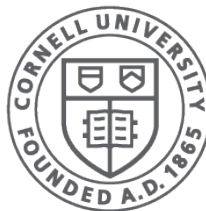
”Double Turnstile” barriers

```
def barrier_enter(self):  
    lock(?mutex);  
    nstarting += 1;  
    if nstarting < NPROC:  
        while nstarting < NPROC:  
            wait(?start);  
        ;  
    else:  
        nfinishing = 0;  
        notifyAll(?start);  
    ;  
    unlock(?mutex);  
    ;
```

```
def barrier_exit(self):  
    lock(?mutex);  
    nfinishing += 1;  
    if nfinishing < NPROC:  
        while nfinishing < NPROC:  
            wait(?finish);  
        ;  
    else:  
        nstarting = 0;  
        notifyAll(?finish);  
    ;  
    unlock(?mutex);  
    ;
```

```
mutex = Lock();  
start = Condition(?mutex);  
finish = Condition(?mutex);  
nstarting = 0;  
nfinishing = 0;
```

Interrupt Handling



Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Interrupt handling

- When executing in user space, a device interrupt is invisible to the user process
- state of user process is unaffected by the device interrupt and its subsequent handling
- This is because contexts are switched back and forth

Interrupt handling

- However, there are also “in-context” interrupts:
 - kernel code can be interrupted
 - user code can handle “signals”
- *Potential for race conditions*

“Traps” in Harmony

```
1      def handler():
2          count += 1;
3          done = True;
4      ;
5      def main():
6          trap handler();
7          await done;
8          assert count == 1;
9      ;
10     count = 0;
11     done = False;
12     spawn main();
```



invoke handler() at
some future time

Within the same process!
(trap ≠ spawn)

But what now?

```
1      def handler():
2          count += 1;
3          done = True;
4      ;
5      def main():
6          trap handler();
7          count += 1;
8          await done;
9          assert count == 2;
10     ;
11     count = 0;
12     done = False;
13     spawn main();
```

But what now?

```
1      def handler():
2          count += 1;
3          done = True;
4      ;
5      def main():
6          trap handler();
7          count += 1;
8          await done;
9          assert count == 2;
10     ;
```

```
#states = 20 diameter = 2
==== Safety violation ====
__init__/( ) [0,27-35] 35 { count: 0, done: False }
main/( ) [10-16,Interrupt,1-8,17-24] 24 { count: 1, done: True }
>>> Harmony Assertion (file=trap2.hny, line=9) failed
```

Locks to the rescue?

```
1      import synch;
2
3      def handler():
4          lock(?countlock);
5          count += 1;
6          unlock(?countlock);
7          done = True;
8      ;
9      def main():
10         trap handler();
11         lock(?countlock);
12         count += 1;
13         unlock(?countlock);
14         await done;
15         assert count == 2;
16     ;
17     countlock = Lock();
18     count = 0;
19     done = False;
20     spawn main();
```

Locks to the rescue?

```
1  import synch;
2
3  def handler():
4      lock(?countlock);
5      count += 1;
6      unlock(?countlock);
7      done = True;
8
9  def main():
10     trap handler();
11     lock(?countlock);
12     count += 1;
13     unlock(?countlock);
14     await done;
```

```
#states = 27 diameter = 2
#components: 21
#bad components: 3
==== Non-terminating State ====
__init__/( ) [0,864-866,584-587,867-876] 876 { count: 0, countlock: False, done:
False }
main/( ) [839-845,589-592,573-582,593-594,846,Interrupt,822-825,589-592,573] 574
{ count: 0, countlock: True, done: False }
blocked process: main/( ) pc = 574
#blocked: 1 #stopped: 0 #running: 0
```

Enabling/disabling interrupts

```
1      import synch;
2
3      def handler():
4          count += 1;
5          done = True;
6      ;
7      def main():
8          trap handler();
9          setintlevel(True);
10         count += 1;
11         setintlevel(False);
12         await done;
13         assert count == 2;
14     ;
15     mutex = Lock();
16     count = 0;
17     done = False;
18     spawn main();
```

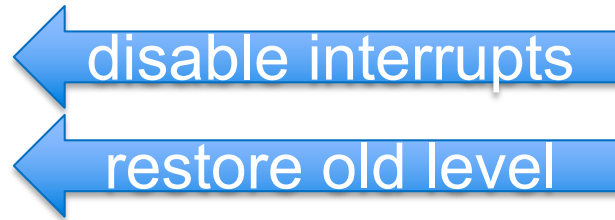
Enabling/disabling interrupts

```
1      import synch;
2
3      def handler():
4          count += 1;
5          done = True;
6      ;
7      def main():
8          trap handler();
9          setintlevel(True);
10         count += 1;
11         setintlevel(False);
12         await done;
13         assert count == 2;
14     ;
15     mutex = Lock();
16     count = 0;
17     done = False;
18     spawn main();
```

```
#states = 11 diameter = 2
#components: 11
no issues found
```

Interrupt-Safe Methods

```
1  import synch;
2
3  def increment():
4      let prior = setintlevel(True);
5          count += 1;
6          setintlevel(prior);
7      ;
8  ;
9  def handler():
10     increment();
11     done = True;
12 ;
13 def main():
14     trap handler();
15     increment();
16     await done;
17     assert count == 2;
18 ;
19 mutex = Lock();
20 count = 0;
21 done = False;
22 spawn main();
```



Interrupt-safe *AND* Thread-safe?

```
1  import synch;
2
3  def increment():
4      let prior = setintlevel(True);
5          lock(?countlock);
6          count += 1;
7          unlock(?countlock);
8          setintlevel(prior);
9      ;
10 ;
11 def handler(self):
12     increment();
13     done[self] = True;
14 ;
15 def process(self):
16     trap handler(self);
17     increment();
18     await done[self];
19 ;
20 def main(self):
21     await all(done);
22     assert count == 4, count;
23 ;
24 count = 0;
25 countlock = Lock();
26 done = [ False, False ];
27 spawn process(0);
28 spawn process(1);
29 spawn main();
```

first disable interrupts



Interrupt-safe *AND* Thread-safe?

```
1  import synch;
2
3  def increment():
4      let prior = setintlevel(True);
5          lock(?countlock);
6          count += 1;
7          unlock(?countlock);
8          setintlevel(prior);
9      ;
10 ;
11 def handler(self):
12     increment();
13     done[self] = True;
14 ;
15 def process(self):
16     trap handler(self);
17     increment();
18     await done[self];
19 ;
20 def main(self):
21     await all(done);
22     assert count == 4, count;
23 ;
24 count = 0;
25 countlock = Lock();
26 done = [ False, False ];
27 spawn process(0);
28 spawn process(1);
29 spawn main();
```

first disable interrupts

then acquire a lock

Interrupt-safe *AND* Thread-safe?

```
1  import synch;
2
3  def increment():
4      let prior = setintlevel(True);
5          lock(?countlock);
6          count += 1;
7          unlock(?countlock);
8          setintlevel(prior);
9      ;
10 ;
11 def handler(self):
12     increment();
13     done[self] = True;
14 ;
15 def process(self):
16     trap handler(self);
17     increment();
18     await done[self];
19 ;
20 def main(self):
21     await all(done);
22     assert count == 4, count;
23 ;
24 count = 0;
25 countlock = Lock();
26 done = [ False, False ];
27 spawn process(0);
28 spawn process(1);
29 spawn main();
```

first disable interrupts

then acquire a lock

why 4?

Signals (virtualized interrupts) in Posix / C

Allow applications to behave like operating systems.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal (e.g. ctrl-z from keyboard)

Sending a Signal

Kernel delivers a signal to a destination process

For one of the following reasons:

- Kernel detected a system event (e.g., div-by-zero (SIGFPE) or termination of a child (SIGCHLD))
- A process invoked the **kill system call** requesting kernel to send signal to a process

Receiving a Signal

A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.

Three possible ways to react:

1. Ignore the signal (do nothing)
2. Terminate process (+ optional core dump)
3. Catch the signal by executing a user-level function called signal handler
 - Like a hardware exception handler being called in response to an asynchronous interrupt

Signal Example

```
#include <stdio.h>
#include <signal.h>
#include <string.h>

int done = 0;

void handler(int signum){
    done = 1;
}

int main(){
    struct sigaction sa;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handler;
    sa.sa_flags = SA_RESTART; /* Restart syscalls if interrupted */
    sigaction(SIGINT, &sa, NULL);

    while (!done)
        ;
    printf("DONE\n");
    return 0;
}
```

Warning: very few C functions are interrupt-safe

- pure system calls are interrupt-safe
 - e.g. `read()`, `write()`, etc.
- functions that do not use global data are interrupt-safe
 - e.g. `strlen()`, `strcpy()`, etc.
- `malloc()` and `free()` are *not* interrupt-safe
- `printf()` is *not* interrupt-safe
- *However, all these functions are thread-safe*