On Abstraction

- Cornerstone of system design
  - managing complexity
- Abstraction
  - Interface: methods + behaviors
    - Queue: Queue(), put(), get()
    - R/W lock: RW(), rAcquire, rRelease, wAcquire, wRelease
    - pool: Pool(), enter(level), exit(level)
  - Behaviors under concurrency??
    - typically want same as if all operations are atomic
    - (but some abstractions might give weaker guarantees in exchange for improved performance)
  - Black box testing:
    - can’t look “under the covers”

On Abstraction, cont'd

- What is a good abstraction?
  - Justice Potter Stewart: know it when I see it
  - *Hide implementation details*
    - abstraction can be implemented in many different ways
      - we saw two different implementations of R/W locks already
      - there are many more
    - helps with maintainability
      - abstraction $\neq$ encapsulation
      - abstraction $\neq$ object-orientation
  - *Cohesion*: focused on a single task
    - no unrelated methods
  - *Separate policy and mechanism*
    - when possible
- What abstractions are good?
  - clock, atm machine, queue, lock, R/W lock, schoolpool, process, thread, virtual memory, file, …

Black Box Testing

- Not allowed to look under the covers
  - can't use *rw*->nreaders, etc.
- Only allowed to invoke the interface methods and observe behaviors
- Your job: try to find bad behaviors
  - need to maintain your own state
  - how would you test a clock? An ATM machine?
  - use your creativity
- In general testing cannot ensure correctness
  - only a correctness proof can
  - testing may or may not expose a bug
  - assertions/invariants help expose bugs
  - model checking helps expose bugs
  - some bugs are harder to find than others

Actors, Barriers, Interrupts

CS 4410
Operating Systems

[Robbert van Renesse]

Actor Model

- An *actor* is a type of 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
- *Actors and message queues are abstractions*

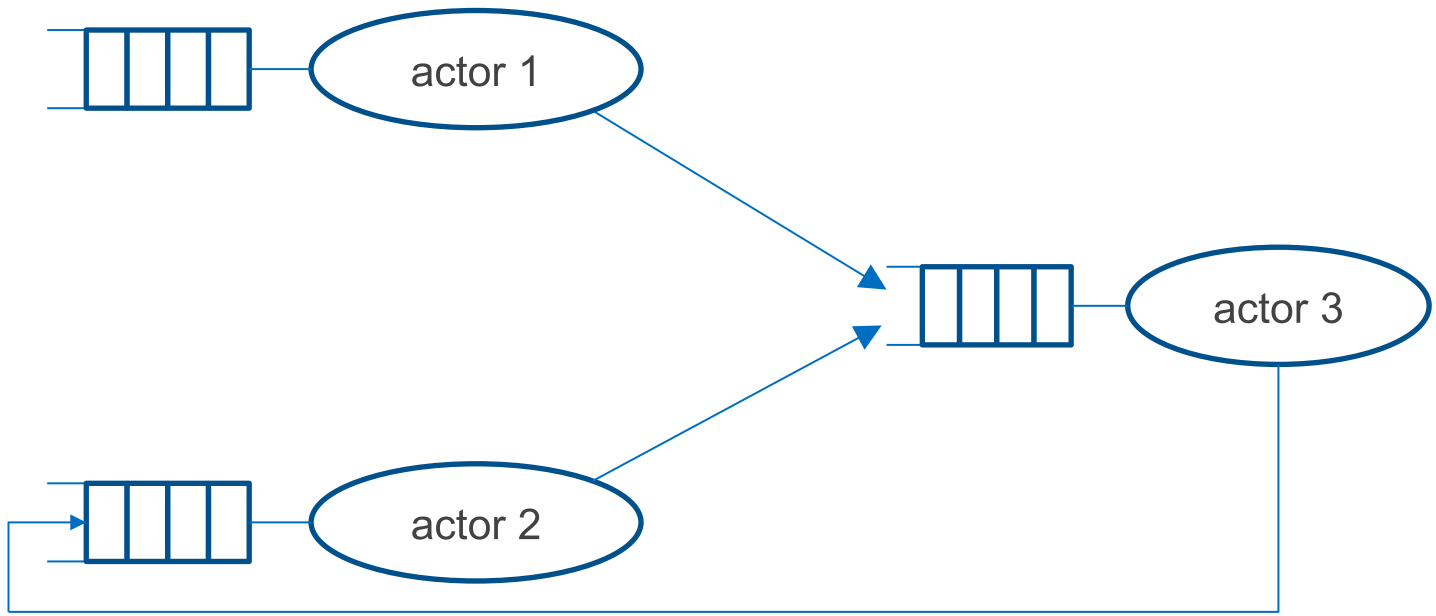
Reminiscent of event-based programming, but each actor has local state

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)

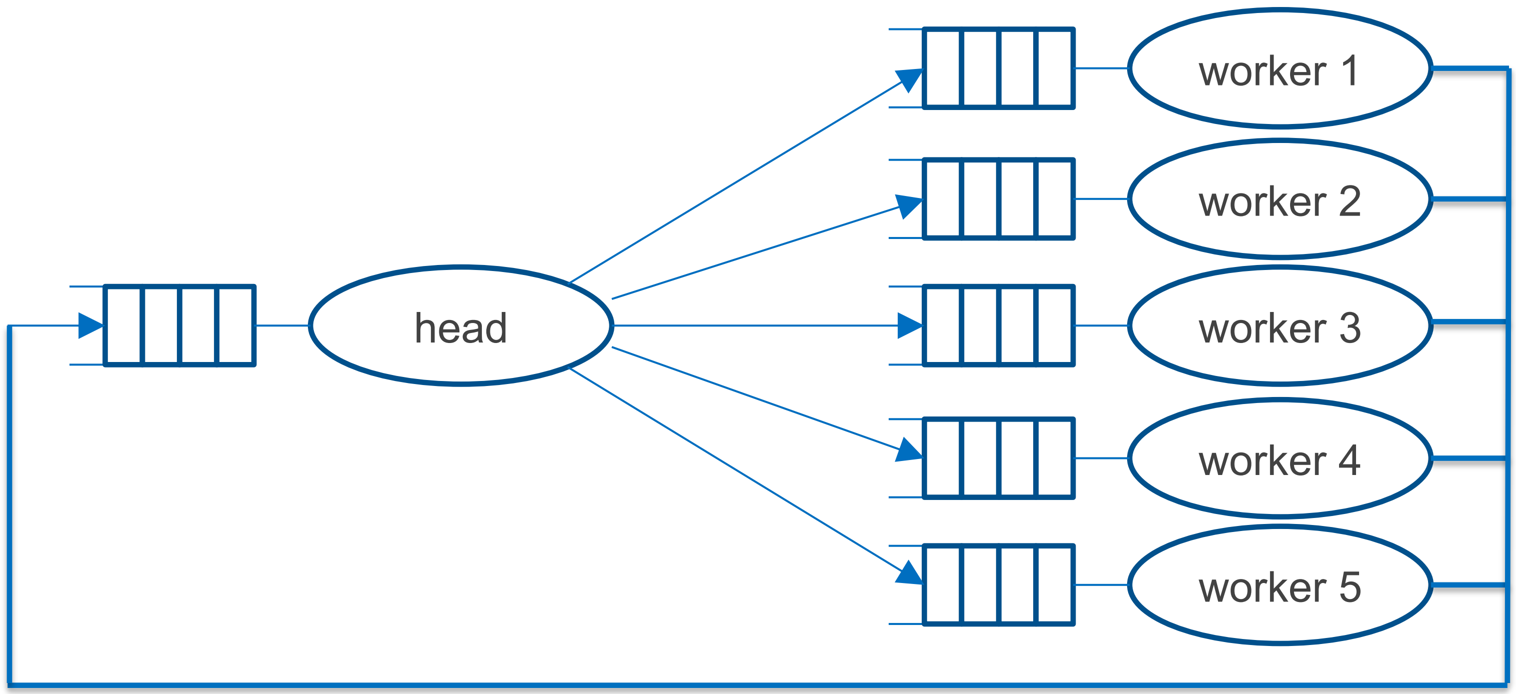


Conditional Critical Sections with Actors

- 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

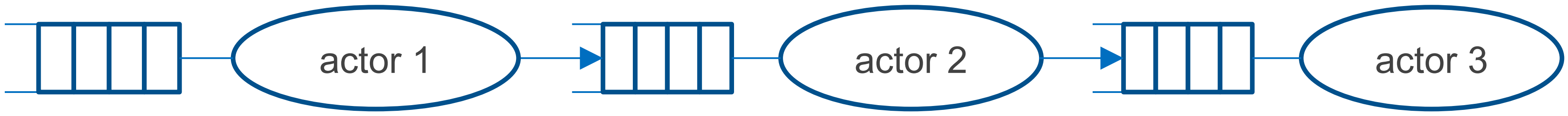
Parallel processing with Actors

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



automatic flow control (when actors run at different rates)

- in both “directions” with bounded buffer queues

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 UNIVERSITY FOUNDED A.D. 1865

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, …

Using barriers

```
import barrier

const NROUNDS = 3
const NTHREADS = 3

barr = barrier.Barrier(NTHREADS)
round = [None,] * NTHREADS

def thread(self):
    for r in {0..NROUNDS–1}:
        barrier.enter(?barr)
        round[self] = r
        assert { x for x in round where x != None } == { r }
        round[self] = None
        barrier.exit(?barr)

for i in {0..NTHREADS–1}:
    spawn thread(i)
```

Using barriers

```
import barrier

const NROUNDS = 3
const NTHREADS = 3

barr = barrier.Barrier(NTHREADS)
round = [None,] * NTHREADS

def thread(self):
    for r in {0..NROUNDS–1}:
        barrier.enter(?barr)
        round[self] = r
        assert { x for x in round where x != None } == { r }
        round[self] = None
        barrier.exit(?barr)

for i in {0..NTHREADS–1}:
    spawn thread(i)
```

same methods as Pool, but different behavior!

Think of rollercoaster car

- Fixed #seats
  1. Can’t go until all seats filled
  2. Must be emptied before next run

Implementation: State maintenance

```
def Barrier(limit):
    result = {
            .limit: limit,
            .mutex: Lock(),
            .empty: Condition(), .full: Condition(),
            .entered: 0, .left: limit
        }
```

State maintenance

```
def Barrier(limit):
    result = {
            .limit: limit,
            .mutex: Lock(),
            .empty: Condition(), .full: Condition(),
            .entered: 0, .left: limit
        }
```

#seats in car

#people who have entered car

#people who have left since last run

Entering the car

```
def enter(b):
    acquire(?b→mutex)
    while b→entered == b→limit:    # wait for car to empty out
        wait(?b→empty, ?b→mutex)
    b→entered += 1
    if b→entered != b→limit:       # wait for car to fill up
        while b→entered < b→limit:
            wait(?b→full, ?b→mutex)
    else:
        b→left = 0
        notifyAll(?b→full)         # car is full and ready to go
    release(?b→mutex)
```

Leaving the car

```
def exit(b):
    acquire(?b→mutex)
    b→left += 1
    if b→left == b→limit:          # car is empty
        b→entered = 0
        notifyAll(?b→empty)
    release(?b→mutex)
```

This is very subtle stuff!

Note:

- *left* not reset until the car has filled up
- *entered* not reset until the car has emptied out

Leaving the car

```
def exit(b):
    acquire(?b→mutex)
    b→left += 1
    if b→left == b→limit:        # car is empty
        b→entered = 0
        notifyAll(?b→empty)
    release(?b→mutex)
```

Interrupt Handling

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
  - So the user space context is exactly restored to the state it was in before the interrupt

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

```
count = 0
done = False

def handler():
    count += 1
    done = True

def main():
    trap handler()
    await done
    assert count == 1

spawn main()
```

invoke handler() at some future time

Within the same process!
(trap $\neq$ spawn)

But what now?

```
count = 0
done = False

def handler():
    count += 1
    done = True

def main():
    trap handler()
    count += 1
    await done
    assert count == 2

spawn main()
```

But what now?

```
count = 0
done = False

def handler():
    count += 1
    done = True

def main():
    trap handler()
    count += 1
    await done
    assert count == 2
```

```
#states 20
Safety Violation
T0: __init__() [0-7,36-40] { count: 0, done: False }
T1: main() [17-24,interrupt,8-15,24-32] { count: 1, done: True }
Harmony assertion failed
```

Locks to the rescue?

```
countlock = Lock()
count = 0
done = False

def handler():
    acquire(?countlock)
    count += 1
    release(?countlock)
    done = True

def main():
    trap handler()
    acquire(?countlock)
    count += 1
    release(?countlock)
    await done
    assert count == 2

spawn main()
```

Locks to the rescue?

```
countlock = Lock()
count = 0
done = False

def handler():
    acquire(?countlock)
    count += 1
    release(?countlock)
    done = True

def main():
    trap handler()
    acquire(?countlock)
    count += 1
    release(?countlock)
```

```
#states 27
27 components, 3 bad states
Non-terminating state
T0: __init__() [0-5,335-337,787-791,539-542,534-537,543,544,792-797,842-846] { b
ag: (), count: 0, countlock: False, done: False, list: (), synch: () }
T1: main() [815-821,552-555,507-516,556,557,822-825] { bag: (), count: 0, countl
ock: True, done: False, list: (), synch: () }
```

Enabling/disabling interrupts

```
count = 0
done = False

def handler():
    count += 1
    done = True

def main():
    trap handler()
    setintlevel(True)
    count += 1
    setintlevel(False)
    await done
    assert count == 2

spawn main()
```

Enabling/disabling interrupts

```
count = 0
done = False

def handler():
    count += 1
    done = True

def main():
    trap handler()
    setintlevel(True)
    count += 1
    setintlevel(False)
    await done
    assert count == 2

spawn main()
```

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

Interrupt-Safe Methods

```
count = 0
done = False

def increment():
    let prior = setintlevel(True):
        count += 1
        setintlevel(prior)

def handler():
    increment()
    done = True

def main():
    trap handler()
    increment()
    await done
    assert count == 2
```

disable interrupts

restore old level

Interrupt-safe *AND* Thread-safe?

```
sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await all(done)
    assert count == 4, count

spawn thread(0)
spawn thread(1)
```

first disable interrupts

Interrupt-safe *AND* Thread-safe?

```
sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await all(done)
    assert count == 4, count

spawn thread(0)
spawn thread(1)
```

first disable interrupts

then acquire a lock

Interrupt-safe *AND* Thread-safe?

```
sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await all(done)
    assert count == 4, count

spawn thread(0)
spawn thread(1)
```

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)

[UNIX]

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

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*