On Abstraction

- Cornerstone of system design
  - managing complexity
- Abstraction
  - Interface: methods + behaviors
    - Queue: Queue(), put(), get()
    - Stack: Stack(), push(), pop(), post()
    - R/W lock: RW(), rAcquire, rRelease, wAcquire, wRelease
  - Behaviors under concurrency??
    - typically want same as if all operations happen atomically sometime between invocation and completion
    - (but some abstractions might give weaker guarantees in exchange for improved performance)

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 four different implementations of R/W locks already
      - there are many more
    - helps with maintainability
      - encapsulation
  - *Cohesion*: focused on a single task
    - no unrelated methods
  - *Separate policy and mechanism*
    - when possible
- What abstractions are good?
  - queue, stack, lock, R/W lock, 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
  - compare against a *specification*
  - how would you test a clock? An ATM machine?
- In general testing cannot ensure correctness
  - only a correctness proof can
  - testing may or may not expose a bug
  - model checking helps expose bugs

Actors, Barriers, Interrupts

Harmony Book Chapters: 20, 21, 22

CS 4410
Operating Systems

[Robbert van Renesse]

Actor Model

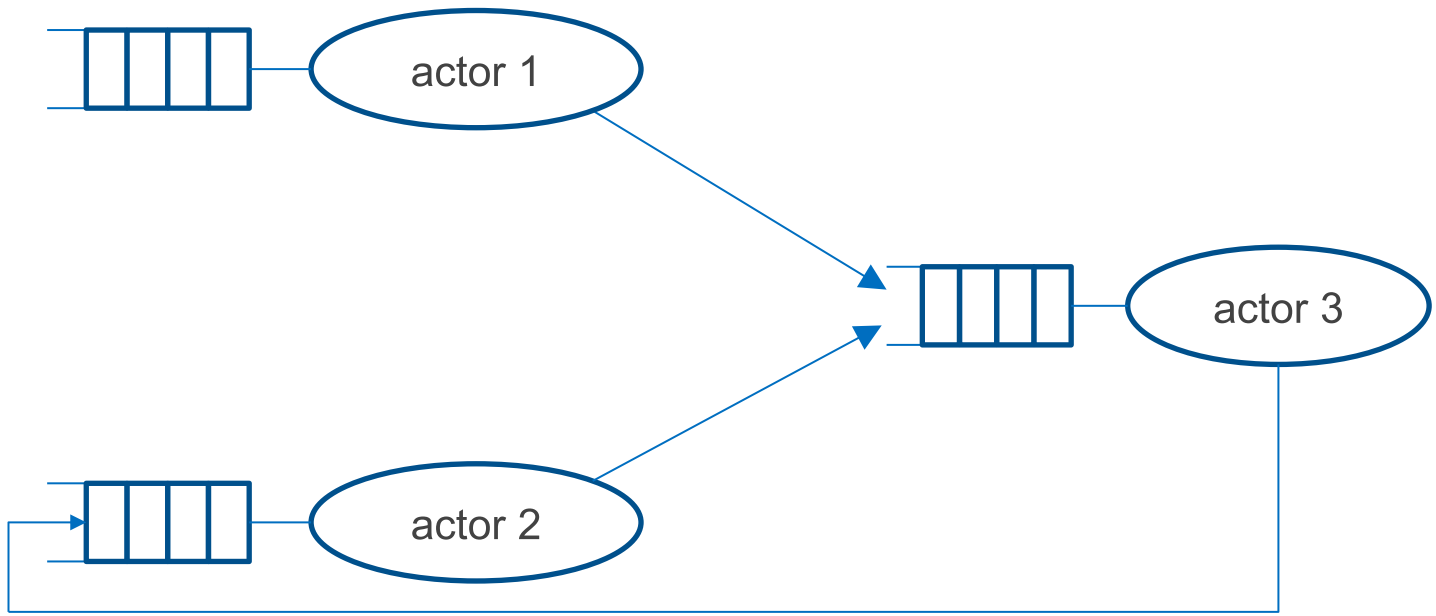
- An *actor* is a type of process
- Each actor has an incoming *message queue*
- No other shared state
- Actors communicate by “message passing”
  - placing messages on message queues
- Supports modular concurrent programs
- *Actors and message queues are abstractions*

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)

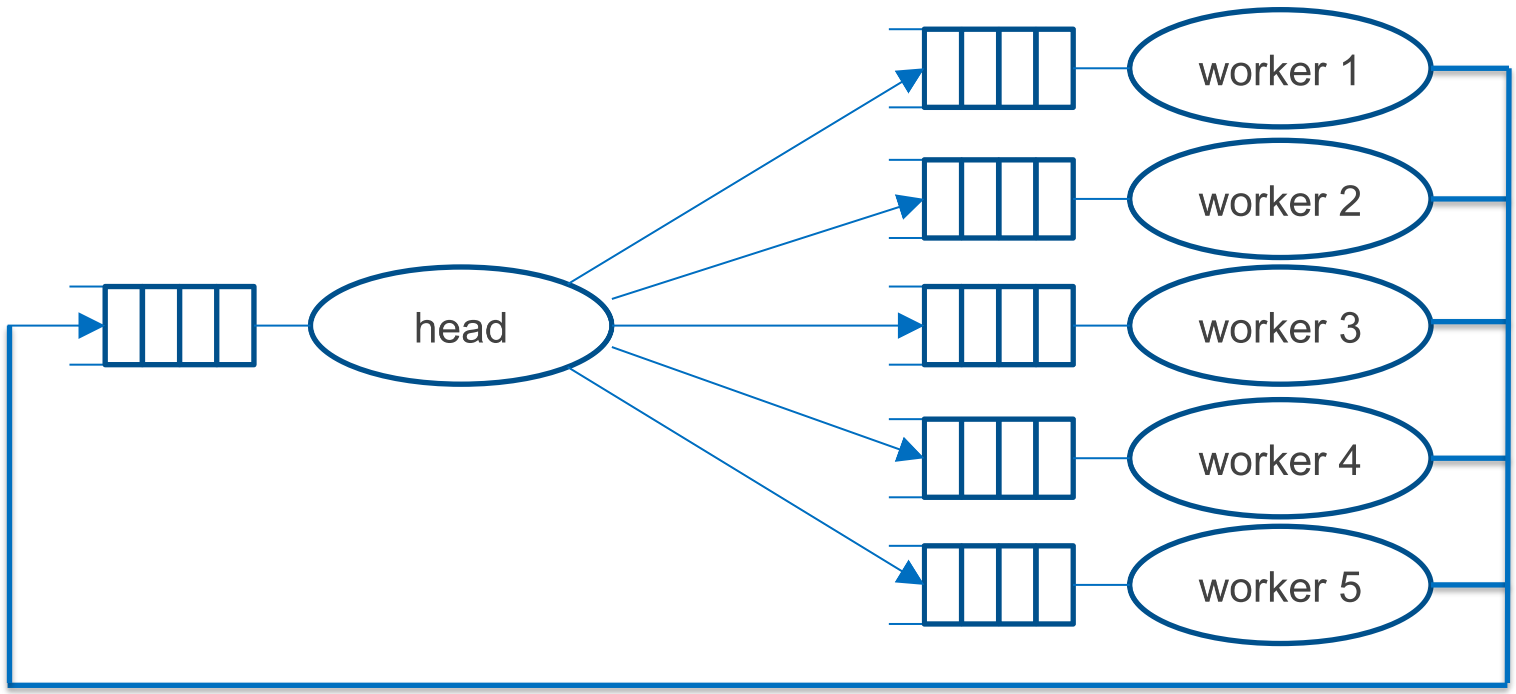


Conditional Critical Sections with Actors

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

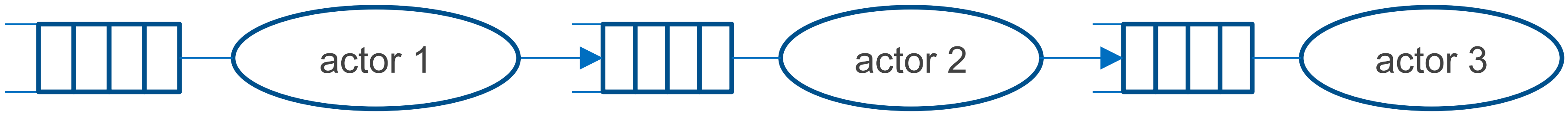
Parallel processing with Actors

- 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)

- 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
- Sending/receiving messages just to access a data structure leads to significant extra code

Barrier Synchronization

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, model checking…

Barrier abstraction

- Barrier(N): barrier for N threads
- bwait(): wait for everybody to catch up

Test program for barriers

```
import barrier

const NTHREADS = 3
const NROUNDS = 4

round = [0,] * NTHREADS
invariant (max(round) - min(round)) <= 1

barr = barrier.Barrier(NTHREADS)

def thread(self):
    for r in {0..NROUNDS-1}:
        barrier.bwait(?barr)
        round[self] += 1

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

Barrier Implementation

```
from synch import *

def Barrier(required) returns barrier:
    barrier = {
        .mutex: Lock(), .cond: Condition(),
        .required: required, .left: required, .cycle: 0
    }

def bwait(b):
    acquire(?b->mutex)
    b->left -= 1
    if b->left == 0:
        b->cycle = (b->cycle + 1) % 2
        b->left = b->required
        notifyAll(?b->cond)
    else:
        let cycle = b->cycle:
            while b->cycle == cycle:
                wait(?b->cond, ?b->mutex)
    release(?b->mutex)
```

State:

- Lock/Condition
- required: #threads
- left: #threads that have not reached the barrier
- cycle: allows re-use of barrier. Incremented each round

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

finally count == 1

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

def main():
    trap handler()
    await done

spawn main()
```

check count == 1 in the final state

invoke handler() at some future time

Within the same thread!
(trap ≠ spawn)

But what now?

```
count = 0
done = False

finally count == 2

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

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

spawn main()
```

But what now?

```
1   count = 0
2   done = False
3
4   finally count == 2
5
6   def handler():
7       count += 1
8       done = True
9
10  def main():
11      trap handler()
12      count += 1
13      await done
14
15  spawn main()
```

Summary: something went wrong in an execution

- Schedule thread T0: **init()**
  - Line 1: Initialize count to 0
  - Line 2: Initialize done to False
  - **Thread terminated**
- Schedule thread T1: main()
  - Line 12: Interrupted: jump to interrupt handler first
  - Line 12: Interrupts disabled
  - Line 7: Set count to 1 (was 0)
  - Line 8: Set done to True (was False)
  - Line 6: Interrupts enabled
  - Line 12: Set count to 1 (unchanged)
  - **Thread terminated**
- Schedule thread T2: finally()
  - Line 4: Harmony assertion failed

Locks to the rescue?

```
from synch import Lock, acquire, release

countlock = Lock()
count = 0
done = False

finally count == 2

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

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

spawn main()
```

Locks to the rescue?

```
1  from synch import Lock, acq
2
3  countlock = Lock()
4  count = 0
5  done = False
6
7  finally count == 2
8
9  def handler():
10     acquire(?countlock)
11     count += 1
12     release(?countlock)
13     done = True
14
15 def main():
16     trap handler()
17     acquire(?countlock)
18     count += 1
19     release(?countlock)
20     await done
21
22 spawn main()
```

Summary: some execution cannot terminate

- Schedule thread T0: **init**()
  - Line 3: Initialize countlock to False
  - Line 4: Initialize count to 0
  - Line 5: Initialize done to False
- Schedule thread T1: main()
  - Line synch/36: Set countlock to True (was False)
  - Line 18: Set count to 1 (was 0)
  - Line synch/39: Interrupted: jump to interrupt handler first
  - Line synch/39: Interrupts disabled
  - Preempted in main() --> release(?countlock) --> handler() --> acquire(?countlock) about to execute atomic section in line synch/35

Final state (all threads have terminated or are blocked):

- Threads:
  - T1: (blocked interrupts-disabled) main() --> release(?countlock) --> handler() --> acquire(?countlock)
    - about to execute atomic section in line synch/35

Enabling/disabling interrupts

```
count = 0
done = False

finally count == 2

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

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

spawn main()
```

disable interrupts

enable interrupts

Interrupt-Safe Methods

```
count = 0
done = False

finally count == 2

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

def handler():
    increment()
    done = True

def main():
    trap handler()
    increment()
    await done

spawn main()
```

disable interrupts

restore old interrupt level

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

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

finally count == 4

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 done[self]

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

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

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

finally count == 4

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 done[self]

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

wait for own interrupt

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

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

finally count == 4

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 done[self]

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

first disable interrupts

wait for own interrupt

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

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

finally count == 4

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 done[self]

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

first disable interrupts

then acquire a lock

wait for own interrupt

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

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

finally count == 4

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 done[self]

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

why 4?

first disable interrupts

then acquire a lock

wait for own interrupt

Signals (virtualized interrupts) in Posix / C

Applications can have interrupts / exceptions too!

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*

On HW5

- You are to implement a “deque” as a bounded buffer
- For example, using 3 slots in the buffer:

operation	**deque**
put_left(A)	[A]
put_right(B)	[AB]
get_right() → B	[A]
put_left(C)	[CA]
put_left(D)	[DCA]
get_right() → A	[DC]

On HW5

- You are to implement a “deque” as a bounded buffer
- For example, using 3 slots in the buffer:

operation	**deque**	**slot 1**	**slot 2**	**slot 3**
put_left(A)	[A]	A		
put_right(B)	[AB]	A	B	
get_right() → B	[A]	A		
put_left(C)	[CA]	A		C
put_left(D)	[DCA]	A	D	C
get_right() → A	[DC]		D	C

green is left-most

Add concurrency

- deque should be thread-safe → add lock
- operations should be blocking → add condition variables
  - what are the waiting conditions?
- don’t ”over-notify”
  - but better be safe than sorry