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 \texttt{rw->nreaders}, etc.
• Only allowed to invoke the interface methods and observe behaviors
• Your job: try to find bad behaviors
  • compare against a \textit{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
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
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
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

```python
from synch import *

def Barrier(required):
    result = {
        .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

```python
3 count = 0
4 done = False

6 def handler():
7     count += 1
8     done = True

10 def main():
11     trap handler()
12     await done
13     assert count == 1
14     spawn main()
```

invoke handler() at some future time

Within the same thread! (trap ≠ spawn)
But what now?

```python
3 count = 0
done = False

6 def handler():
7     count += 1
8     done = True

10 def main():
11     trap handler()
12     count += 1
13     await done
14     assert count == 2
16     spawn main()
```
But what now?

```python
3 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?

```python
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] { bag: (), count: 0, countlock: False, 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

```python
3     count = 0
4     done = False

6     def handler():
7         count += 1
8         done = True

10    def main():
11       trap handler()
12       setintlevel(True)
13       count += 1
14       setintlevel(False)
15       await done
16       assert count == 2
17
18     spawn main()
```

Interrupt-Safe Methods

```python
3  count = 0
4  done = False

5  def increment():
6      let prior = setintlevel(True):
7          count += 1
8          setintlevel(prior)

9  def handler():
10     increment()
11     done = True

12  def main():
13     trap handler()
14     increment()
15     await done
16     assert count == 2
```
Interrupt-safe AND Thread-safe?

```python
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?

```python
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?

```python
sequential t_done, i_done

count = 0
countlock = Lock()
t_done = i_done = [False, False]

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

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

def thread(self):
    trap handler(self)
    increment()
    await i_done[self]
    t_done[self] = True
    await all(t_done)
    assert count == 4

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

- **first disable interrupts**
- **then acquire a lock**
- **wait for own interrupt**
- **why 4?**
Signals (virtualized interrupts) in Posix / C

Allow applications to behave like operating systems.

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