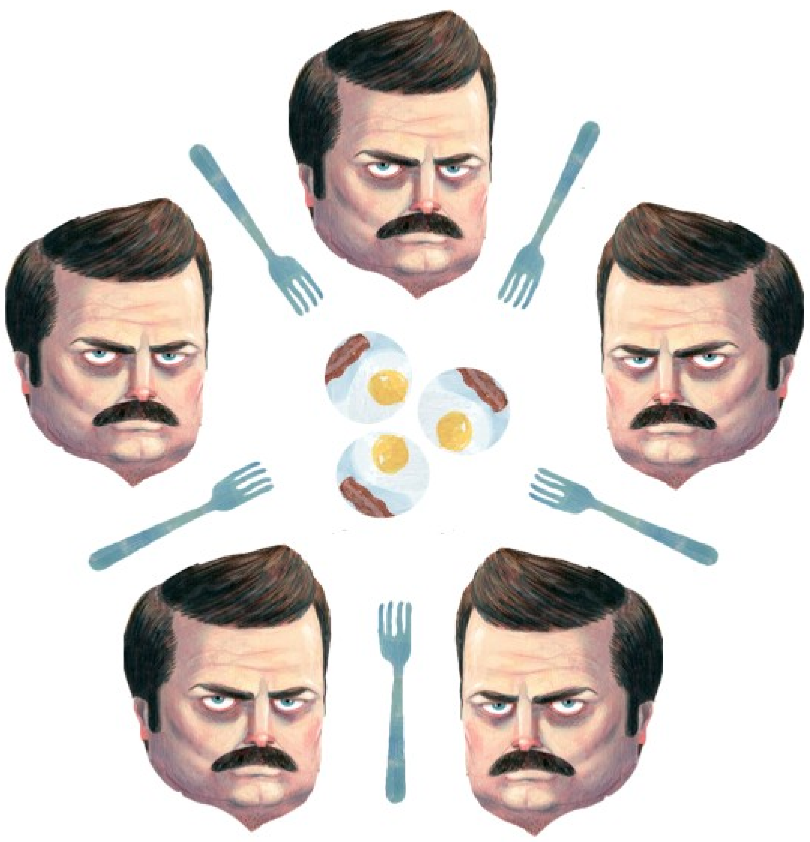
CS 430 Spring 2022

Mike Lam, Professor

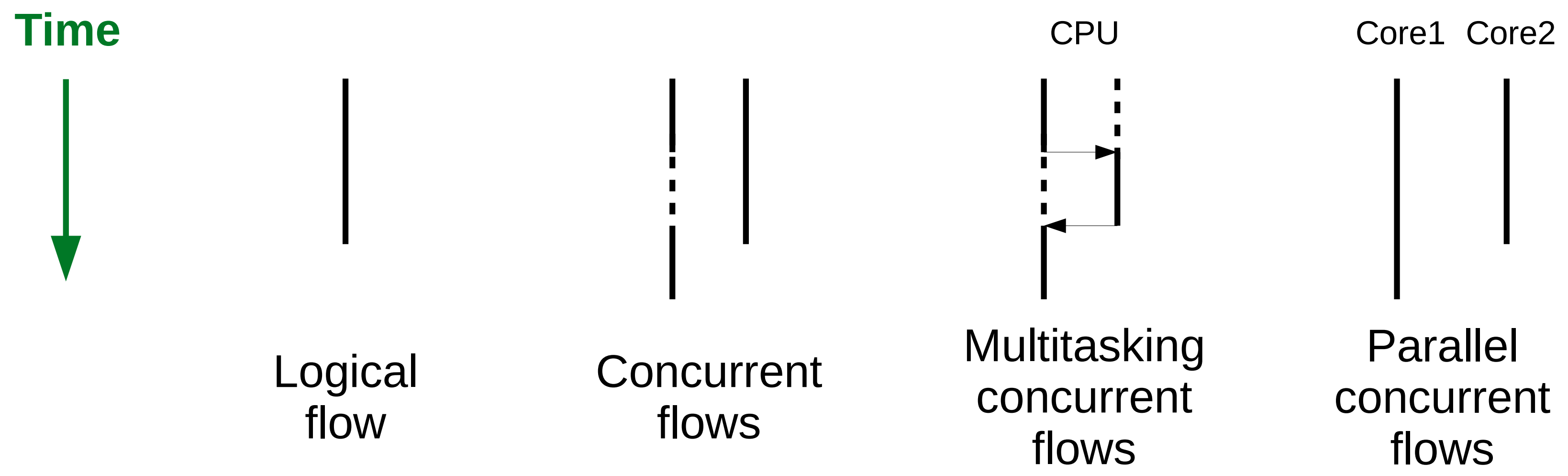


http://adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html

Concurrency and Error Handling

Concurrency (CS 261 review)

- Logical flow: sequence of executed instructions
- Concurrency: overlapping logical flows
- Multitasking: processes take turns
- Parallelism: concurrent flows on separate CPUs/cores



Concurrency

- Instruction-level concurrency
  - Mostly an architecture and compilers issue (**CS 261/456/432**)
- Statement-level concurrency
  - Often enabled by language or library features (**CS 361/470**)
- Unit (subprogram)-level concurrency
  - Sometimes enabled by language features
  - Often a distributed/parallel systems issue (**CS 361/470**)
- Program-level concurrency
  - Mostly an OS or batch scheduler issue (**CS 450/470**)

Concurrency

- Motivations?
  - It's faster!
  - Take advantage of multicore/multiprocessor machines
  - Take advantage of distributed machines
  - Faster execution even on single-core machines
  - Enables new approaches to solving problems

History of Parallelism

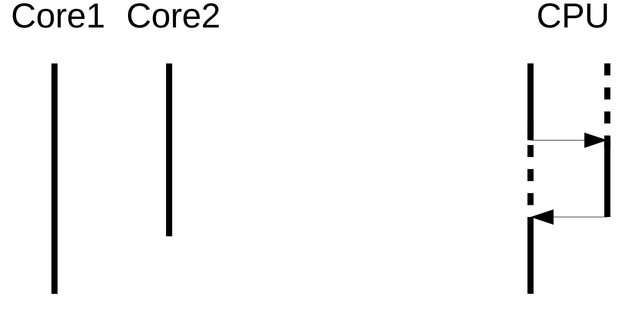
- 1950s: special-purpose I/O or graphics processors
- 1960s: multiple complete processors
- 1970s: vector processors
- 1980s: computing clusters
- 1990s-2000s: rise of multicore consumer machines and graphical processing units (GPUs)
- 2010s: hybrid CPU/GPU architectures
- Future: low-cost, low-power, many-core
  - Example: new ARM64 M1 “Apple Silicon” chips

Categories (Flynn's Taxonomy)

- Single-Instruction, Single-Data (SISD)
  - Traditional CPUs
- Single-Instruction, Multiple-Data (SIMD)
  - Vector processors
  - GPUs
  - SSE/AVX instructions on x86
- Multiple-Instruction, Multiple-Data (MIMD)
  - Multicore processors
  - Distributed computing (single-program SPMD variant)

Concepts

- Physical vs. logical concurrency
  - Is the concurrency actually happening on the hardware level, or are executions being interleaved?
  - Sometimes referred to as parallelism vs. multitasking
  - Users and language designers might not care
  - Language implementers and OS designers must care



Parallel concurrency

Multitasking concurrency

Concepts

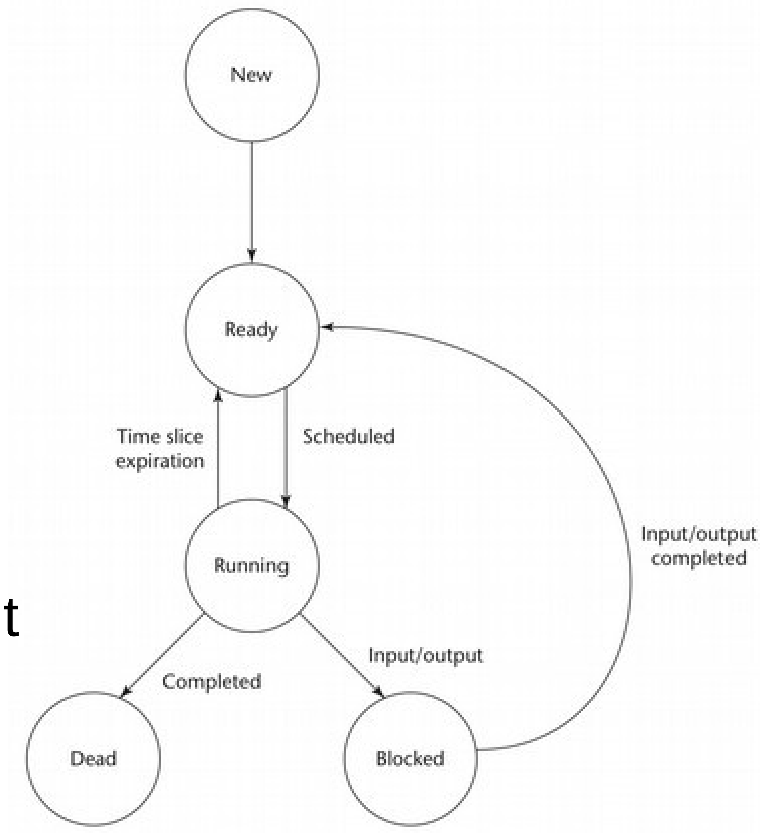
- Task/process/thread: program unit that supports concurrent execution
  - Typically, a process may contain multiple threads
  - All threads in a process share a single address space
  - Textbook: heavyweight = process, lightweight = thread
  - Some OSes support lightweight processes
  - See **CS 361**, **450**, or **470** for more details

Concepts

- Single-threaded vs. multi-threaded
  - Thread: sequence of control flow points
  - Coroutines are single threaded (quasi-concurrent)
  - Multi-threaded programs may still be executed on a single CPU via interleaving / multitasking
- Synchronous vs. asynchronous
  - Synchronous tasks must take turns and wait for each other
  - Asynchronous tasks may execute simultaneously

Scheduling

- Scheduler: a system program that manages the sharing of processors between tasks
  - Priority-based scheduling
  - Round-robin scheduling
  - Real-time scheduling
- Task states
  - **New**: created but not yet begun
  - **Ready**: not executing, but may be started
    - Often stored in a ready queue
  - **Running**: currently executing
  - **Blocked**: running, but waiting on an event
  - **Dead**: no longer active



Concepts

- Liveness: a program executes to completion
- Deadlock: loss of liveness due to mutual waiting
  - E.g., dining philosophers!
- Race condition: concurrency outcome depends on interleaving order
  - Example: Two concurrent executions of `bump()`

Thread A

```
def bump(x)
  tmp = $counter    (A.1)
  tmp += x          (A.2)
  $counter = tmp    (A.3)
end
```

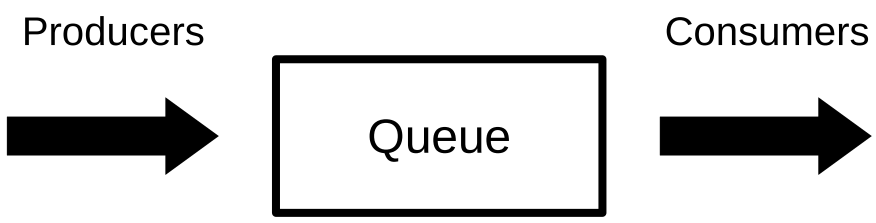
Thread B

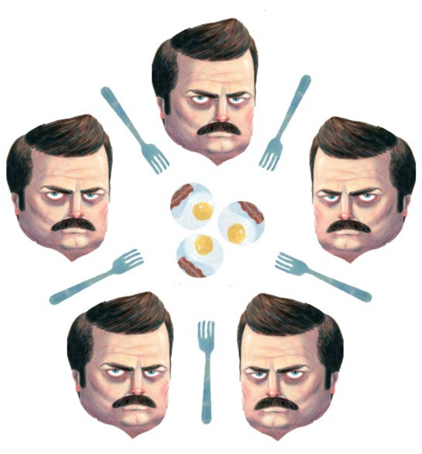
```
def bump(x)
  tmp = $counter    (B.1)
  tmp += x          (B.2)
  $counter = tmp    (B.3)
end
```

OK:	BAD:
A.1	A.1
A.2	B.1
A.3	A.2
B.1	B.2
B.2	A.3
B.3	B.3

Concepts

- Synchronization: mechanism that controls task ordering
  - Cooperation: ordering based on inter-task dependencies
    - E.g., Task A is waiting on task B to finish an activity
    - Common example: producer/consumer problem
  - Competition: ordering based on resource contention
    - E.g., Task A and Task B both need access to a resource
    - Common example: dining philosopher problem





Synchronization

- Semaphore: guarding mechanism (1965)
  - Integer (n = “empty slots”) and a task queue
  - **Produce** (P / “wait”)
    - decrement n
    - if (n < 0): enqueue current process and block
  - **Consume** (V / “signal”)
    - increment n
    - if (n >= 0): dequeue and unblock a process
  - Binary semaphore: single “slot” (mutex)
  - Issue: burden of correct use falls on the programmer

Synchronization

- Monitor: encapsulation mechanism (1974)
  - Abstract data types for concurrency
  - Handles locking and corresponding thread queue
  - Shifts responsibility to language implementer and runtime system designer
  - Generally considered safer
- Message passing: communication model (1978)
  - Fairness in communication
  - Synchronous vs. asynchronous
  - Can be difficult to program and expensive
  - Necessary in distributed computing

High-Performance Fortran

- Motivation: higher abstractions for parallelism
  - Predefined data distributions and parallel loops
  - Optional **directives** for parallelism specified in comments
- Development based on Fortran 90
  - Proposed 1991 w/ intense design efforts in early 1990s
  - Wide variety of influences on the design committee
  - Standardized in 1993 and presented at Supercomputing '93

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
          + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
```

Listing 8: Simple relaxation loop in HPF.

For the full story, see "The Rise and Fall of High Performance Fortran: An Historical Object Lesson"
`http://dl.acm.org/citation.cfm?id=1238851`

High-Performance Fortran

- Issues
  - Immature compilers and no reference implementation
  - Poor support for non-standard data distributions
  - Poor code performance; difficult to optimize and tune
  - Slow uptake among the HPC community
- Legacy
  - Effort in 1995-1996 to fix problems with HPF 2.0 standard
  - Eventually dropped in popularity and was largely abandoned
  - Some ideas still had a profound influence on later efforts

OpenMP

- Modern statement-level parallelism
- C or C++ compiler extension
- Uses "`pragma omp`" preprocessor directives
- Compiler adds threading code automatically
- Responsibility for correctness remains on programmer!

```
void add_vectors(int nvectors, int length,
                 double *vectors[], double sum[])
{
    #pragma omp parallel for
    for (int i = 0; i < length; ++i) {
        sum[i] = 0.0;
        for (int v = 0; v < nvectors; ++v) {
            sum[i] += vectors[v][i];
        }
    }
}
```

Language Support

- C/C++/Fortran
  - Pthreads, OpenMP, MPI
- Java
  - Threads, `synchronized` keyword and `wait`/`notify`
- Haskell
  - `Control.Parallel` and `Control.Concurrent`
- High-Performance Fortran (HPF)
  - `DISTRIBUTE` and `FORALL`
- Go
  - Goroutines, channels, and mutexes
- Chapel
  - `coforall`, `cobegin`, and domains

Exceptional control flow

- Concurrency is often implemented using exceptional control flow
  - Variants: interrupt, trap, fault, abort
  - (remember this from **CS 261**?)

- Related question: how to handle errors in high-level programming languages?
  - (how did you do it in 159/240? what about 261?)

Approaches

- Do nothing (worst possible approach!)
  - No indication that anything has gone wrong
  - Could lead to "silent propagation" of errors
- Terminate the program (e.g., abort or segfault)
  - I.e. delegate error handling to the operating system
  - Also rather drastic, but at least it provides some kind of notification (OS-dependent)
  - No opportunity to correct problems

```
int div(int a, int b) {
    return a / b;
}
```

```
int div(int a, int b) {
    if (b == 0) exit(-1);
    return a / b;
}
```

Approaches

- Pass around error handlers
  - Extra function parameters (and runtime overhead)
  - Confusing and difficult to reason about
  - What if you pass the wrong error handler?
- Handle all errors at their source
  - Error handling often depends on current context
  - Lots of (possibly duplicate) error handling code

```
int div(int a, int b, void (handler)(char*)) {
    if (b == 0) handler(“Division by zero!”);
    return a / b;
}
```

```
if (y != 0) {
    r = div(x, y);
} else {
    printf(“Division by zero!”);
    exit(-1);
}
```

Approaches

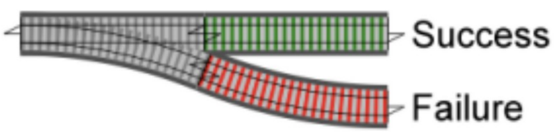
- Return an error value (in same variable)
  - Error value must come from variable domain
  - Blurs the line between program logic and program data
  - Burden shifts to callers, who must test for error value
- Return an error value (in separate variable)
  - Cleaner (separation between logic and data)
  - Burden is still on the caller to remember to test for errors

```
int div(int a, int b) {
    if (b == 0) return 0;
    return a / b;
}
```

```
int div(int a, int b, bool *err) {
    if (b == 0) {
        *err = true;
        return 0;
    }
    return a / b;
}
```

Aside: Monad design pattern

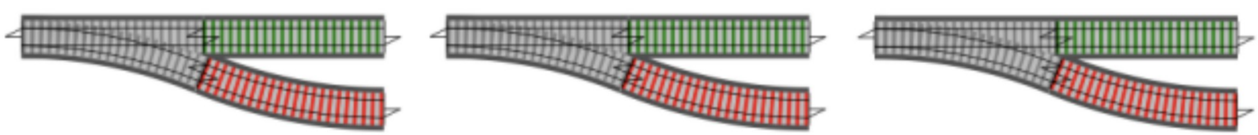
- Error handling in functional languages requires tracking of state
  - E.g., whether or not an error has occurred
- Example of a Monad pattern: functions return an Either value to propagate errors
  - Success <val> or Failure <err> (Right and Left in Haskell)
  - This is a variant of the “return an error value in separate variable” idea
  - Generic function composition via a bind operation (>>= in Haskell)

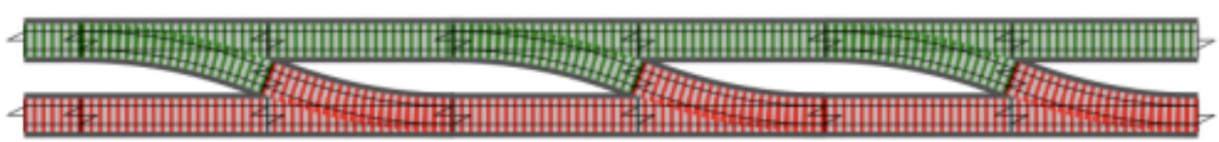


(+ value)

(+ error)

```
instance Monad (Either e) where
    Left  err >>= _  = Left err
    Right val >>= f  = f val
```





https://fsharpforfunandprofit.com/posts/recipe-part2/

Exception Handling

- Exception: unusual event (possibly erroneous) that requires special handling
- Exception handler: code unit that processes the special handling for an exception
- An exception is raised when the unusual event is detected, and is caught when the exception handler is triggered
- This framework is called formal exception handling
  - First introduced in PL/I (1976)

```
try {
    do_something_dangerous();
} catch (DangerousError e) {
    gracefully_handle(e);
}
```

```
do_something_dangerous() {
    …
    if (bad_thing_happened) {
        throw new DangerousError();
    }
    …
}
```

Benefits of Formal Exceptions

- Less program complexity and clutter; increased readability
- Standardized handling mechanisms
- Increased programmer awareness
- Decouples exception handling from program logic
- Handler re-use via exception propagation
- More secure due to compiler analysis

Design Issues

- How and where are exception handlers specified?
- What is the scope of exception handlers?
  - What information (if any) is available about the error?
- Are there any built-in exceptions? If so, what are they?
- Can programmers define new exceptions?
- How is an exception bound to a handler at runtime?
- Where does execution resume (if at all) after an exception handler finishes?

Binding and Continuation

- When an exception is thrown (binding)
  - Look for matching handler in local scope
    - Could be an "else" handler
  - If no handler is found, continue through ancestors
    - Usually via dynamic scoping
  - If no handler is found, abort the program
- When a handler finishes (continuation)
  - If the handler threw another error, handle that
    - First execute any "finally" clause if present
  - Continue execution after the handler
    - First execute any "finally" clause if present
  - Changes made by the error handler are visible

Language Debate

- Are formal exceptions any different from GOTO statements? If not, are they just as dangerous? If so, how are they different?

```
void get_N()
{
  n = compute_value();
  if (n > LIMIT) {
    goto exceed_limit_error;
  }
  return n;

exceed_limit_error:
  printf("Value exceeds limit!\n");
  exit(EXIT_FAILURE);
}
```

C version (w/ GOTO)

```
void get_N()
{
  try {
    n = compute_value();
    if (n > LIMIT) {
      throw new ExceedLimitException;
    }
  } catch (ExceedLimitException e) {
    System.out.println("Value exceeds limit!");
    System.exit(-1);
  }
  return n;
}
```

Java version (w/ exceptions)

Language Debate

- Are formal exceptions any different from GOTO statements? If not, are they just as dangerous? If so, how are they different?
  - Basic difference: formal exceptions are more *structured*
    - More rules and restrictions governing their uses
    - Language facilities provide (mostly) safe usage
  - Care should be taken to limit their complexity
    - Main issue: proximity of detection and handling
    - Avoid “spaghetti code” (hard-to-trace control flows)

Event Handling

- Similarity between error handling and event handling
  - Asynchronous events that must be handled by the program
- Primary difference: events are "normal", errors are "unusual"
  - Events come from users; errors come from elsewhere in the code or originate in hardware
- Another difference: events are often handled in a separate thread
  - Keeps the program feeling "responsive"

Event Loops

- Event loop: code that explicitly receives and handles events
- Traditional form:

```
while(GetMessage(&Msg) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

- Often run in its own thread
- Requires explicit dispatch routine
  - Can become extremely complex and unwieldy

Aside: Observer design pattern

- Cleaner solution: Observer pattern (OOP)
  - Single event thread, implemented in language runtime
    - Dispatches events to relevant objects
  - Objects maintain a list of "observers"/"listeners"
  - Upon receiving an event, the object passes it to a designated routine in every registered observer
  - Optional improvement: anonymous functions or event handling classes
    - Very similar to lambda functions or closures!

```
JButton toggleDetailsButton = new JButton("Details On/Off");
toggleDetailsButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    showDetails = !showDetails;
    updateDisplay();
  }
});
```