Concurrent and Distributed Systems

System Architectures
- Sequential systems
  - a single thread of execution
- Concurrent systems
  - multiple threads/tasks/processes
  - may or may not be executed on multiple processors
- Distributed systems
  - multiple threads
  - multiple processors, usually geographically distributed

Concurrency
- Parallelism
  - usually supported by special purpose multiprocessors
  - massive parallelism
  - often have shared memory
  - special purpose languages—Parallel FORTRAN
- Concurrent systems
  - usually implemented in a programming language that provides constructs for synchronization and shared data (e.g., Ada, Java monitors)
  - could be implemented on a single processor or multiple processors
- Distributed systems
  - autonomous processors that do not share memory
  - usually supported by operating system commands
    - RPC, RMI, message passing, event based notification, etc.

Why use concurrency?
- To improve performance because computation can occur in parallel
  - N processes does not result in an N fold improvement
- To process information concurrently
  - Process different streams of information as each arrives
- To improve availability
  - No single point of failure
- Redundancy
- To increase flexibility
  - Loose interaction models allow different processes to be added and modified with minimal intrusion

Reading Assignment (previously assigned)
  - Sections 1-4
- For reference only

Interaction models
- shared data
  - monitors
  - transactions
- remote procedure call
- rendezvous
- message passing
  - asynchronous and synchronous
  - point-to-point, broadcast, multicast
- event-based notification/publish-subscribe
**Monitor**

1. Process i
2. Process j

**Transaction model**

- Start transaction(s)
- Decide to commit transaction or throw out results
- Transactions appear to be atomic actions—all or nothing
- Transactions are often used in databases for very short computations
  - E.g., update saving account information

**Remote procedure call**

- Call foo
  - ...
  - foo
  - return

**Rendezvous model**

- Task T1 is
  - begin
  - ...
  - T2.A;
  - accept A;
  - ...
  - end T1;

- Task T2 is
  - begin
  - ...
  - rendezvous
  - accept A;
  - ...
  - end T2;

**Non-determinism**

Select among waiting tasks non-deterministically

- select
  - when NOT_FULL accept Put(C: inchar);
  - end;
  - or
  - when NOT_EMPTY accept Get(C: out char);
  - end;
  - end select;

if both guards are true, non-deterministically select a rendezvous

**Message passing**

- Asynchronous: P1 continues without waiting for an acknowledgement
- P2 may block waiting for a message from P1
Message passing

- Synchronous: P1 waits for an acknowledgement (and perhaps data) before continuing
- P1 may block waiting for an ack
- P2 may block waiting for a message from P1

```
send (p2, info)
... ...
receive (p1, info)
... ...
```

Event based notification/Implicit invocation

- Sender does not indicate actual receivers
- Receivers register their interest in being notified about events
- Provides very loose coupling between senders and receivers

```
Event e
```

Registered for a, b, e
Registered for a, b, c
Registered for e

Point-to-Point, Multicast, Broadcast

- Rendezvous
- Remote procedure call
- Message passing
- Event based notification
- Monitors

Concurrent systems are complex

- Non-determinism means that
  - the same inputs might produce different outputs on different executions
- When reasoning about a system there are numerous alternatives to consider
  - Usually more than a human can reasonably consider
- In addition to the problems that can arise with sequential programs, have problems that are unique to concurrent systems
  - Data access problems
  - Synchronization problems

Data Access Anomalies

- Want mutual exclusion
  - shared resource (e.g., data) that should only have a single access at a time
  - e.g., don’t want two travel agents assigning the last seat on a plane
- Don’t want race conditions
  - order of execution affects results
  - undesirable nondeterminism

```
task A
x := x + 1;
write x;
```
```
task B
x := x - 1;
write x;
```
```
task A
x := x + 1;
write x;
```
```
task B
x := x - 1;
write x;
```

Interleaved Model of Execution: Examples

```
task A
x := x + 1;
write x;
```
```
task B
x := x - 1;
write x;
```

```python
if initially x = 5, then could output (6, 9), (4, 5), or (5, 5)
```
Synchronization Problems: Infinite wait anomalies

- Starvation
  - At least one task does not make progress on attaining a goal
  - May continue to execute

```
repeat
  select
    when not_ready
      clean-up;
  or
    when ready
      accept Goal;
end select;
until forever;
```

Infinite wait anomalies (continued)

- Deadlock
  - A set of tasks mutually waiting on each other; none can advance
  - Often tasks hold resources that other tasks need
  - Also called, deadly embrace

```
   dining philosophers problem
```

- Livelock
  - Execution does not come to a standstill but none of the tasks can advance

Deadlock handling has been investigated extensively

- Deadlock prevention techniques
  - Preassign all resources before beginning a task
  - If all resources are not available, then assign none
  - Less efficient use of resources since not all may be needed at once
  - Preempt a process that holds a needed resource
  - But, not all resources can be preempted

Deadlock (continued)

- Deadlock avoidance
  - Only allocate a resource if there is a deadlock free path through the system
  - Hard to determine deadlock free paths
  - Must maintain global resource information
  - Not frequently used because cost is so high

- Deadlock detection
  - Use static analysis techniques to determine if there is a potential resource allocation cycle
  - May be computationally expensive and may return spurious results

Validating concurrent systems

- Dynamic analysis
- Static analysis

Dynamic analysis

- Repeating an execution with the same test cases may produce different values
  - Execution order may depend on system load, time of day, processors selected for execution, etc.
  - Much harder to do debugging
  - Much harder to do regression testing

```
task A
  x := x + 1;
  write x
```
```
task B
  x := x - 1;
  write x
```
Dynamic analysis approaches for concurrent systems

- Monitor and replay
- Coverage criteria
- Specification based result evaluation

Monitor and replay

- Monitor execution
- Provide an execution harness so that exactly the same decisions can be made on subsequent executions
  - Try to minimize number of probes to reduce overhead
  - Monitoring and forcing decisions sometimes perturbs program behavior

Coverage criteria

- All paths taking concurrency into account
  - Explodes: Must consider each interleaving
- "All" synchronizations
  - Execute each synch statement
  - Included with all stmt coverage
  - Execute each send with at least one receive
  - Similar to all-defs
  - Execute each send and receive pair
  - Similar to all-uses
- ... Extend dependence analysis to include synchronization control dependence information

Coverage criteria

- ... Extend dependence analysis to include synchronization control dependence information

Event-based monitoring

- Insert probes and monitor event sequences
- Compare event sequence to a specification of desired behavior (spec-based testing)

Speciation-based testing

- Event-based notification makes monitoring relatively easy
  - Dispatcher must send all events that occur in the property specification to the specification checker
- Other interaction models usually require inserting probes into the tasks or into the runtime system
Notations for specifying concurrent behavior

- Quantified Regular Expressions
- Deterministic Finite-State Automata
- Temporal Logics

Kripke Structure

- Kripke Structure: A nondeterministic finite state machine whose states are labeled with atomic propositions, which are true at that state
  - may be extended with fairness constraints
- Atomic propositions: A declarative sentence that can be true or false

Kripke Structure

\[ M = (S, R, S_0, A, P) \]
- \( S \) - a finite set of states
- \( R \subseteq S \times S \) - a transition relation
  - equivalently \( R : S \Rightarrow \tau(S) \)
- \( S_0 \subseteq S \) - a set of initial states
- \( A \) - a set of atomic propositions
- \( L : S \Rightarrow \tau(A) \) - labels each state with propositions satisfied in that state

Kripke structures may represent infinite execution

Kripke structure or State Transition Graph

Infinite computational tree
Temporal Logic describes properties of Kripke structures

- Several different temporal logics
  - Computational tree logic, CTL* and CTL
  - Linear Time Logic, LTL
- Describes sequences of states (values)
  - Describes time implicitly
- Primarily used for reactive systems
  - Non-terminating computation

Temporal Logic

- Path quantifiers
  - $A$, $E$
- Temporal quantifiers
  - Assume $p, q$ are atomic propositions
  - $Gp$ - $p$ is globally true or always true
  - $Fp$ - $p$ holds sometime in the future or eventually
  - $Xp$ - $p$ holds in the next state
  - $p U q$ - $p$ holds until $q$ holds
  - $p R q$ - $q$ is released from being true if $p$ is true
- CTL operator
  - path quantifier + temporal quantifier

Examples

- $AGp$
- $EGp$
- $AFp$
- $EFp$

More Examples

- $A(pUq)$
- $E(pUq)$
- $AXp$
- $EXP$

More examples

- $p R q$ - $q$ is released from being true if $p$ is true
  (p is not required to become true)

Correctness properties

- Safety properties
  - Informal: state that something bad will never happen
  - freedom from deadlock
  - Can be finitely refuted
  - e.g., dining philosophers problem:
    *who two adjacent philosophers will eat at the same time*
- Liveness properties
  - Informal: state that something good will eventually happen
  - e.g.
    *sent messages will eventually be received*
    *termination of a program*
    *all the philosophers get to eat repeatedly often*
Static analysis
• Concurrent & distributed systems are inherently more difficult to analyze
• Alias resolution
  • in addition to array references and pointers, can have dynamically allocated tasks
  • e.g., Accept T.i
• Path feasibility
  • inconsistent path conditions
  • infeasible synchronization

Static analysis approaches for concurrent systems
• Specification approaches
  • Quantified regular expressions
  • Deterministic Finite State Automata
  • Temporal logics -- an extension to classical logic that supports statements about sequences and time
• Reachability graphs and reachability analysis
• Petri nets and Petri net based analysis
• Finite-State Verification
  • Data-flow analysis
  • Extended to distributed systems
  • Model checking
  • Flow equations
    • represent necessary conditions for all or some program executions