Concurrent Programming

CS105 Programming Languages Supplement
Outline

• Introduction
  – Categories
  – Concepts
• Semaphores
• Monitors
• Message Passing
• Statement-level concurrency
Introduction

• Definitions
  – *Process*: execution of sequence of statements
    • Private state, including associated resources and execution context
  – *Concurrent program*: two or more execution contexts
    • *Multithreaded*
  – *Parallel program*: concurrent program w/ more than one execution context active at the same time
  – *Distributed program*: programs run at same time on network of processors (w/out shared memory)
Introduction

• Types of concurrency
  – Instruction level
  – Statement level
  – Subprogram (unit) level
  – Program level
Multiprocessor Architectures

- **Single-Instruction Multiple-Data (SIMD)**
  - Each processor executes same instruction per step
  - No synchronization required
  - Vector machines or vector processors

- **Multiple-Instruction Multiple-Data (MIMD)**
  - Each processor executes private instruction stream
  - Two subtypes: shared memory and distributed
  - Both require synchronization
Categories of Concurrency

- Physical vs. logical concurrency
  - If the language supports logical concurrency, then we don't care whether multiple execution contexts actually active simultaneously
- Single thread of control
  - Concurrent program run on single processor virtually multithreaded
Significance of Concurrency

- A way of conceptualizing solutions to problems
  - Especially, simulation of physical systems
- Distributed architectures increasingly available
  - Need to write software to take advantage
  - Need language support for that software
Concepts for Concurrency

- Tasks/processes
- Synchronization
- Producer-Consumer Problem
- Race condition
- Deadlock
Concepts for Concurrency

- Tasks or processes
  - A unit of program allowing concurrent execution
  - Contrast with subprograms
    - Implicitly started vs. explicitly called
    - Program unit invoking task carries on w/out waiting
    - Task completion need not return control to caller

- Synchronization
  - Cooperative: task A needs result from task B
  - Competitive: tasks A and B need same resource
  - Producer-Consumer problem
Producer-Consumer Problem

- Shared buffer
- “Producer” generates items stored in buffer
- “Consumer” removes items from buffer
- Must prevent buffer underflow/overflow
Concepts for Concurrency

- Race condition
  - Competitive synchronization problem
  - `{total++;}{total*=2;}` involving fetch, change, store
  - Multiple possible outcomes
    - If first completes before second starts, ...
    - If second completes before first starts, ...
    - If both fetch, then first finishes before second, ...
    - If both fetch, then second finishes before first, ...
Concepts for Concurrency

- **Deadlock**
  - Tasks A and B require resources X and Y
  - Task A obtains X and B gets Y
  - Both must wait for other to release their resource
  - Neither will release, thus neither will complete
Supporting Concurrency

- Managing mutually exclusive access to shared resources
  - Process requests exclusive access to resource
  - Releases exclusivity when done
- Management methods
  - Semaphores
  - Monitors
  - Message passing
- A scheduler manages requests and sharing
States of Tasks

- New (created but not run)
- Ready (not running but runnable)
- Running (execution context active on processor)
- Blocked (previously running but now waiting)
- Dead (completed or killed)
Language Design Issues

• Support for concurrency:
  – Synchronization (both competition & cooperation)
  – Task scheduling
  – Task initiation and termination
Outline

- Introduction
- Semaphores
- Monitors
- Message Passing
- Statement-level concurrency
Semaphores

- Formulated by Dijkstra (1965)
- Data structure consists of
  - Integer
  - Queue of task descriptors
- Provides limited access, or guard, to shared resource
- Queue ensures delayed requests eventually get served
- Originally, P and V for Dutch pass and release
  - We'll use wait and release
Cooperative Synchronization

- Use shared buffer for example
- Use two semaphores to manage access to buffer
- Counter of semaphores used to track contents
  - Emptyspots semaphore tracks available capacity
  - Fullspots tracks used
  - Respective queues contain tasks waiting for respective resource
Cooperative Producer-Consumer

- Buffer as ADT
  - DEPOSIT adds data, FETCH removes
  - DEPOSIT needs check for available capacity
  - FETCH needs check for available data
- The semaphores: emptyspots and fullspots
  - DEPOSIT: if emptyspots, first decrements emptyspots, then increments fullspots after; else caller added to wait queue
  - FETCH: if fullspots, first decrements fullspots, then increments emptyspots after; else caller added to wait queue
Operations on Semaphores

- Wait and release subprograms access semaphore

```plaintext
wait(aSemaphore)
if aSemaphore's counter > 0 then
    decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task (if none, deadlock)
end
release(aSemaphore)
if aSemaphore's queue is empty (no task waiting) then
    increment aSemaphore's counter
else
    put calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```
semaphore fullspots, emptyspots;
fullspots.count := 0;
emptyspots.count := BUFLEN;
task producer;
    loop
        ... produce VALUE ...
        wait(emptyspots);  { wait for a space } 
        DEPOSIT(VALUE);
        release(fullspots);  { increase filled spaces } 
    end loop;
end producer;
task consumer;
    loop
        wait(fullspots);  { make sure it is not empty } 
        FETCH(VALUE);
        release(emptyspots);  { increase empty spaces } 
        ... consume VALUE . . . 
    end loop;
end consumer;
Competitive Synchronization

- Suppose multiple producers and consumers
  - Previous solution supports coordination among two cooperating processes
  - Use third semaphore to control access to buffer
    - Wait on this grants access if not being used (1)
    - If 0, then in use and caller placed on wait queue
    - (doesn't really use counter – *binary semaphore*)
semaphore access, fullspots, emptyspots;
access.count := 1;
fullspots.count := 0;
emptyspots.count := BUFLEN;
task producer;
    loop
        ... produce VALUE ...
        wait(emptyspots);  { wait for a space }
        wait(access); { wait for access }
        DEPOSIT(VALUE);
        release(access);  { relinquish access }
        release(fullspots); { increase filled spaces }
    end loop;
end producer;
task consumer;
    loop
        wait(fullspots);  { make sure it is not empty }
        wait(access); FETCH(VALUE); release(access);
        release(emptyspots); { increase empty spaces }
        ... consume VALUE . . .
    end loop;
end consumer;
Analysis of Semaphores

• Note: semaphore operations must be \textit{uninterruptible}.
• Semaphore solution to synchronization
  – Cannot be statically checked for correctness
  – Forgetting \texttt{wait(emptyspots)} yields buffer overflow
  – Forgetting \texttt{wait(fullspots)} yields buffer underflow
  – Forgetting releases yields deadlock
  – Similar problems for access semaphore
• “The semaphore is an elegant synchronization tool for an ideal programmer who never makes mistakes.” (Brinch Hansen, 1973)