Concurrencyn

- **def**: result of interleaved execution of multiple processes/threads in pursuit of multiprogramming
- is of great concern in operating systems to prevent errors when dealing with *shared resources*
Shared Resources

- processes or threads that execute asynchronously but cooperate towards a common goal may need to share a common resource
  - ex: shared memory (for processes) or variable (for threads), file, I/O device

Race Condition

- uncontrolled access to the resource may result in a race condition
  - an erroneous situation where parallel processes attempt to update a shared resource such that the final result is dependent on the execution timing
    - i.e., who gets there last?
    - ex: tracking total requests in a multithreaded web server
    - ex: document content in a WP w/ autosave
    - ex: bank account transactions
Mutual Exclusion

- **def**: a programmatic (controlled) approach to preventing race conditions between multiple, concurrently executing processes/threads by ensuring that only 1 process is allowed to access the shared resource at a time
  - i.e., *serializing* access to the resource

Critical Region

- **def**: the smallest part of a program that accesses the shared resource
- must be controlled via mutual exclusion
- should be executed quickly
  - because other processes may be waiting
  - considered an *atomic operation*
- must not block!
  - else *deadlock* could occur
**Deadlock**

- **def:** a condition where multiple processes are blocking each other for access to a shared resource and unable to proceed
  - ex: left turns at a traffic intersection
  - ex: one-lane bridge
- highly undesired!

**Example: Producer/Consumer**

- a relationship that divides data processing into two "agents"
  - **producer:** generates data into a shared resource
    - must notify consumer that data is available
    - must not write over unconsumed data
  - **consumer:** retrieves data from the shared resource
    - must not consume data that wasn't yet produced
    - must notify producer that data was consumed
Mutual Exclusion

Producer/Consumer

- example scenario...
  - consequence is incorrect results
- each agent needs a state model
  - waiting & ready
- solution requires synchronization
  - for proper cooperation
  - ex: a "notify" signal

Implementing Mutual Exclusion

- a key problem in concurrent programming
  - how to do it?
- various solutions exist, using various approaches
  - hardware-based (CPU features)
  - software-based: low-level & high-level
  - use of strict protocols
    - easily "broken"
Mutual Exclusion

Mutual Exclusion Problems

- **busy-waiting**
  - wasted CPU utilization

- **deadlock**
  - 2 processes/threads blocked on each other

- **indefinite postponement**
  - *processor starvation* due to "favorite thread"

Mutual Exclusion Requirements

- A good solution to avoiding race must enforce 5 requirements:
  1. *no fighting* – no 2 cooperating threads may be inside their critical regions simultaneously
  2. *no bullying* – no thread running outside its critical region may block other processes
  3. *play fair* – no thread should have to wait forever to enter its critical region
  4. *no hogging* – no thread shall stay in its critical region indefinitely
  5. *no gambling* – no assumptions about speeds or number of CPUs may be made
Mutual Exclusion

Mutual Exclusion Solutions

- **hardware**
  - interrupt control
  - atomic instructions

- **software**
  - algorithms
    - Dekker's, Peterson's, Lamport's
  - software mechanisms
    - semaphores, monitors, condition variables

Hardware Approaches

- **disabling interrupts**
  - no interrupts = no context switches!
  - highly undesirable; can result in "hangs"
    - except for *trusted code* in kernel mode

- **CPU atomic instructions**
  - guaranteed to finish once started
  - ex: `testAndSet(a,b)`
  - works, but results in busy-waiting
  - also suffers from priority inversion
Software Approaches

- lock variable(s)
  - 0 = permission to enter;
    1 = some process is in CR
  - violates req. 5 & 1; won't work!

- strict alternation; i.e. "take turns"
  - ex: prod-cons, ver. 2
  - results in busy-waiting, violates req. 2

Dekker's & Peterson's Algorithms

- first s/w-only correct solution to 2-thread mutual exclusion
- adds concept of "favorite thread" to use of lock variable flag
  - favorite thread alternates back-and-forth
- still suffers from busy-wait
- Peterson's: simplified version, same concept & problem
Lamport's Bakery Algorithm

- implements mutual exclusion for N threads
- based on "take a ticket" model
  - as found in a busy bakery
- thus, is first-come-first-served (FCFS)
- employs 2 arrays:
  - bool choosing[] – indicates which threads are "taking a ticket"
  - int ticket[] – contains each thread's ticket
    - 0=not in CR nor waiting

Dijkstra's Semaphores

- from Netherlands, father of concept "GOTO is bad"
- semaphore:
  - a protected, integer variable of the number of wakeups received
- modified only with 2 atomic ops:
  - P (proberen), "to test"
  - V (verhogen), "to increment"
  - implemented in kernel
Semaphore Operations

- on a semaphore S, assume initial value = 1
- P (test & wait):
  - if S>0
    - S = S – 1 (and continue the calling thread)
  - else
    - suspend calling thread on S
- V (increment):
  - if any threads waiting on S
    - resume next waiting thread
  - else
    - S = S + 1

Producer/Consumer via Semaphores

- uses 2 semaphores
  - valueProduced – triggers consumer
  - valueConsumed – triggers producer

Producer:
1. P on valueConsumed
2. <critical region>
3. V on valueProduced

Consumer:
1. P on valueProduced
2. <critical region>
3. V on valueConsumed

- ensures mutual exclusion & alternation without busy-waiting!
Mutual Exclusion

Monitors

- a high-level mechanism for mutex
- an object that contains both data and procedures to allocate & manage serial reuse of a shared resource
  - monitor data is private to the monitor
- avoids concurrency problems by enforcing threads to "enter" & "exit" monitor
  - via a monitor entry routines
- con: must exist as a prog. language construct
  - ex: built into the Java language!

Condition Variables

- Dijkstra's semaphore concept applied to a variable, with addition of a wait queue
  - wait(conditionVar)
  - signal(conditionVar)
- used in monitors to enforce mutual exclusion
- can solve various concurrency problems
  - producer/consumer, resource allocation, circular buffers, readers/writers
Readers & Writers Problem

- models concurrent access to a shared database
  - multiple readers (consumers – no changes) & writers (producers – make changes)
  - may have n readers with 0 writers or 1 writer with 0 readers
  - may be solved with condition variables and a monitor

Monitor App: Readers & Writers

- data:
  - int readers – counts number of readers
  - bool writeLock – true when 1 write active
  - Condition canWrite – allows writer to write
  - Condition canRead – allows readers to read

- procedures:
  - beginRead(), endRead()
  - beginWrite(), endWrite()