CS3841 – Design of Operating Systems
Concurrency

Problem

- Race Condition – Two processes/threads are manipulating the same data at the same time with the potential of having different outcomes
- Critical Section - A segment of code that accesses a shared resource that must not be concurrently accessed by more than one thread of execution.

Solutions?

- Multiple processes that don't share address spaces
  - Modify values using pipes, messages, etc.
- Atomic CPU operations can help (Fetch and Add, Fetch and Subtract)
  - Atomic CPU operations are not always portable
  - Does not work for multiple variables (can only atomically change one at a time)

Critical Section Guarantees

- Mutual Exclusion
  - If a process is executing in its critical section, then no other processes can be executing in their critical sections
- Progress
  - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- Bounded Waiting (starvation freedom / fairness)
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Critical Section Progress

- Progress
  - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- Deadlock (aggressive holding of resources)
  - Process ‘A’ is executing in its critical section while waiting for process ‘B’
  - At the same time, process ‘B’ is executing in its critical section while waiting for process ‘A’
  - Can either continue?
- Livelock (passive ‘holding’ of resources)
  - Process ‘A’ releases a resource so process ‘B’ can continue, but must wait for ‘B’ to be done
  - Once the resource is received, process ‘B’ releases the resource so process ‘A’ can continue
  - Can either continue?

Examples of Mutual Exclusion Algorithms

- See code samples
  - cons1, cons2, cons3, cons4, cons_dekkers
  - Progression of potential software solutions for mutual exclusion
  - cons1 – Take turns
    - Advantages – provides mutual exclusion
    - Disadvantages
      - Does not work for more than 2 processes
      - Does not provide progress if one process doesn't ever want to enter the critical section
  - cons2 – Entry flags
    - Advantages –
      - A process expresses the desire to enter the critical section (no need to take turns)
      - Allows progress if one process never wants the critical section
    - Disadvantages –
      - No control over when the flags are set, does not ensure mutual exclusion

Examples of Mutual Exclusion Algorithms

- See code samples
  - cons1, cons2, cons3, cons4, cons_dekkers
  - Progression of potential software solutions for mutual exclusion
  - cons3 – Entry flags with wait loop
    - Advantages – Provides mutual exclusion: if one process sets the flag, the other will wait
    - Disadvantages – Deadlock if both processes set their flag at the same time
  - cons4 – Entry flags with deadlock check
    - Advantages –
      - Provides mutual exclusion
      - Ensures progress with flags and deadlock detection
    - Disadvantages –
      - Prone to live lock

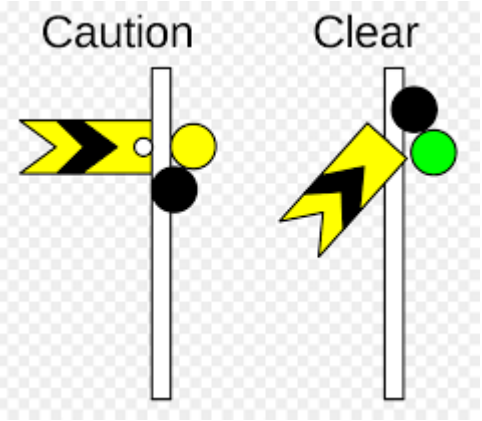
Examples of Mutual Exclusion Algorithms

- See code samples
  - cons1, cons2, cons3, cons4, cons_dekkers
  - Progression of potential software solutions for mutual exclusion
  - cons3_dekkers – Entry flags with wait loop, deadlock check, and taking turns
    - Advantages
      - Provides provable mutual exclusion and progress
      - Prevents deadlock with inner wait loop with flag checking
      - Prevents livelock by taking turns
    - Disadvantages – Must know ahead of time how many processes are participating

- None of the software methods solve the problem with weak memory consistency!

Semaphore

- Signaling mechanism from one process to another
- Keeps a current value count
- Operations: signal and wait
  - Signal -> atomically increments the value by 1 -> releases a process if value was < 0
  - Wait -> atomically decrements the value by 1 -> forces the process to wait if new value < 0
- Special case – binary semaphore – has a max value of 1



- Semaphore concerns:
  - If more than 1 process is waiting, which one will be released after a signal?
    - Linux uses a FIFO wait queue, first process forced to wait is the first process to be released
  - Semaphore do no enforce ownership – NOT A LOCK

Mutex Lock

- Like a binary semaphore
- Can be locked or unlocked
- Maintains ownership
- Operations: lock and unlock
  - lock -> locks the mutex if it is unlocked (recording ownership)
    if mutex is locked by someone else the locker is forced to wait
  - unlock -> unlocks the mutex if it is locked
    if someone is waiting for the lock, it locks the mutex under their ownership and lets them continue

- Only the owner of a mutex can unlock it
- Mutex Concerns:
  - What if the owner locks the mutex more than once?
  - What if the owner unlocks the mutex more than once?

Semaphore vs Mutex

Mutex	Semaphore
Can only be used to synchronize threads	Can synchronize across processes as well as threads
Only binary	Can be binary or counting
Lighter weight in implementation	Heavier weight in implementation
Can only be released by the thread that locked it	Can be released by any running thread
Locking Mechanism	Signaling Mechanism

Producer and Consumer

- Control access to a shared buffer
- Producer puts items into the buffer
- Consumer removes items from the buffer
- Need to make sure the producer doesn't put items into a full buffer
- Need to make sure the consumer doesn't remove items from an empty buffer