Semaphores

CSCI 444/544 Operating Systems
Fall 2008

Agenda

• What is semaphore?
  – Two basic operations
  – Two types of semaphores

• Usage of semaphores

• Semaphore implementation

• Producer/consumer problem

Background

Semaphore = a synchronization primitive
  • higher level of abstraction than locks
  • is like a generalized lock
  • invented by Dijkstra in 1968, as part of the THE operating system

Locks only provide mutual exclusion
  • Ensure only one thread is in critical section at a time

Semaphore have two purposes
  • Mutex: Ensure threads don’t access critical section at same time
  • Scheduling constraints: Ensure threads execute in specific order

Semaphore

A semaphore is:
  • an integer variable that is manipulated through two operations,
    P and V (Dutch for “wait” and “signal”)
    – P(sem) (wait/down)
      • block until sem > 0, then subtract 1 from sem and proceed
    – V(sem) (signal/up)
      • add 1 to sem

Do these operations atomically
Two Types of Semaphores

Binary semaphore (aka mutex semaphore)
- sem is initialized to 1
- guarantees mutually exclusive access to resource (e.g., a critical section of code)
- only one thread/process allowed entry at a time

Counting semaphore
- sem is initialized to N
  - N = number of units available
- represents resources with many (identical) units available
- allows threads to enter as long as more units are available

Usage of Semaphores

To achieve mutual exclusion
- ensure that only 1 thread (or more generally, less than N threads) is in critical section
- initial value of semaphore is set to 1 (or more generally N)
  - wait()
  - <critical section>
  - signal()

Like lock/unlock, but more general

Usage of Semaphores (con’d)

To achieve scheduling constraints
- used when a thread should wait for some event (not just another thread leaving critical section)
- initial value of semaphore is set to 0 (usually but not always)

Thread A: wait() continue execution
Thread B: wait() Do task

Implementing Semaphores (I)

The function wait():

```c
wait(S) {
    while (S <= 0);
    // no-op
    S--;
}
```

The function signal():

```c
signal(S) {
    S++; 
}
```
Implementing Semaphores (II)

typedef struct {
    int value;
    queue tlist;
} semaphore;

wait (semaphore *S) {  // Must be executed atomically
    S->value--;
    if (S->value < 0) {
        add this process to S->tlist;
        block();
    }
}

signal (semaphore *S) {  // Must be executed atomically
    S->value++;
    if (S->value <= 0) {
        remove thread t from S->tlist;
        wakeup(t);
    }
}

Queue in Semaphores

Each semaphore has an associated queue of threads
• when P (sem) is called by a thread,
  – if sem was “available” (>0), decrement sem and let thread continue
  – if sem was “unavailable” (<=0), block thread and place it on associated queue, run some other thread
• when V (sem) is called by a thread
  – if thread(s) are waiting on the associated queue, unblock one
    • place it on the ready queue
    • might as well let the “V-ing” thread continue execution
  – otherwise (when no threads are waiting on the sem), increment sem
    • the signal is “remembered” for next time P(sem) is called

Simple Observations

• Sem value is negative --> Number of waiters on queue

• Sem value is positive --> Number of threads that can be in critical section at same time

Semaphores vs Locks

Threads that are blocked at the level of program logic are placed on queues, rather than busy-waiting

Busy-waiting may be used for the “real” mutual exclusion required to implement P and V
• but these are very short critical sections – totally independent of program logic
Producer/Consumer Problem

Also known as bounded-buffer problem

- \( N \) buffers, each can hold one item
- Three semaphores: one binary, two counting
- Binary Semaphore \texttt{mutex} initialized to the value 1
- Counting semaphore \texttt{full} initialized to the value 0
- Counting semaphore \texttt{empty} initialized to the value \( N \).

Producer Thread

The structure of the producer

\begin{verbatim}
    do {
        // produce an item
        wait (empty);
        wait (mutex);
        // add the item to the buffer
        signal (mutex);
        signal (full);
    } while (true);
\end{verbatim}

Consumer Thread

The structure of the consumer

\begin{verbatim}
    do {
        wait (full);
        wait (mutex);
        // remove an item from buffer
        signal (mutex);
        signal (empty);
        // consume the removed item
    } while (true);
\end{verbatim}

Problems with Semaphores

They can be used to solve any of the traditional synchronization problems, but:

- semaphores are essentially shared global variables
  - can be accessed from anywhere (bad software engineering)
- there is no connection between the semaphore and the data being controlled by it
- used for both critical sections (mutual exclusion) and for coordination (scheduling)
- no control over their use, no guarantee of proper usage

Thus, they are prone to bugs

- another (better?) approach: use programming language support
  - monitor