The Swap Primitive

- Semantics: Atomically exchange the values of variables X and Y
  - temp = X; X = Y; Y = temp;

- Swap and mutual exclusion:
  - can emulate test-and-set by swap

function test-and-set(var v: boolean): boolean
  var t := true;
  swap (v, t);
  return t;

- Can we also get bounded waiting?

Bounded Waiting Using Test-and-Set

1: waiting[i] = true;
2: key := true;
3: while ( waiting[i] and key )
   do key := Test-and-Set(lock);
4: waiting[i] = false;

CRITICAL SECTION
5: j := i+1 mod n;
6: while (j != i) and (!waiting[j])
   do j := j + 1 mod n;
7: if (j == i) then lock = false;
   else waiting[j] = false;

- Correctness?
Solutions So Far

- Mutual exclusion
  - software algorithms
    - 2-process: Turn, Flag, Petersen
    - N-process: Bakery
  - hardware primitives

- Limitations of these algorithms
  - do not solve more general synchronization problems
    - only mutual exclusion
    - involve “busy-waiting”

- Are there more general solutions?

Classical Synchronization Problems

- Commonly encountered problems in operating systems
  - used to test any proposal for a new synchronization primitive

  ① Mutual exclusion
    - only one process executes a piece of code (critical section) at any time

  ② Sequencing
    - a process waits for another process to finish executing some code

  ③ Bounded-buffer (also referred to as the Producer-Consumer problem)
    - a pool of \( n \) buffers
      - producer process(es) put items into the pool
      - consumer process(es) take items out of the pool
    - issues: mutual exclusion, empty pool, and full pool

Classical Synchronization Problems (contd.)

  ④ Readers- Writers
    - multiple processes access a shared data object \( X \)
      - any number of readers can access \( X \) at the same time
      - no writer can access it at the same time as a reader or another writer
    - mutual exclusion is too constraining
    - variations:
      - reader-priority: a reader must not wait for a writer
      - writer-priority: a writer must not wait for a reader

  ⑤ Dining Philosophers
    - 5 philosophers
    - 5 chopsticks placed between them
      - to eat requires two chopsticks
    - philosophers alternate between thinking and eating
    - issues: deadlock, starvation, fairness

Higher-level Synchronization Primitives

- Hard to express the above synchronization problems only using solutions for mutual exclusion
  - need higher-level primitives

- Semaphores
  - a single integer variable \( S \)
    - accessed via two atomic operations
      - \( \text{WAIT} \) (sometimes denoted by \( P \))
        - \( \text{while } S \leq 0 \text{ do waitbit; } S := S-1; \)
      - \( \text{SIGNAL} \) (sometimes denoted by \( V \))
        - \( S := S+1; \)
    - \( \text{WAITing} \) process(es) cannot “lock out” a \( \text{SIGNALing} \) process
    - this definition still involves busy-waiting
      - we will rectify this later in the lecture
Uses of Semaphores

- Mutual exclusion: Semaphore initialized to 1
  \[ P(S); \]
  CRITICAL SECTION
  \[ V(S); \]

- Sequencing: Semaphore initialized to 0
  \[
  \begin{array}{ll}
  \text{process 1} & \text{process 2} \\
  B(); & V(S); \\
  P(S); & A(); \\
  \end{array}
  \]

- Bounded-buffer: Semaphore \( \text{numitems} \) initialized to 0
  - producer: \( \text{insert item; } V(\text{numitems}); \)
  - consumer: \( \text{P(\text{numitems}); remove item; } \)
  - still need to handle (solution in text)
    - full buffer: ensure that \( \text{numitems} \) value does not go above \( N \)
    - take care of mutual exclusion (as before)

Readers-Writers Using Semaphores

Reader

\[ P(x); \]
\[ rcount := rcount + 1; \]
\[ \text{if } (rcount == 1) \text{ then } P(wsem); \]
\[ V(x); \]
\[ \text{READ} \]
\[ P(x); \]
\[ rcount := rcount - 1; \]
\[ \text{if } (rcount == 0) \text{ then } V(ssem); \]
\[ V(x); \]

Writer

\[ P(wsem); \]
\[ \text{WRITE } V(wsem); \]
\[ P(y); \]
\[ wcount := wcount + 1; \]
\[ \text{if } (wcount == 1) \text{ then } P(rsem); \]
\[ V(y); \]
\[ P(wsem); \]
\[ \text{WRITE } V(wsem); \]
\[ P(y); \]
\[ wcount := wcount - 1; \]
\[ \text{if } (wcount == 0) \text{ then } V(rsem); \]
\[ V(y); \]
**Dining Philosophers Using Semaphores**

- Deadlock
  - a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set
  - details in Lectures 9 and 10.

  - Alternate solutions
    - allow at most 4 philosophers to sit simultaneously at the table
    - allow a philosopher to pick up chopsticks only if both are available
  - All of these solutions suffer from the possibility of starvation!

**Binary Semaphores**

- Semaphores as discussed so far
  - sometimes called counting semaphores since their values range over an unrestricted domain

- A binary semaphore
  - takes on values 0 or 1 only
  - typically easier to implement
  - is universal in that counting semaphores can be built out of them
  - in a sense, is a minimal structure
    - the rest can be realized without additional requirements on atomicity

**Universality of Binary Semaphores**

- Implement operations on a (counting) semaphore `mutex`
  - use binary semaphores `S1 = 1, S2 = 0`
  - integer `C = initial value of counting semaphore`

  ```plaintext
  P(mutex) V(mutex)
  P(S1); P(S1); C := C-1; C := C+1;
  if ( C < 0 ) then begin V(S1); P(S2); end
  else V(S1); V(S1);
  V(S2); V(S1);
  ```

  - `S1` ensures mutual exclusion for accessing `C`
  - `S2` is used to block processes when `C < 0`
  - is a race condition possible after `V(S1)` but before `P(S2)`?
Implementation of Semaphores

- Revisit code for P
  
  ```
  while S <= 0 do waitabit;
  S := S-1;
  ```
  
  - a process spins on this loop till it gets a chance to enter critical section
  - can waste *substantial* amount of CPU cycles idling
    
    - Even if `waitabit` is implemented as
      
      ```
      give up CPU (i.e. put at the end of ready queue)
      ```
    
    - not a very useful utilization of valuable cycles

Efficient Semaphores

- Implement P and V differently
  
  - maintain an explicit *wait queue* organized as a scheduler structure

  ```
  type semaphore = record
    value: integer;
    L: list of processes;
  end;

  P(S): S.value := S.value - 1;
  if ( S.value < 0 ) then begin
    add process to S.L block;
  end;

  V(S): S.value := S.value + 1;
  if ( S.value <= 0 ) then begin
    remove P from S.L
    wakeup(P);
  end;
  ```
  
  - need to ensure mutual exclusion: can use previously discussed solutions
    
    - still have spinning but only for a small period of time (~10 instructions)
    
    - queue enqueue/dequeue must be fair
      
      - not required by semantics of semaphores

Lecture Summary

- Classical problems of synchronization
  
  - mutual exclusion, sequencing, bounded-buffer, readers-writers, dining philosophers

- Higher-level synchronization primitive: Semaphores
  
  - semantics and use
  
  - universality of binary semaphores
  
  - implementation of semaphores without busy-waiting

- Limitations of semaphores?

Limitations of Semaphores

- No abstraction and modularity
  
  - a process that uses a semaphore has to know which other processes use the semaphore, and how these processes use the semaphore
  
  - a process cannot be written in isolation

- Very easy to write incorrect code
  
  - changing the order of P and V
    
    - can violate mutual exclusion requirements
    
    - can cause deadlock
    
    - similar problems with omission

- Extremely difficult to verify programs for correctness
  
  - Need for still higher-level synchronization abstractions!
Next Lecture

- Language support for concurrency
  - conditional critical regions
  - monitors
  - path expressions

Readings
- Silberschatz/Galvin: Sections 6.6-6.8