Synchronization

H. Levy

Synchronization

- **Basic Problem:**

  If two concurrent processes are accessing a shared variable, and that variable is <u>read, modified, and written</u> by those processes, then the variable must be controlled to avoid erroneous behavior.

ATM Example

- **Suppose each cash machine transaction is controlled by a separate process, and the withdraw code is:**

```
curr_balance = get_balance (acct_ID)
withdraw_amt = read_amount_from_ATM()
if withdraw_amt > curr_balance then error
curr_balance = curr_balance - withdraw_amt
put_balance (act_ID, curr_balance)
deliver_bucks(withdraw_amt)
```

- **Now, suppose that you and your s.o. share an account. You each go to separate cash machines and withdraw $100 from your balance of $1000.**

ATM Example

```
you:  curr_balance = get_balance(acct_ID)
you: withdraw_amt =read_amount()
you: curr_balance=curr_balance-withdraw_amt
so: curr_balance = get_balance(acct_ID)        <---- context switch
so: withdraw_amt =read_amount()
so: curr_balance=curr_balance-withdraw_amt
so: put_balance(acct_ID, curr_balance)
so: deliver_bucks(withdraw_amt)
you: put_balance(acct_ID, curr_balance)        <---- context switch
you: deliver_bucks(withdraw_amt)
```

- **What happens?**
- **Why does it happen?**

Problems

- **A problem exists because a shared data item (curr_balance) was accessed <u>without control</u> by processes that read, modified, and then rewrote that data.**
- **We need ways to control access to shared variables.**

Semaphores

- **Dijkstra, in the THE system, defined a type of variable and two synchronization operations that can be used to control access to <u>critical sections</u>.**
- **A <u>semaphore</u> is a <u>variable</u> that is manipulated <u>atomically</u> through operations *signal*(s) and *wait*(s).**
- **To access a critical section, you must:**

  wait(s) ; wait until semaphore is available

  <critical section code>

  signal(s) ; signal others to enter

Semaphores

- Associated with each semahore is a queue of waiting processes.
- If you execute *wait*(s) and the semaphore is free, you continue; if not, you block on the waiting queue.
- A *signal*(s) unblocks a process if it's waiting.

Signal and Wait

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

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

signal(S):
      S.value = S.value + 1;
      if S.value <= 0
      then begin
            remove a process P from S.L;
            wakeup P
            end;
```

Signal and Wait must be atomic

Example: Readers/Writers Problem

- Basic Problem:
  - an object is shared among several processes, some which only read it, and some which write it.
  - we can allow multiple readers at a time, but only one writer at a time.
  - how do we control access to the object to permit this protocol?

Readers/Writers Sema Implementation

```
var mutex: semaphore  ; controls access to readcount
    wrt: semaphore       ; control entry to a writer or first reader
    readcount: integer  ; number of readers

write process:
     wait(wrt)              ; any writers or readers?
        <perform write operation>
     signal(wrt)            ; allow others

read process:
     wait(mutex)           ; ensure exclusion
          readcount = readcount + 1  ; one more reader
          if readcount = 1 then wait(wrt) ; if we're the first, synch with writers
     signal(mutex)
          <perform reading>
     wait(mutex)           ; ensure exclusion
          readcount = readcount - 1 ; one fewer reader
          if readcount = 0 then signal(wrt) ; no more readers, allow a writer
     signal(mutex)
```


Readers/Writers Impl. Notes

- Note that:
  1. the first reader blocks if there is a writer; any other readers who try to enter will then block on *mutex*.
  2. once a writer exits, all readers will fall through.
  3. the last reader to exit signals a waiting writer.
  4. when a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.

Semaphore Types

- In general, there are two types of semaphores:
  - a mutex semaphore guarantees mutually exclusive access to a resource (only one entry). the mutex sema is intialized to 1.
  - a counted semaphore represents a resource with many units available (as indicated by the count to which it is initialized). a counted semaphore lets a process pass as long as more instances are available.

Example: Bounded Buffer Problem

- **The Problem:**

  There is a buffer shared by *producer* processes, which insert into it, and *consumer* processes, which remove from it.

  The processes are concurrent, so we must control their access to the (shared) variables that describe the state of the buffer.

Bounded Buffer Sema Implementation

```
var mutex: semaphore = 1  ; mutual exclusion to shared data
    empty: semaphore = n  ; count of empty buffers (all empty to start)
    full: semaphore = 0   ; count of full buffers (none full to start)

producer:
    wait(empty)  ; one fewer buffer, block if none available
    wait(mutex) ; get access to pointers
        <add item to buffer>
    signal(mutex) ; done with pointers
    signal(full) ; note one more full buffer

consumer:
    wait(full)  ; wait until there's a full buffer
    wait(mutex)  ; get access to pointers
        <remove item from buffer>
    signal(mutex) ; done with pointers
    signal(empty) ; note there's an empty buffer
        <use the item>
```


Busy Waiting with Test-and-Set

- **A test-and-set instruction provides the atomic guarantees needed to implement critical sections**

```
Boolean Function test-and-set (var target: boolean)
begin
   test-and-set := target    ; return old value of target
   target := true            ;  set target to true
end
```

using test-and-set (initialize aLock:= false at beginning)

```
while test-and-set(aLock) do no-op    ;  spin until lock is free (and atomically
                                      ;  acquite the lock)
        <do the critical section>

aLock := false                        ;  free the lock for someone else
```