WASHINGTON STATE UNIVERSITY
VANCOUVER

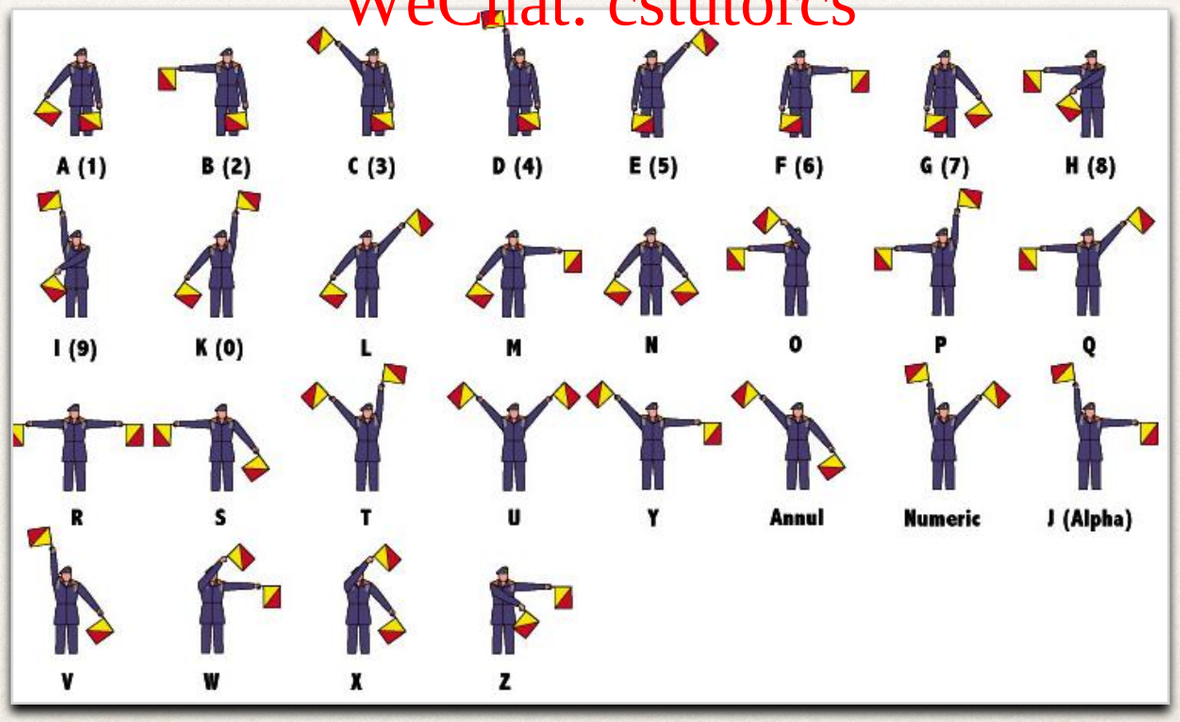
CS 563

Concurrent Programming

Lecture 14: Semaphores

Semaphores

- History: Dijkstra -- 1968
- Basic idea: comes from train semaphores to signal whether the track is free



Definition


```
sem s := 0    # or some non-negative value
P(s):
   < await (s>0) s = s-1; >
V(s):
   < s = s+1; >
```

What P and V stand for?

Probeer (try) and Verhoog (increment)

Basic Uses of Semaphores

- Critical sections: mutual exclusion


```
sem mutex = 1;
```



```
process CS[i = 1 to n] {
```



```
    while (true) {
        P(mutex);
        critical section;
        V(mutex);
        noncritical section;
    }
}
```

Barriers: Signaling

```
sem arrive[1:n] = ([n] 0);   # initially
                  # zeros to indicate that
                  # conditions have not yet
                  # occurred

Pr[i]:   V(arrive[i]);  # signal I am here
         P(arrive[j]);  # wait for partner

Pr[j]:   V(arrive[j]);  # signal I am here
         P(arrive[i]);  # wait for partner
```


- Combine instances of these to form a dissemination or butterfly structure

Producer/Consumer: Split Binary Semaphores

- Recall the problem


```
sem empty = 1, full = 0;
Producer:  P(empty); deposit; V(full);
Consumer:  P(full); fetch; V(empty);
```


- Split binary semaphores: empty and full can be seen as a single binary semaphore that has been split into two binary sems

- Key property: If they are used as above, mutual exclusion is satisfied between a P and next V

Generalization

```
sem empty = 1, full = 0;
Producer:  P(empty); deposit; V(full);
Consumer:  P(full); fetch; V(empty);
```


- Generalize producer/consumer solution
- What changes: representation of buffer and initial value of empty. That's all!

Producer/Consumer

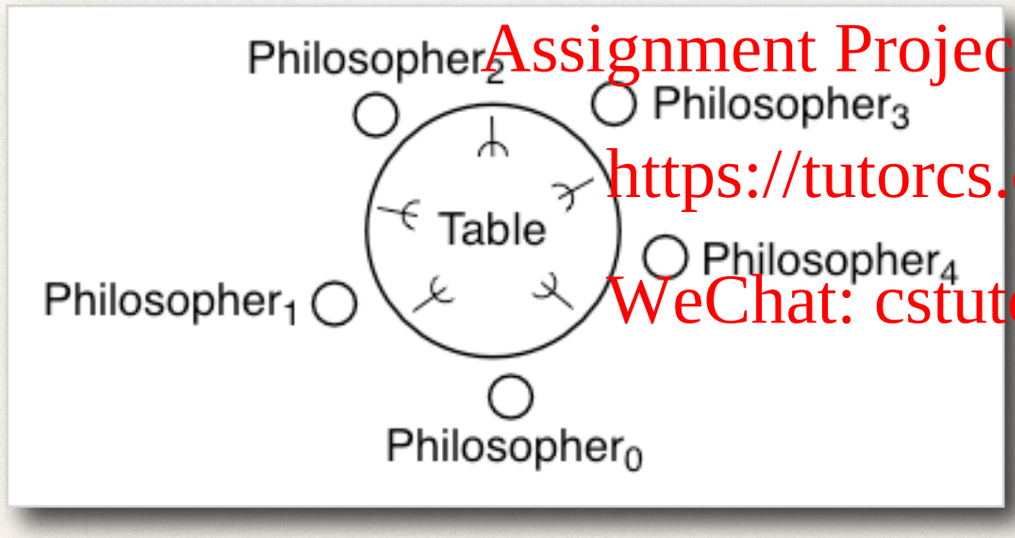
```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;

process Producer {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    buf[rear] = data; rear = (rear+1) % n;
    V(full);
  }
}
process Consumer {
  while (true) {
    fetch message result and consume it;
    P(full);
    result = buf[front]; front = (front+1) % n;
    V(empty);
    ...
  }
}
```

Producer/Consumer

```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;   /* for mutual exclusion   */
process Producer[i = 1 to M] {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear+1) % n;
    V(mutexD);
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    fetch message result and consume it;
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

Dining Philosopher



Solution Idea

- Pick up two forks


```
sem forks[1:5] = ([5] 1);    # each fork
                 # is a critical section
                 # so use 1 for initial value
```


- First attempt for philosopher code

```
P(fork[i]); P(fork[i%5 + 1])
        "eat"
        V(fork[i]); V(fork[i%5 + 1]);
        "think"
```

What is the Problem?

- Deadlock due to circular waiting
- Solutions:
- (a) change order for some, but not all
- (b) limit number at table


```
sem limit = 4;
        P(limit); above code; V(limit);
```

Readers/Writers Problem

```
sem rw = 1;
```



```
P(rw);                    P(rw);
read                      write;
V(rw);                    V(rw);
```


Over-constrained!

Allow Concurrent Readers

- Idea: first reader to arrive does P(rw) and last to leave does V(rw)
  - This requires keeping a counter


```
P(mutexR);
   nr++
   if (nr == 1) P(rw);
V(mutexR);
"read"
P(mutexR);
   nr--
   if (nr == 0) V(rw);
V(mutexR);


P(rw);
"write"
V(rw);
```

Now nr is a shared variable that has to be accessed atomically

Predicates

- Develop a predicate that exactly characterize the good and bad states
- nr = number of readers; nw = number of writers

- BAD state (to be avoided):
  - (nr > 0 and nw > 0) or (nw >1)
- GOOD states == not BAD states == RW
  - (nr = 0 or nw = 0) and (nw <= 1)

Coarse-Grained Solution

- We want RW to be a global invariant
- so,


```
reader:
   < await (nw == 0) nr++; >
   "read"
   < nr--; >
writer:
   < await (nr == 0 and nw == 0) nw++; >
   "write"
   < nw--; >
```

Fine-Grained Solution

- How can we turn the above into a solution that just uses semaphores?
- Need CS protection and to implement the delay in the await statements
- We actually know how to do this using split binary semaphores

Outline of Basic Idea of "Passing the Baton"

```
Start with baton on table
Each process:
 1.waits to be able to pick up the baton
2.if (not OK to proceed) {
   a. increment a counter to record that you have to wait
   b. put the baton back on the table
   c. take a seat and wait (DELAY)
   }
3.once you are awakened (or if it was OK to proceed):
   a. change nr or nw
   b. give the baton to somebody else OR put it on the table
   (SIGNAL)
```

Outline

```
reader:
    < await (nw == 0) nr++; >
    "read"
    < nr--; >
writer:
    < await (nr == 0 and nw == 0) nw++; >
    "write"
    < nw--; >
```

```
int nr = 0,    ## RW: (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1,     # controls entry to critical sections
    r = 0,     # used to delay readers
    w = 0;     # used to delay writers
               # at all times 0 <= (e+r+w) <= 1
int dr = 0,    # number of delayed readers
    dw = 0;    # number of delayed writers
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      SIGNAL;
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      SIGNAL;
  }
}
```

```
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      SIGNAL;
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      SIGNAL;
  }
}
```

Signal

(nr = 0 or nw = 0) and (nw <= 1)


```
if (nw == 0 and dr > 0) {
    dr = dr-1; V(r); # awaken a reader, or
    }
elseif (nr == 0 and nw == 0 and dw > 0) {
    dw = dw-1; V(w); # awaken a writer, or
else
```

Solution

```
int nr = 0,    ## RW: (nr==0 or nw==0) and nw<=1
    nw = 0;
sem e = 1,     # controls entry to critical sections
    r = 0,     # used to delay readers
    w = 0;     # used to delay writers
               # at all times 0 <= (nr+nw) <= 1
int dr = 0,    # number of delayed readers
    dw = 0;    # number of delayed writers
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      if (dr > 0) { dr = dr-1; V(r); }
      else V(e);
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

Signal:

```
if (nw == 0 and dr > 0) {
      dr = dr-1; V(r); # awaken a reader, or
      }
elseif (nr == 0 and nw == 0 and dw > 0) {
      dw = dw-1; V(w); # awaken a writer, or
else
      V(e);          # release the entry lock
```

```
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      V(e);
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      if (dr > 0) { dr = dr-1; V(r); }
      elseif (dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

Shortest Job Next

```
request(time, id)
   if (free)
      take resource
   else
      delay by time

release(id)
   if (some process delayed)
      awaken first one (min value of time) and give it the resource
   else
      make resource free
```


General Solution Pattern:

```
request(parameters):
P(e);
if (request cannot be satisfied) DELAY;
take units;
SIGNAL;

release(parameters):
P(e);
return units
SIGNAL;
```

Shortest Job Next

```
bool free = true;
sem e = 1, b[n] = ([n] 0);  # for entry and delay
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
## SJN:  pairs is an ordered set ∧ free == (pairs == ∅)
request(time,id):
  P(e);
  if (!free) {
    insert (time,id) in pairs;
    V(e);              # release entry lock
    P(b[id]);          # wait to be awakened
  }
  free = false;
  V(e);     # optimized since free is false here
release():
  P(e);
  free = true;
  if (P != ∅) {
    remove first pair (time,id) from pairs;
    V(b[id]);  # pass baton to process id
  }
  else V(e);
```