Bounded Buffer Problem

- Consider 2 threads:
  - one producer, one consumer
  - real OS example: ps | grep dkl
    - shell forks a thread for "ps" and a thread for "grep dkl"
  - "ps" writes its output into a fixed size buffer; "grep" reads the buffer
  - access to a specific buffer slot is a critical section, but not between slots:
    - also may need to wait for buffer to be empty or full

Bounded Buffer Cont.

- Have the following:
  - buffer of size n (i. e., char buffer[n])
  - one producer thread
  - one consumer thread
- Locks are inappropriate here
  - if producer grabs lock, must release it if buffer is full
  - producer and consumer often access distinct locations
    - can be concurrent!
- Need something more general

Bounded Buffer, one slot buffer (shared) buf = NULL initially

Thread 1:

buf = data

Thread 2:

result = buf

We want the result in thread 2 to always be equal to “data”, not NULL

Bounded Buffer, one slot buffer (shared) buf = NULL initially

Thread 1:

CSEnter( )

buf = data

CSExit( )

Thread 2:

CSEnter( )

result = buf

CSExit( )

This does not ensure result is "data". Why not?

Bounded Buffer, one slot buffer (shared) buf = NULL initially

Thread 1:

```
buf = data
```

Thread 2:

```
while (buf == NULL) ;
result = buf
```

This works, but only for one slot buffers.

Semaphores (Dijkstra)

- Semaphore is an object
  - contains a (**private**) value and 2 operations
- **Semaphore value must be nonnegative**
- P(s): <await (s > 0) s = s – 1>
  - Implementation: if value is 0, block; else decrement value by 1
- V(s): <s = s + 1>
  - Implementation: if at least one thread is blocked, wake up one thread; else value++
- Semaphores are “resource counters”

Semaphore use #1: Critical Sections

sem mutex := 1

entry( )

- P(mutex)

exit( )

- V(mutex)

- Semaphores are at least as powerful than locks
- For mutual exclusion, initialize semaphore to 1

Semaphore use #2: Implementing Thread Create/Join

sem implJoin := 0

threadExit( )

- V(implJoin)

threadJoin( )

- P(implJoin)

- Semaphores are more powerful than locks
- Note here the semaphore is initialized to 0

Semaphore use #3: Bounded Buffer, (shared) char buf = NULL (shared) sem empty = 1, full = 0

Thread 1 (producer):

```
while (1) {
  P(empty)
  buf = data
  V(full)
}
```

Thread 2 (consumer):

```
while (1) {
  P(full)
  result = buf
  V(empty)
}
```

This finally does what we want (though it's only single slot)!

Notes on single slot bounded buffer

- Semaphores empty and full are *binary semaphores*
  - Their values are restricted to{0,1}; *general semaphores* simply have a nonnegative value
    - Note that **the programmer** is ensuring that the values are restricted to{0,1} (**not** the semaphore mechanism).
- Further, empty and full are *split binary semaphores*
  - **At most one** of empty or full can have value 1 at any point in time

Split binary semaphores

- Important because:
  - Split binary semaphores guarantee mutual exclusion if every execution path starts with a P on one of the semaphores and ends with a V on another
  - Of course, one of them must have an initial value 1 or deadlock occurs
  - We will talk more about this in the Readers/Writers problem (later in this unit)

Bounded Buffer, Multiple Slots
(1 producer, 1 consumer)
“+” indicates modulo arithmetic

char buf[n], int front := 0, rear := 0

sem empty := n, full := 0

Producer( )

```
while (1) {
  produce message m
  P(empty)
  buf[rear] := m;
  rear := rear “+” 1
  V(full)
}
```

Consumer( )

```
while (1) {
  P(full)
  m := buf[front]
  front := front “+” 1
  V(empty)
  consume message m
}
```

Bounded Buffer, Multiple Slots
(1 producer, 1 consumer)
“+” indicates modulo arithmetic

char buf[n], int front := 0, rear := 0

sem empty := n, full := 0

In the single slot bounded buffer, these values were 1

Producer( )

```
while (1) {
  produce message m
  P(empty)
  buf[rear] := m;
  rear := rear “+” 1
  V(full)
}
```

Consumer( )

```
while (1) {
  P(full)
  m := buf[front]
  front := front “+” 1
  V(empty)
  consume message m
}
```

Bounded Buffer, Multiple Slots
(Multiple producers and consumers)

char buf[n], int front := 0, rear := 0

sem empty := n, full := 0, mutexC := 1, mutexP := 1

```
Producer( )
  while (1) {
    produce message m
    P(empty); P(mutexP)
    buf[rear] := m;
    rear := rear "+" 1
    V(mutexP); V(full)
  }
```

```
Consumer( )
  while (1) {
    P(full); P(mutexC)
    m := buf[front]
    front := front "+" 1
    V(mutexC); V(empty)
    consume m
  }
```

Only difference from single producer and consumer is mutexP and mutexC

Dining Philosophers (Dijkstra)

- Round table
- Five philosophers sit at the table
- Five plates of spaghetti are at the table, one per philosopher
- Five forks are at the table
  - Each fork is between two philosophers
- Each philosopher alternately thinks and (wants to) eat
  - Must have two forks to eat; can only pick up the two nearest (left and right) forks

Dining Philosophers (incorrect)

```
sem fork[0:4] = {1}
Philosopher(i):
   P(fork[i]); P(fork[(i+1)%5]
   eat
   V(fork[i]); V(fork[(i+1)%5]
   think
```

Dining Philosophers (incorrect)

```
sem fork[0:4] = {1}
Philosopher(i):
   P(fork[i]); P(fork[(i+1)%5]
   eat
   V(fork[i]); V(fork[(i+1)%5]
   think
```

- Can deadlock (if all philosophers grab right fork before any grabs left fork)

Dining Philosophers (correct)

```
sem fork[0:4]={1}
Philosopher(i=0 to 3):
   P(fork[i])
   P(fork[(i+1)%5]
   eat
   V(fork[i])
   V(fork[(i+1)%5]
   think
```

```
Philosopher(4):
   P(fork[0])
   P(fork[4])
   eat
   V(fork[0])
   V(fork[4])
   think
```

Readers/Writers

- Given a database
  - can have multiple “readers” at a time
    - don’t ever modify database
  - can only have one “writer” at a time
    - will modify database
    - readers not allowed in while writer is
- Problem has many variations

Readers/Writers: Overconstrained “Solution”

- Put database in a critical section
- Technically satisfies the constraint that readers and writers never are in the database at the same time

Readers/Writers: Overconstrained "Solution"

- Put database in a critical section
- Technically satisfies the constraint that readers and writers never are in the database at the same time
  - Significant problem: readers cannot read concurrently!

Readers/Writers High-Level Solution, #1

sem rw = 1; int nr = 0

readEnter: <nr++; if (nr == 1) P(rw)>

readExit: <nr--; if (nr == 0) V(rw)>

writeEnter: <P(rw)>

writeExit: <V(rw)>

Readers/Writers Implementation #1

```
sem rw = 1, mutexR = 1; int nr = 0
readEnter:  P(mutexR); nr++;
            if (nr == 1) P(rw);
            V(mutexR)
readExit:  P(mutexR); nr--;
           if (nr == 0) V(rw);
           V(mutexR)
writeEnter: P(rw)
writeExit: V(rw)
```

Readers/Writers High-Level Solution, #2 (Passing the Baton)

int nr = 0, nw = 0

readEnter: <await (nw == 0) nr++>

readExit: <nr-->

writeEnter: <await (nr == 0 and nw == 0) nw++>

writeExit: <nw-->

Readers/Writers Solution #2 (Passing the Baton)

- Need mutual exclusion in both entry and exit
  - use mutex semaphore, initialized to one
- Keep state of database, enforce constraints
  - number of delayed readers and writers
  - number of readers and writers in database
  - Example: prevent nr, nw simultaneously > 0
- One semaphore blocks readers, different semaphore blocks writers (both initialized to 0)
- Readers going in can let other readers go in

Resource Allocation: Basic Idea

request: <await (ok to satisfy request) take units>

release: <return units>

- For this, can use passing the baton
- But what if we want general resource allocation
  - Where every thread can be in a unique class

Shortest Job Next (SJN)

- Have N jobs and 1 core
- All jobs have an id and a (known) execution time
- Each job J executes:

  Request(J.time, J.id)

  Wait for both:

  - Core to be available
  - J is the waiting job with the smallest J.time value

  Run to **completion** on core

  Release( )

Shortest Job Next (incorrect)

bool free = true

request(time, id): <await (free) free = false>

release( ): <free = true>

Shortest Job Next (incorrect)

bool free = true

request(time, id): <await (free) free = false>

release( ): <free = true>

- This doesn’t work, because there is no notion of ordering

Shortest Job Next: Private Semaphores

bool free = true; sem e := 1, b[0:n-1] = {0}; List l

One per thread

request(time, id)

```
P(e)
if (!free) {
  l.SortedInsert(time, id)
  V(e)
  P(b[id])
}
else
  free = false
V(e)
```

release( )

```
P(e)
if (!empty(l)) {
  int id = l.RemoveFront( )
  V(b[id])
}
else {
   free = true
   V(e)
}
```

Assume SortedInsert sorts by increasing time

Note: no delay counter needed; SortedInsert list makes it unnecessary