More Synchronization Problems, Wait-Free Synchronization

ECE595
Jan 30

Y. Charlie Hu

Roadmap

- Interprocess communication *with shared data*
  - Synchronization with locks, semaphores, condition var
  - Classic sync. problem 1: producer-consumer
  - Semaphore implementations (uniprocessor, multiprocessor)

Today:

- Classic sync. problems 2 & 3
- Wait-free synchronization

Classic Synchronization Problems

1. Producer-consumer problem (bounded buffer problem)

2. Readers-writers problem

3. Dining philosophers problem

Dining philosophers problem

Abstraction of concurrency-control problems

The need to allocate several resources among several processes while being deadlock-free and starvation-free

Classic Synchronization Problems

1. Producer-consumer problem (bounded buffer problem)

2. Readers-writers problem

3. Dining philosophers problem

Readers-Writers problem

Abstraction of concurrent access to shared data problem

- A data object is shared among multiple processes

Reader:

```
While (1) {

acq(mutex)
read();
rel(mutex)

}
```

Writer:

```
While (1){

acq(mutex)
write();        /* abstraction */
rel(mutex)

}
```


Readers-Writers problem

Abstraction of concurrent access problem

- A data object is shared among multiple processes
- Allow concurrent reads, but exclusive writes
  - Implication: need to move read() and write() outside Critical Sec
  - Can we do it using lock+ flags?
  - Can we use semaphore to count readers/writers?

Reader:

```
acq(mutex)
????
rel(mutex)

read();

acq(mutex)
???
rel(mutex)
```

Writer:

```
acq(mutex)
????
rel(mutex)

write();

acq(mutex)
???
rel(mutex)
```


Readers-Writers problem

Abstraction of concurrent access problem

- A data object is shared among multiple processes
- Allow concurrent reads, but exclusive writes
- Solution needs lock, semaphores, and counting!
- Constraints:
  - Writers can only proceed if there are no active readers/writers
    ➔ use semaphore OKtoWrite
  - Readers can proceed only if there are no active/waiting writers
    ➔ use semaphore OKtoRead
  - To keep track of how many are reading / writing / waiting
    ➔ use some shared variables, called *state variables*
  - Only one process manipulates state variable at once
    ➔ use a lock Mutex

Readers-Writers problem (cont)

- State variables:
  - AR = number of active readers
  - WR = number of waiting readers
  - AW = number of active writers
  - WW = number of waiting writers

  AW is always 0 or 1

  AR and AW can not both be non-zero
- Initialization:
  - OKtoRead = 0;
  - OKtoWrite = 0;
  - Mutex = 1;
  - AR = WR = AW = WW = 0;
- Scheduling: writers get preference

Readers-Writers problem (cont)

- Reader

```
acq(mutex)

rel(mutex)
wait(OKtoRead);
read necessary data;
```

```
acq(mutex);

rel(mutex);
```


Readers-Writers problem (cont)

- Writer

```
acq(mutex)

rel(mutex)
wait(OKtoWrite);
write necessary data;
```

```
acq(mutex);

rel(mutex);
```


Readers-Writers problem (cont)

- Reader

```
acq(mutex)
if (????) {
   signal(OKtoRead);
   AR ++;
} else {
   WR ++;
}
rel(mutex)
wait(OKtoRead);
read necessary data;
```

```
acq(mutex);
AR--;
if (????) {
   V(OKtoWrite);
   AW ++;
   WW --;
}
rel(mutex);
```

Readers-Writers problem (cont)

- Writer

```
acq(mutex)
if (????) {
  signal(OKtoWrite);
  AW ++;
} else {
  WW++;
}
rel(mutex);
wait(OKtoWrite);
write necessary data;
```

```
acq(mutex);
AW --;
if (????) {
  signal(OKtoWrite);
  AW ++;
  WW --;
} else while (????) {
  signal(OKtoRead);
  AR ++;
  WR --;
}
rel(mutex);
```


What happens if

- Reader enters and leaves system
- Write enters and leaves system
- Two writers enter system
- Two readers (a,b) enter system
- Writer(c) enters system and waits
- Reader(d) enters system and waits
- Readers(a,b) leave system, write(c) continues
- Write(c) leaves system, last reader(d) continues and leaves

Questions:

- In case of conflict between readers and writers, who gets priority?
  - Readers can get locked out
- Is the WW necessary in the writer's first if?
  - No: if there is a waiting writer, there must be an active writer or at least one active reader
- Can OKtoRead ever get greater than 1? What about OKtoWrite?
  - Yes, no
- Is the first writer to execute acq(mutex) guaranteed to be the first writer to access the data
  - No, waiting writers can get granted in any order

Roadmap

- Interprocess communication *with shared data*
  - Synchronization with locks, semaphores, condition var
  - Classic sync. problem 1: producer-consumer
  - Semaphore implementations (uniprocessor, multiprocessor)

Today:

- Classic sync. problems 2 & 3
- Wait-free synchronization

[lec5] Uniprocessor solution: disable interrupts!

```
void  wait(semaphore s)
{
   disable interrupts;
   if (s->count > 0) {
      s->count --;
      enable interrupts;
      return;
   }
   add(s->q, current_process);
   enable interrupts;
   /* implying re-dispatch */
}
```

```
void  signal(semaphore s)
{
   disable interrupts;
   if (isEmpty(s->q)) {
      s->count ++;
   } else {
      process = removeFirst(s->q);
      wakeup(process);
      /* put process on Ready Q */
   }
   enable interrupts;
}
```


[week3] Use TAS to implement semaphores on multiprocessor?

```
void  wait(semaphore s)
{
   disable interrupts;
   while (tsa(&lock,1)==1);
   if (s->count > 0) {
      s->count --;
      lock=0;
      enable interrupts;
      return;
   }
   add(s->q, current_process);
   lock=0;
   sleep(); /* re-dispatch */
   enable interrupts;
}
```

```
void  signal(semaphore s)
{
   disable interrupts;
   ???
   if (isEmpty(s->q)) {
      s->count ++;
   } else {
      thread = removeFirst(s->q);
      wakeup(process);
      /* put process on Ready Q */
   }
   ???
   enable interrupts;
}
```

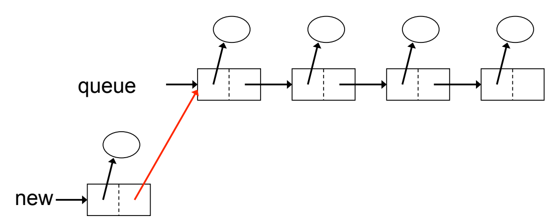
Do we still need to disable interrupts?

Wait-free Synchronization

- Finally we need tsa or ldl&stc anyway to implement sync. primitives (on multiprocessors)
- Can we design data structures in a way that allows safe concurrent accesses?
  - no OS-supplied mutual exclusion necessary
  - no possibility of deadlock
  - only using tsa / ldl^stc
  - no busy waiting

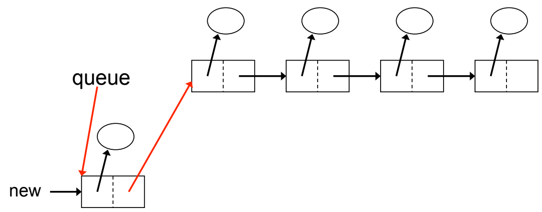
Simple example – Queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



Simple example – Queue insertion

```
typedef struct {
  QItem *item;
  QElem *next;
} QElem;
```



Singly-linked Queue Insertion

```
QElem *queue;

void Insert(item) {
  QElem *new = malloc(sizeof(QElem));
  new->item = item;
  new->next = queue;
  queue = new;
}
```


Wait-free Synchronization

- Design data structures in a way that allows safe concurrent accesses
  - no mutual exclusion (lock acquire & release) necessary
  - no possibility of deadlock
- Approach: use a single atomic operation to
  - commit all changes
    - move the shared data structure from one consistent state to another

Read-modify-write on CISC

- Most CISC machines provide atomic *read-modify-write* instruction
- Assume a test-and-set instruction

```
X = TAS(addr, old_value, new_value);

read value V at addr;
if (V == old_value) set it to new_value;
return V;
```

Singly-linked Queue Insertion using TSA?

```
QElem *queue;

void Insert(item) {
   QElem *new = malloc(sizeof(QElem));
   new->item = item;
   new->next = queue;
   queue = new;
}
```


Limitation

- Example only works for simple data structures where changes can be committed with *one store instruction*
- What about more complex data structures?

More General Approach

- Maintain a pointer to the "master copy" of the data structure
- To modify,
  1. remember current value of the master pointer
  2. copy shared data structure to a scratch location
  3. modify copy
  4. *atomically*:
     - verify that master pointer has not changed
     - write pointer to refer to new master
  5. if verification fails (another process interfered), start over at step 1
- Downside?
  - When does it work reasonably well?

Reading

- Chapter 6