Communication with Messages, Wait-Free Synchronization

ECE595
Sep 11
Y. Charlie Hu

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 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, counting, and semaphores!
- 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

```
wait(mutex)
if ((AW + WW) == 0) {
   signal(OKtoRead);
   AR ++;
} else {
   WR ++;
}
signal(mutex)

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

```
wait(mutex);
AR--;
if (AR == 0 && WW > 0) {
   /* no need to check AW == 0 */
   signal(OKtoWrite);
   AW ++;
   WW --;
}
signal(mutex);
```


Readers-Writers problem (cont)

- Writer

```
wait(mutex)
if ((AW + AR + WW) == 0) {
   signal(OKtoWrite);
   AW ++;
} else {
   WW++;
}
signal(mutex);

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

```
wait(mutex);
AW --;
if (WW > 0) {
   signal(OKtoWrite);
   AW ++;
   WW --;
} else while (WR > 0) {
   signal(OKtoRead);
   AR ++;
   WR --;
}
signal(mutex);
```

Dining philosophers problem

Abstraction of concurrency-control problems

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

Roadmap

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

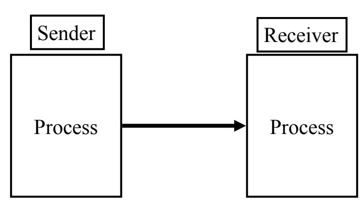
Today:

- Classic sync. problems 2 & 3
- Interprocess communication *with messages*
- Wait-free synchronization

Inter-process Communication with Messages

- Messages provide for communication without shared data
  - One process or the other owns the data, (guaranteed) never two at the same time
  - Think about usmail

Big Picture



Why use messages?

- Many types of applications fit into the model of processing a sequential flow of information
- Communication across address spaces – no side effects
  - Less error-prone
  - They might have been written by diff programmers who aren't familiar with code
  - They might not trust each other
  - They may not be running on different machines!
  - Examples?

Message Passing API

- Generic API
  - `send( mailbox, msg )`
  - `recv( mailbox, msg )`
- What is a mailbox?
  - A *buffer* where messages are stored between the time they are sent and the time when they are received
- What should "msg" be?
  - Fixed size msgs
  - Variable sized msgs: need to specify sizes

Buffering leads to design options

- When should send() return?
- When should recv() return?

Send

- *Fully Synchronous*
  - Will not return until data is received by the receiving process
- *Synchronous*
  - Will not return until data is received by the mailbox
  - Block on full buffer
- *Asynchronous*
  - Return immediately
  - Completion
    - Require the application to check status (appl polls)
    - Notify the application (OS sends interrupt)
  - Block on full buffer

Receive

- *Synchronous*
  - Return data if there is a message
  - Block on empty buffer
- *Asynchronous*
  - Return data if there is a message
  - Return status if there is no message (probe)

OS implementation

- What is the conceptual problem for OS implementation here?
  - Assume sender and receiver are on the same machine

Buffering

- No buffering
  - Sender must wait until the receiver receives the message
  - Rendezvous on each message
- Bounded buffer
  - Finite size
  - Sender blocks when the buffer is full
  - Receiver blocks when the buffer is empty
  - Using lock/condition variable (or semaphore)

Direct Communication

- Each process must name the sending or receiving process
- A communication link
  - is set up between the pair of processes
  - is associated with exactly two processes
  - exactly one link between each pair of processes

```
P: send( process Q, msg )
Q: recv( process P, msg )
```

Producer-Consumer Problem with Message Passing

```
Producer(){
  while (1) {
    …
    produce item
    …
    send( consumer, item);
  }
}
```

```
Consumer(){
  while (1) {
    recv( producer, item );
    …
    consume item
    …
  }
}
```


Indirect Communication

- Use a "mailbox" or "ports" to allow many-to-many communication
  - Mailbox typically owned by the OS
    - Requires open/close a mailbox before allowed to use it
- A "link"
  - is set up among processes only if they have a shared mailbox
  - Can be associated with more than two processes

```
P: open (mailbox); send( mailbox, msg);
   close(mailbox)
Q: open (mailbox); recv( mailbox, msg );
   close(mailbox)
```


The big debate in parallel computing: Messaging vs. Sharing Data

- Two programming models are equally powerful
- But result in very different-looking programming styles

- Most people find shared-data programming easier to work with
  - Debugging?

- What about machines that do not share memory?
  - Can be simulated in software [SDSM – hot topic in 80-90's]
  - But often not as efficient as message passing

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
- Interprocess communication *with messages*
- Wait-free synchronization

[lec6] 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;
  sleep(); /* 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;
}
```


[lec6] 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;
}
```

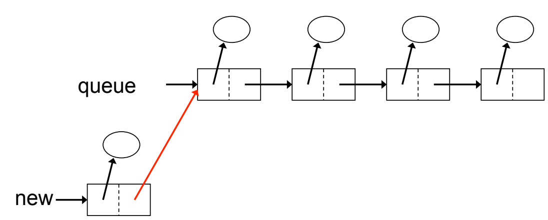
Do we still need to disable interrupts?

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 mutual exclusion necessary
  - no possibility of deadlock
  - only using tsa / ldl^stc
  - no busy waiting

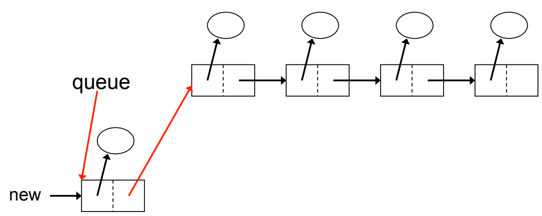
Simple example – Queue insertion

```
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

```
QElem *queue;

void Insert(item) {
   QElem *new = malloc(sizeof(QElem));
   new->item = item;
   do {
         new->next = queue;
   } while (tsa(&queue, new->next, new) !=
                  new->next);
}
```

Is this busy waiting a problem?

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?

Next week

- We will move on to CPU scheduling !