8: Classic Synchronization Problems and Deadlock

Last Modified:
6/14/2004 11:54:49 AM

	Value	Queue	"acquire" op	"release" op	"broadcast" or "release all" op
Lock	0/1	?	**Lock** Block till value = 0; set value = 1	**Unlock** Value = 0	No

Semaphore	INT	Y	**Wait**
			value—
			If value < 0
			Add self to queue
				Signal
				Value++
				If value <=0
				Wake up one
					No?
					While (getValue()<0){
					Signal
					}

Condition variable	N/A	Y	**Wait** Put self on queue	**Signal** If process on queue, wake up one	Easy to Support A signal all
Event	0/1	Y	**Wait** If Value = 0, put self on queue	**Signal** Value = 1 Wake up all	Default signal does
Monitor	0/1	Y	Call proc in monitor	Return from proc in monitor	No

Classical Synchronization Problems

- Bounded-Buffer Problem
  (also called Producer-Consumer)
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded Buffer Producer/Consumer

- Finite size buffer (array) in memory shared by multiple processes/threads
- Producer threads "produce" an item and place in in the buffer
- Consumer threads remove an item from the buffer and "consume" it
- Why do we need synchronization?
  - Shared data = the buffer state
  - Which parts of buffer are free? Which filled?
- What can go wrong?
  - Producer doesn't stop when no free spaces; Consumer tries to consume an empty space; Consumer tries to consume a space that is only half-filled by the producer; Two producers try to produce into same space; Two consumers try to consume the same space,...

Monitor Solution to Bounded-Buffer

```
container_t {
   BOOL free = TRUE;
   item_t   item;
}
monitor boundedBuffer {
   conditionVariable notAllFull;
   conditionVariable notAllEmpty;
   container_t buffer[FIXED_SIZE];
   int numFull = 0;
```


Monitor Solution to Bounded-Buffer

```
//monitor boundedBuffer cont

void producer (){
   while(allBuffersFull()){
      wait(notAllFull)
   }

   which = findFreeBuffer();
   which->free = FALSE;
   which->item = produceItem();
   numFull++

   signal(notAllEmpty);
 }
```

```
void consumer (){
   while(allBuffersEmpty()){
      wait(notAllEmpty)
   }

   which = findFullBuffer();
   consumeItem(which->item);
   which->free = TRUE;
   numFull--;

   signal(notAllFull);
   }

} //end Monitor
```

Semaphore Solution to Bounded-Buffer

```
semaphore_t mutex;
semaphore_t full;
semaphore_t empty;

container_t {
   BOOL free = TRUE;
   item_t   item;
}
container_t
   buffer[FIXED_SIZE];

void initBoundedBuffer{
   mutex.value = 1;
   full.value = 0;
   empty.value = FIXED_SIZE

}
```


Semaphore Solution to Bounded-Buffer

```
void producer (){
   container_t *which;
   wait(empty);
   wait (mutex);

   which = findFreeBuffer();
   which->free = FALSE;
   which->item = produceItem();

   signal (mutex);
   signal (full);
}
```

```
void consumer (){
   container_t *which;
   wait(full);
   wait (mutex);

   which = findFullBuffer();
   consumeItem(which->item);
   which->free = TRUE;

   signal (mutex);
   signal (empty);
}
```

•Can we do better? Lock held while produce/consume?

Readers/writers

- Shared data area being accessed by multiple processes/threads
- Reader threads look but don't touch
  - We can allow multiple readers at a time. Why?
- Writer threads touch too.
  - If a writer present, no other writers and no readers. Why?
- Is Producer/Consumer a subset of this?
  - Producers and consumers are both writers
  - Producer = writer type A; Consumer = writer type B and no readers
  - What might be a reader? Report current num full.

Semaphore Solution to Readers/ Writers (Reader Preference)

```
semaphore_t mutex;
semaphore_t okToWrite;
int         numReaders;

void init{
   mutex.value = 1;
   okToWrite.value = 1;
   numReaders = 0;
}
void writer (){
   wait(okToWrite);

   do writing (could pass
 in pointer to write function)

   signal(okToWrite);
}
```

```
void reader (){
   wait(mutex);
   numReaders++;
   if (numReaders ==1)
     wait(okToWrite); //not ok to write
   signal(mutex);

   do reading (could pass in pointer to
   read function)

   wait(mutex);
   numReaders--;
   if (numReaders == 0)
     signal(okToWrite); //ok to write again

   signal (mutex);
}
```

Can we do better? Fairness?

Monitor solution to readers/writers

This one favors writers; How? Can readers starve?

```
Monitor readersWriters {
   int numReaders = 0;
  BOOL writeInProgress =
   FALSE;

   int okToWriteQueued = 0;
   conditionVariable okToWrite;
   conditionVariable okToRead;

  void startRead() {
     while (writeInProgress ||
   okToWriteQueued){
        wait (okToRead);
     }
     numReaders++;
     signal(okToRead);
   }

  void finishRead(){
      numReaders--;
      if (numReaders == 0){
         signal(okToWrite);
      }
```

```
  void startWrite() {
      while (numReaders ||
         writeInProgress){

         okToWriteQueued++;
         wait(okToWrite);
         okToWriteQueued--;
      }
      writeInProgress = TRUE;
  }

  void finishWrite(){
    writeInProgress = FALSE;
    if (okToWriteQueued){
        signal(okToWrite);
    } else {
       signal(okToRead);
    }

   }
} //end monitor
```


Semaphore Solution to Readers/ Writers (Fair)

```
semaphore_t readCountMutex, incoming, next;
int         numReaders;
BOOL        writeInProgress, readInProgress;

void init{
   readCountMutex.value = 1;
   incoming.value = 1;
   next.value = 1;
   numReaders = 0;
   writeInProgress = FALSE;
  readInProgress = FALSE;
}
```

Semaphore Solution to Readers/ Writers (Fair)

```
void writer (){
    wait (incoming);
    wait(next);

   writeInProgress = TRUE;

   //Let someone else move on
   //to wait on next
   signal(incoming);

   do writing

   writeInProgress = FALSE;
   if (next.value == 0){
        signal (next);
    }
}
```

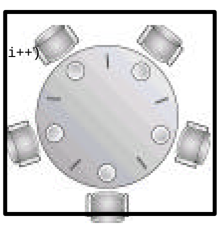
```
void reader (){
   wait(incoming);

   if (!readInProgress) {
       wait (next);
    }

   wait(readCountMutex);
   numReaders++;
   readInProgress = TRUE;
   signal(readCountMutex);

   //If next on incoming is
   //writer will block on next
   //If reader will come in
   signal(incoming);

   do reading

   wait(readCountMutex);
   numReaders--;
   if (numReaders == 0){
       readInProgress = FALSE;
       if (next.value == 0){
          signal (next);
       }
    }
    signal(readCountMutex);
```


Remember

- Game is obtaining highest possible degree of concurrency and greatest ease of programming
- Tension
  - Simple and high granularity locks easy to program
  - Simple and high granularity locks often means low concurrency
- Getting more concurrency means
  - Finer granularity locks, more locks
  - More complicated rules for concurrent access

Dining-Philosophers Problem

```
//array of chopsticks, chopstick i is
to the right of philosopher i

semaphore_t
chopstick[NUM_PHILOSOPHERS];


void init(){

   for (i=0; i< NUM_PHILOSOPHERS; i++)

        chopstick[i].value = 1;

}
```



Semaphore Solution to Dining Philosophers

Problem?

philosopher 0 gets chopstick 0
philosopher 1 gets chopstick 1
....
philosopher N gets chopstick N

Deadlock!   Solution?

```
void philosophersLife(int i){
    while (1) {
       int rightChopstick, leftChopstick;
       think();

       //figure out which chopsticks I need
       rightChopstick = i;
       leftChopstick = i-1+ NUM_PHILOSOPHERS % NUM_PHILOSOPERS;

       //grab chopsticks
       wait(chopstick[rightChopstick])
       wait(chopstick[leftChopstick]);

      eat();

       //putdown chopsticks
       signal(chopstick[ rightChopstick])
       signal(chopstick[ leftChopstick]);


    }
}
```


Deadlock

- Deadlock exists in a set of processes/threads when all processes/threads in the set are waiting for an event that can only be caused by another process in the set (which is also waiting!).
- Dining Philosophers is a perfect example. Each holds one chopstick and will wait forever for the other.

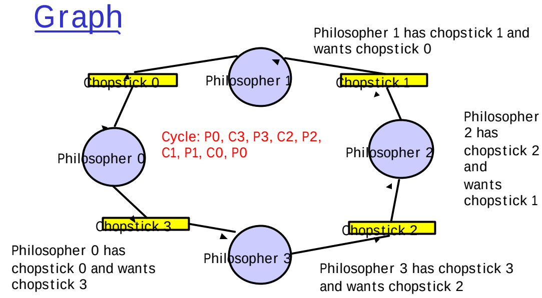
Fixing Dining Philosophers

- Make philosophers grab both chopsticks they need atomically
  - Maybe pass around a token (lock) saying who can grab chopsticks
  - Get a global lock before can lock any chopsticks
- Make a philosopher give up a chopstick
- Others?

Resource Allocation Graph

- Deadlock can be described through a resource allocation graph
- Each node in graph represents a process/thread or a resource
- An edge from node P to R indicates that process P had requested resource R
- An edge from node R to node P indicates that process P holds resource R
- If graph has cycle, deadlock *may* exist. If graph has no cycle, deadlock **cannot** exist.

Cycle in Resource Allocation Graph




```
wait(chopstick[i])
wait(chopstick[(i-1) % NUM_PHILOSOPHERS])
```


Better Semaphore Solution to Dining Philosophers

Always wait for low chopstick first

```
void philosophersLife(int i){
   while (1) {
      think();
      if ( rightChopstick < leftChopstick}{
           wait(chopstick[rightChopstick]);
           wait(chopstick[leftChopstick]);
      } else {
           wait(chopstick[leftChopstick]);
           wait(chopstick[rightChopstick]);
      }

      eat();

      signal(rightChopstick);
      signal(leftChopstick);

   }
}
```

Why better?

One philosopher reaches right first when all others reach left

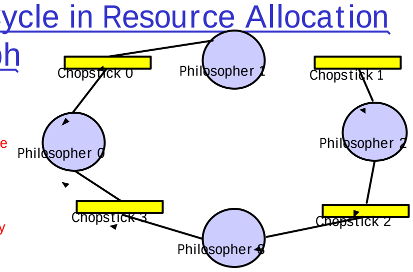
Two philsophers reach for same one and one will lose; The one who wins will get both chopsticks and finish – allowing everyone to finish eventually

No circular wait! No deadlock!!

No Cycle in Resource Allocation Graph

No cycle!
No deadlock!

This is not the only possible outcome. P1 could get chopstick 0 first, similarly P3 could get chopstick 3 first. Regardless, someone will get both their chopsticks and be able to finish




```
if ( rightChopstick < leftChopstick}{
           wait(chopstick[rightChopstick]);
           wait(chopstick[leftChopstick]);
} else {
           wait(chopstick[leftChopstick]);
           wait(chopstick[rightChopstick]);
}
```


Conditions for Deadlock

- Deadlock can exist only if the following four conditions are met;
  - Mutual Exclusion – some resource must be held exclusively
  - Hold and Wait – some process must be holding one resource and waiting for another
  - No preemption – resources cannot be preempted
  - Circular wait – there must exist a set of processes (p1,p2, …pn) such that p1 is waiting for p2, p2 is waiting for p3, … pn is waiting for p1
- All these held in the Dining Philosopher's first "solution" we proposed

Deadlock Prevention

- Four necessary and sufficient conditions for deadlock
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait
- Preventing mutual exclusion isn't very helpful. If we could allow resources to be used concurrently then we wouldn't need the synchronization anyway!
- Preventing the others?

Preventing Hold and Wait

- Do not allow processes to hold a resource when requesting others
  - Make philosophers get both chopsticks at once
  - Window's WaitForMultipleObjects
- Make processes ask for all resources they need at the beginning
  - Disadvantage: May not need all resources the whole time
  - Can release them early but must hold until used
- Make processes release any held resources before requesting more
  - Hard to program!

Preventing No Preemption

- Preemption (have to love those double negative ☺)
- Allow system to take back resources once granted
  - Make some philosopher give back a chopstick
- Disadvantage: Hard to program
  - System figures out how to take away CPU and memory without breaking programmer's illusion
  - How do you take away access to an open file or a lock once granted?? Would need API to notify program and then code to deal with the removal of the resource at arbitrary points in the code
    - Checkpoint and Rollback?

Preventing Circular wait

- Impose an ordering on the possible resources and require that processes request them in a specific order
- How did we prevent deadlock in dining philosophers?
  - Numbered the chopsticks
  - Made philosophers ask for lowest number chopstick first
- Disadvantage:
  - Hard to think of all types of resources in system and number them consistently for all cooperating processes
  - I use a resource X and Y , you use resource Y and Z and W, someone else uses W, T, R – which is resource 1? (shared files, databases, chopsticks, locks, events, …)
  - For threads in the same process or closely related processes often isn't that bad

Prevention vs Avoidance

- Both actually prevent deadlock
  - Deadlock Prevention does so by breaking one of the four necessary conditions
  - Deadlock Avoidance allows processes to make any request they want (not constrained in ways so as to break one of the four conditions) *as long as* they declare their maximum possible resource requests at the outset
- Both can deny resource requests that would not actually lead to deadlock in practice
  - Philosophers may never get into deadlock at all even with no intervention
  - Likelihood? How long do they think? How long eat?

Deadlock avoidance

- Say we don't want to write the code such that it is impossible to deadlock could still prevent deadlock by having the system examine each request and only grant if deadlock can be avoided
- Processes declare maximum resources they may ever request at the beginning
- Then during execution, system will only grant a request if it can ensure that all processes can run to completion without deadlock

Grant a resource?

- Consider a set of processes P1, P2, ..Pn which each declare the maximum resources they might ever request
- When Pi actually requests a resource, the system will grant the request only if the system could grant Pi's maximum resource requests with the resource currently available plus the resources held by all the processes Pj for j < I
- May need P1 to complete then P2 all the way to Pi but Pi can complete

Banker's Algorithm

- Decide whether to grant resource (loan or invest money, give a philosopher a chopstick, allow process to obtain a lock, …)
- Let there be P processes and R resources; Keep track of
  - Number of units of each resource available
  - Maximum number of units of each resource that each process could request
  - Current allocation of each resource to each process
- Real bankers cannot return money to everyone at once
  - Have a reserve requirement and rely on federal gov't to bail them out (FDIC)
  - Play odds on who will return money
  - Also bankers typically loan one processes resource to another; OS starts out owning the resources not borrowing

Banker's Algorithm

```
unsigned available[R];
unsigned allocation[P][R];
unsigned maximum[P][R];

startProcess(unsigned p){
  for (i=0; i< R; i++){
     maximum[p][i] = max number of resource i
  that process p will need at one time;
  }
}
```


Banker's Algorithm

```
BOOL request(unsigned p, unsigned r){
   if (allocation[p][r] + 1 > maximum[p][r]){
       //p lied about its max
       return FALSE;
   }

   if (available[p][r] == 0){
       //can't possibly grant; none available
       return FALSE;
   }

   if (canGrantSafely(p, r))
       allocation[p][r]++;
       available[r]--;
       return TRUE;
   } else {
       return FALSE;
   }
}
```


Banker's Algorithm

```
BOOL canGrantSafely(unsigned p, unsigned r){
   unsigned free[R];
   unsigned canFinish[P];

   for (j=0; j< R; j++) free[j] = available[j];
   for (i=0; i< P; i++) canFinish[i] = FALSE;

    lookAtAll: for (i=0; i< P; i++){
     allCanFinish = TRUE;
     if (!canFinish[i])
        allCanFinish = FALSE;
        couldGetAllResources = TRUE;
        for (j=0; j< R; j++){
           if (maximum[i][j] - allocation[i][j] >  free[j]){
                 couldGetAllResources = FALSE;
           }
        }
       if (couldGetAllResources){
           canFinish[i] = TRUE;
           for (i=0; i< R; i++) free[j] += allocation[i][j];
        }
     }
   } //for all processes

        if (allCanFinish) {
             return TRUE;
        }else {
             goto lookAtAll;
        }
     }
```


Avoidance vs Prevention

- Typically, system does avoidance; Programmer does prevention
- Deadlock avoidance usually results in higher resource allocation by allowing more combinations of resource requests to proceed than deadlock prevention
  - Not always – depends on ratio of processes maximum resource demands to average resource demands

If don't prevent deadlock?

- If don't prevent deadlock - either deadlock prevention or deadlock avoidance)- then how will the system deal with deadlock if (when!) it occurs:
- Two choices
  - Enable the system to detect deadlocks and if it does recover
  - Hope they never happen and rely on manual detection and recovery ("darn my process is hung again..kill process")
- Dining Philosophers?
  - Force a philosopher to put down a chopstick = preemption
  - Kill a philosopher? (sounds a bit brutal)
  - Kill all philosophers?

Deadlock Detection

- If don't want to ever deny requests when have resources to grant them, then deadlock may occur

```
BOOL request(unsigned p, unsigned r){
   if(available[p][r] >  0){
        allocation[p][r]++;
        available[r]--;
        return TRUE;
   } else {
        return FALSE;
   }
}
```


Deadlock Detection Algorithm

```
BOOL deadlockHasOccured(unsigned p, unsigned r)
{
   unsigned work[R];
   unsigned canFinish[P];

   //initialization
   for (j=0; j< R; j++) work[j] = available[j];
   for (i=0; i< P; i++){
        numResourcesAllocated = 0;
         for (j=0; j< R; j++) {
            numResourcesAllocated += allocation[i][j];
        }
        if (numResourcesAllocated == 0){
            canFinish[i] = TRUE; //can't be deadlocked if no hold and
   wait
        } else {
            canFinish[i] = FALSE; //don't know if this one is
   deadlocked
        }
   }
 .. .. ..
```


Deadlock Detection Algorithm

```
  tryToFinishOne: for (i=0; i< P; i++){
      finishedSomeoneThisTime = FALSE;
      allFinished = TRUE;
      if (!canFinish[i]){
          allFinished = FALSE;
           if ( (i != p) || (work[r] > 1) ) ) {
                canFinish[i] = TRUE;
                finishedSomeoneThisTime = TRUE;
                for (j=0; j< R; j++) work[j] += allocation[i][j];
           }
         }
   }
   if (allFinished){
         return FALSE; //no deadlock
   } else {
         if (! finishedSomeoneThisTime){
            return TRUE; //deadlock for pi st canFinish[i] == FALSE
         } else {
             goto tryToFinishOne;
         }
   }
} //end deadlockHasOccured
```


Running deadlock detection?

- Unlike with deadlock avoidance algorithm have choice of when to run
- If run on every resource request approaches avoidance
  - No sense of maximum resource requirement though
- Deciding how often
  - How often is deadlock likely to occur?
  - How many processes will be affected?
  - When CPU utilization drops below X%? (Overall or just for some processes? What if spin locks?)
  - What are signs that it might be good to run deadlock detection algorithm?

Recovery from Deadlock

- If system detects deadlock, what can it do to break the deadlock
- What do people do after manual detection?
  - Kill a process (es)
  - Reboot the system

Recovering from deadlock

- How many?
  - Abort all deadlocked processes
  - Abort one process at a time until cycle is eliminated (If one doesn't resolve deadlock, wait till deadlock detection algorithm runs again? Specifically run again with assumption that one of the processes is dead?)
- Which ones?
  - Lowest priority with canFinish = FALSE?
  - One that has been running the least amount of time (less work to redo)
  - Process that hasn't been killed before? Anyway to tell?

Prevention vs Avoidance vs Detection

- Spectrum of low resource utilization
  - Prevention gives up most chances to allocate resources
  - Detection always grants resource if they are available when requested
- Also spectrum of runtime "overhead"
  - Prevention has very little overhead; programmer obeys rules and at runtime system does little
  - Avoidance uses banker's algorithm (keep max request for each process and then look before leap)
  - Detection algorithm basically involves building the full resource allocation graph
  - Avoidance and detection algorithms both $O(R*P^2)$

-43

Real life?

- Most used prevention technique is resource ordering – reasonable for programmers to attempt
- Avoidance and Detection to expensive
- Most systems use manual detection and recovery
  - My process is hung – kill process
  - My machine is locked up – reboot
- Write code that deadlocks and run it on Linux and on Windows – what happens?

-44

Outtakes

-45

Gridapp

- What would it be like to do deadlock avoidance/detection for gridapp?
- For avoidance:
  - Would have to declare it's max usage each time through the loop for a thread or max usage would be the whole grid and get no concurrency?
- For detection, that would be be cool

-46

Dining Philosophers Example

```
monitor diningPhilosophers
{
 enum State{thinking, hungry, eating};

 State moods[NUM_PHILOSOPHERS];
 conditionVariable self[NUM_PHILOSOPHERS];

 void pickup(int i);
 void putdown( int i) ;
 void test(int i) ;
 void init() {
     for (int i = 0; i < NUM_PHILOSOPHERS; i++)
           state[i] = thinking;
 }
}
```

-47

Dining Philosophers

```
void pickupChopsticks(int i) {
     state[i] = hungry;
     test[i];
     if (state[i] != eating)
          self[i].wait();
}

void putdownChopsticks(int i) {
     state[i] = thinking;
     // test left and right neighbors
     test((i+ (NUM_PHILOSOPHERS-1 ) ) %
NUM_PHILOSOPHERS);
     test((i+1) % NUM_PHILOSOPHERS);
}
```

-48

Dining Philosophers

```
void test(int i) {
    if ( (state[(I + NUM_PHILOSOPHERS -1) %
NUM_PHILOSOPHERS] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % NUM_PHILOSOPHERS] != eating)) {
         state[i] = eating;
         self[i].signal();
    }
}
```

-49

Dining Philosophers

```
void philosophersLife(int i) {
    while(1){
         think();
         pickupChopticks();
         eat();
         putdownChopsicks();
    }
}
```

-50

Semaphore Solution to Readers/ Writers (Writer Preference)

```
semaphore_t mutex1, mutex2;
semaphore_t writePending, readersBlock, writersBlock;//☺
   ;
int         numReaders, numWriters;

void init{
   mutex1.value = 1;
   mutex2.value = 1;
   writePending.value = 1;
   readersBlock.value = 1;
   writersBlock.value = 1;
   numReaders = 0;
   numWriters = 0;
}
```

-51

Semaphore Solution to Readers/ Writers (Writer Preference)

```
void writer (){
   wait(mutex2);
    numWriters++;
   if (numWriters ==1){
       wait( readersBlock );
   }
   signal(mutex2);

   wait(writersBlock);
   do the writing
   signal(writersBlock);

   wait (mutex2);
   numWriters --;
   if (numWriters == 0){
     signal(readersBlock);
   }
   signal(mutex2);
}
```

```
void reader (){
   wait(writePending);
   wait ( readersBlock );
   wait (mutex1);
   numReaders ++;
   if (numReaders == 1){
     wait(writersBlock);
   }
   signal(mutex1);

   signal(readersBlock);
   signal(writePending);

   do reading

   wait (mutex1);
   numReaders --;
   if (numReaders ==0){
     signal(writersBlock);
   }
   signal(mutex1);
}
```

First writer waits in line with the readers;
Other writers wait with writers

-52

Other Classic Synchronization Problems

- Sleepy Barber
- Traffic lights for two lane road through a one lane tunnel (McNutt ch8 + 9)

-53