Barriers, Deadlocks & Dining Philosophers

Azza Abouzied

From Last Class: Semaphores

Semaphores: Initialization

```
struct semaphore{
    int count;
    int lock;
};

void sema_init(semaphore *s, int count){
    s->count = count;
    s->lock = 0;
}
```

Semaphores: down & up

```
//up or signal
void up(semaphore *s){

 //can be test & set
 acquire(s->lock);


  s->count++;


 release(s->lock);
}
```

```
//down or wait
void down(semaphore *s){
 while(1){
  acquire(s->lock);
  if(s->count > 0){
    s->count--;
    release(s->lock);
    return;
  }
  release(s->lock);
  yield();
 }
}
```

Producer/Consumer with Semaphores

```
semaphore empty, mutex, full;
sema_init(&empty, N);
sema_init(&mutex, 1);
sema_init(&full, 0);
```

```
void producer(){
  while(1){
    down(&empty);
    down(&mutex);
    push(m, b);
    up(&mutex);
    up(&full);
  }
}
```

```
void consumer(){
  while(1){
    down(&full);
    down(&mutex);
    pull(m, b);
    up(&mutex);
    up(&empty);
  }
}
```

The binary semaphore mutex does not have to be a semaphore!

What happens if we flip down(mutex) and down(full)?

Rule of thumb: Spin Lock or Mutex/Semaphore?

Spin Lock	Mutex/Semaphore
	
Short lock hold time Interrupt context locking Quick & Low overhead	Long lock hold time Process context locking Overhead of sleeping, maintaining wait queues, waking up threads can surpass lock time!

What about Order?

Initially, a = 1, b= 2. What can you say about the values of c & d?

Thread 1	Thread 2
a = 3	
b = 4	
	c = b
	d = a

Out of order processing

1. Processors can run read and write instructions out of order for performance (say by keeping the pipeline full), especially if there are no clear dependencies

2. A compiler can also reorder instructions when optimizing code

3. In SMP, a processor has no information on what is going on another processor

What about Order? Memory Barriers to the rescue!

Initially, a = 1, b= 2.

Thread 1	Thread 2
a = 3	
memory_barrier()	
b = 4	
	c = b
	memory_barrier()
	d = a

What does each barrier ensure?

Protects against c = 4 and d = 1 but not c = 2 and d = 3

What about Phases? Barriers to the rescue!

A **Pthreads** barrier defines a set of participating threads at program startup or barrier instantiation.

```
#define THREADS 10
pthread_barrier_t barr;
int main(){
    pthread_t thr[THREADS];
    for(int i = 0; i < THREADS; ++i)
        pthread_create(&thr[i], NULL,
            &entry_point, (void*)i));
    ...
}
void * entry_point(void *arg){
    /* First phase of computation */
    // Synchronization point
    int rc = pthread_barrier_wait(&barr);
    /* Second phase of compuation */
}
```

Implementing software barriers

Shared Memory Implementation

1. Each thread indicates its arrival at the barrier
2. Updates some shared state (counter++)
3. Busy-waits on shared state to determine when all the other threads have arrived (counter >= THREADS)
4. Once all threads arrived, each thread exits the busy loop

Other *dynamic* implementations exist.

Implicit barriers: Message passing systems that require global communication.

MapReduce/Hadoop is a popular distributed systems framework that uses barriers!


Deadlock

Deadlocks

Two or more tasks do not make progress because each is waiting for a resource held by another process

Related Concept: Starvation: Tasks wait indefinitely.

Some Examples

Memory

A wants 1.5 GB of memory, B wants 1.5 GB of memory.

System has 2 GB and A has 1, B has 1.

IO

A wants keyboard has screen.

B wants screen has keyboard

Bidirectional pipes

A outputs B, B outputs C, A consumes C.

Coffman’s Conditions for deadlock - 1971

All conditions must hold for a deadlock

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

Preventing Deadlocks

Break one condition and you prevent a deadlock

1. Mutual Exclusion

2. Hold and Wait

3. No Preemption

4. Circular Wait

Preventing Deadlocks

Break one condition and you prevent a deadlock

1. Mutual Exclusion
   Read only files; Resource Partitioning;
   Lock free data structures

2. Hold and Wait
   One resource only at a time; Consolidate into one; Request all at once (Issues?)

3. No Preemption
   Allow preemption helps with Priority Inversion Problems;
   Rollback to safe state

4. Circular Wait
   Require process to grab resources according to some order (Issues?)

Dealing with deadlocks in practice

Do Nothing

This will teach the user a lesson!

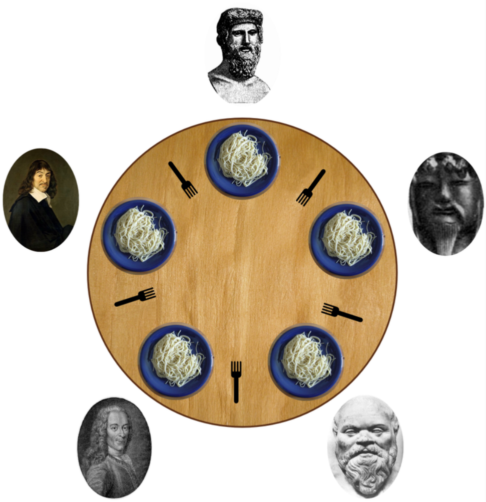
Reboot!

Kill Process

Bloodthirsty: kill everyone

Serial: Kill one at a time until there is no deadlock

Dealing with synchronization & deadlocks abstractly



An initial solution - will this work?

```
void philosopher() {
    while(1) {
        think();
        get_left_fork();
        get_right_fork();
        eat();
        put_left_fork();
        put_right_fork();
    }
}
```

Solution 2

Examine the handout. Will it work? Why so?

Make sure you read other concepts covered in the textbook like Monitors, Deadlock Avoidance, Banker's algorithm, Dining Philosopher's, etc.

Questions?