Concurrency issues
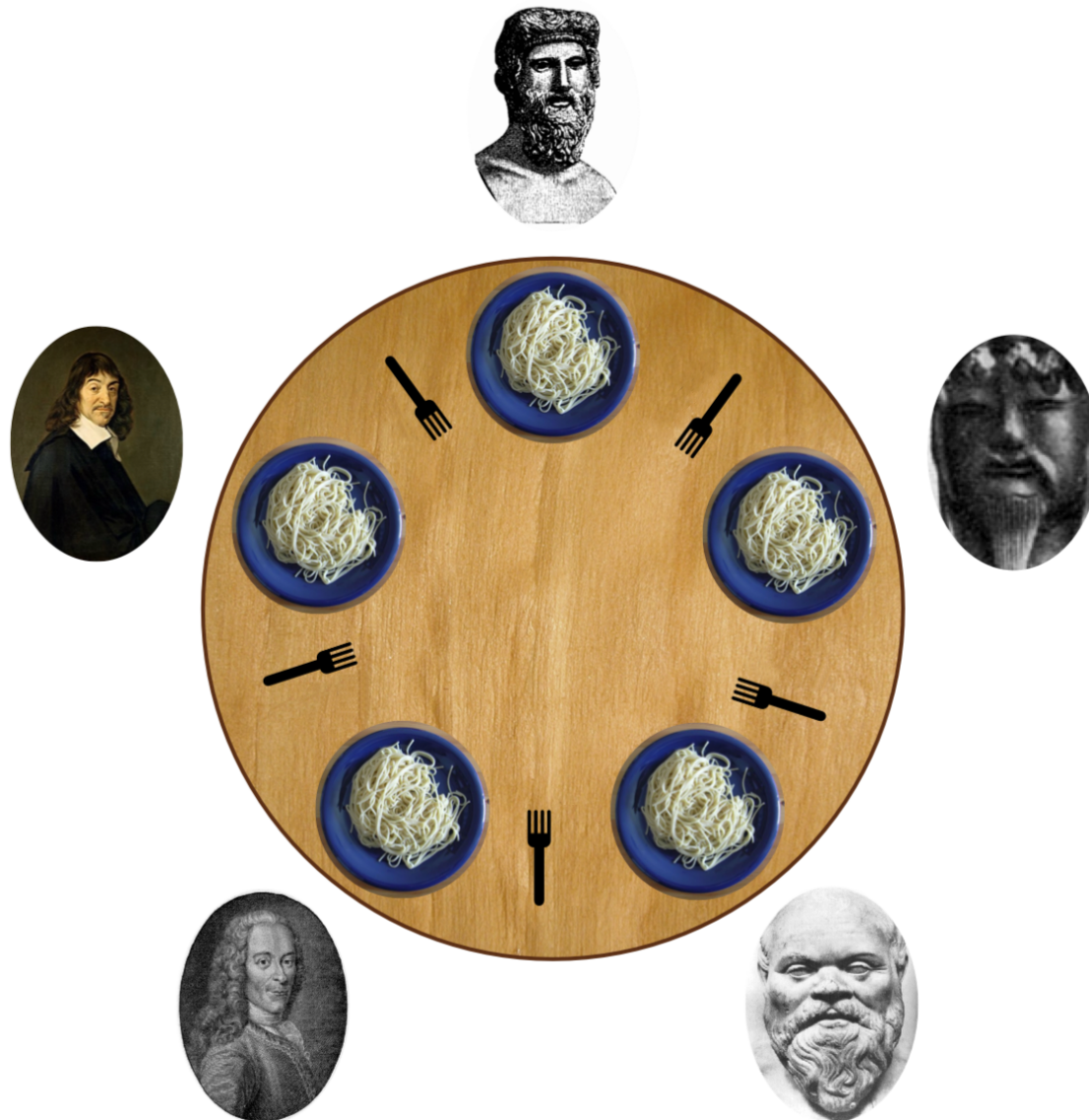
David Hovemeyer

4 December 2019

Outline

- Deadlocks

- Condition variables

- Amdahl's Law

- Atomic machine instructions, lock free data structures

Code examples on web page: synch2.zip

Deadlocks

Dining Philosopher's Problem

```
// Data structure
typedef struct {
  volatile int count;
  pthread_mutex_t lock, lock2;
} Shared;
```

```
// thread 1 critical section
pthread_mutex_lock(&obj->lock);
pthread_mutex_lock(&obj->lock2);
obj->count++;
pthread_mutex_unlock(&obj->lock2);
pthread_mutex_unlock(&obj->lock);
```

```
// thread 2 cricital section
pthread_mutex_lock(&obj->lock2);
pthread_mutex_lock(&obj->lock);
obj->count++;
pthread_mutex_unlock(&obj->lock);
pthread_mutex_unlock(&obj->lock2);
```

Modified shared counter program

```
// Data structure
typedef struct {
  volatile int count;
  pthread_mutex_t lock, lock2;
} Shared;
```

```
Acquire obj->lock,
then obj->lock2
```

```
// thread 1 critical section
pthread_mutex_lock(&obj->lock);
pthread_mutex_lock(&obj->lock2);
obj->count++;
pthread_mutex_unlock(&obj->lock2);
pthread_mutex_unlock(&obj->lock);
```

```
// thread 2 cricital section
pthread_mutex_lock(&obj->lock2);
pthread_mutex_lock(&obj->lock);
obj->count++;
pthread_mutex_unlock(&obj->lock);
pthread_mutex_unlock(&obj->lock2);
```

```
// Data structure
typedef struct {
  volatile int count;
  pthread_mutex_t lock, lock2;
} Shared;
```

```
Acquire obj->lock2,
then obj->lock
```

```
// thread 1 critical section
pthread_mutex_lock(&obj->lock);
pthread_mutex_lock(&obj->lock2);
obj->count++;
pthread_mutex_unlock(&obj->lock2);
pthread_mutex_unlock(&obj->lock);
```

```
// thread 2 cricital section
pthread_mutex_lock(&obj->lock2);
pthread_mutex_lock(&obj->lock);
obj->count++;
pthread_mutex_unlock(&obj->lock);
pthread_mutex_unlock(&obj->lock2);
```

```
$ make incr_deadlock
gcc -Wall -Wextra -pedantic -std=gnu11 -O2 -c incr_deadlock.c
gcc -o incr_deadlock incr_deadlock.o -lpthread
$ ./incr_deadlock
```

hangs indefinitely...

Deadlock

Use of blocking synchronization constructs such as semaphores and mutexes can lead to *deadlock*

In the previous example:

- Thread 1 acquires obj->lock and waits to acquire obj->lock2

- Thread 2 acquires obj->lock2 and waits to acqurie obj->lock

Neither thread can make progress!

Resource allocation graph

Resource allocation graph: ▌

- Nodes represent threads and lockable resources ▌

- Edges between threads and resources ▌

- Edge from thread to resource: thread has locked the resource ▌

- Edge from resource to thread: thread is waiting to lock the resource ▌

Cycle indicates a deadlock

Deadlock situation

Computer Systems Fundamentals: Concurrency issues
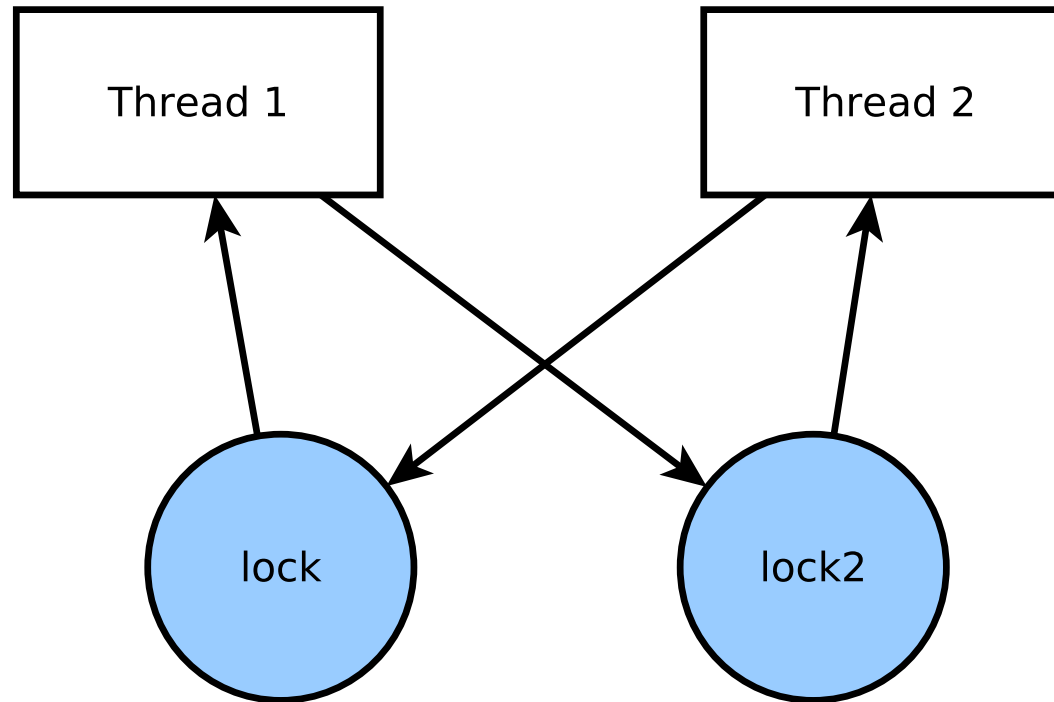
Deadlocks can only occur if ▮

- threads attempt to acquire multiple locks simultaneously, and ▮

- there is not a globally-consistent lock acquisition order ▮

Trivially, if threads only acquire one lock at a time, deadlocks can't occur

Maintaining a consistent lock acquisition order also works

Trivial self-deadlock

Can you spot the error in the following critical section?

```
pthread_mutex_lock(&obj->lock);
obj->count++;
pthread_mutex_lock(&obj->lock);
```

This mistake is easy to make because pthread_mutex_lock and pthread_mutex_unlock have very similar names

Another type of self-deadlock can occur if multiple functions have critical sections, and one calls another:

```
void func1(Shared *obj) {
    pthread_mutex_lock(&obj->lock);
    // critical section...
    pthread_mutex_unlock(&obj->lock);
}

void func2(Shared *obj) {
    pthread_mutex_lock(&obj->lock);
    // another critical section...
    func1(obj);
    pthread_mutex_unlock(&obj->lock);
}
```

A good approach to avoiding self-deadlock is:

- avoid acquiring locks in helper functions

- make ''higher-level'' functions (often, the ''public'' API functions of the locked data structure) responsible for acquiring locks

Example:

```
void highlevel_fn(Shared *obj) {
  pthread_mutex_lock(&obj->lock);
  helper(obj);
  pthread_mutex_unlock(&obj->lock);
}

void helper(Shared *obj) {
  // critical section...
}
```

Condition variables

Condition variables are another type of synchronization construct supported by pthreads

They allow threads to wait for a condition to become true: for example,

- Wait for queue to become non-empty

- Wait for queue to become non-full

- etc.

They work in conjunction with a mutex

Condition variable API

Data type: `pthread_cond_t`

Functions:

- `pthread_cond_init`: initialize a condition variable

- `pthread_cond_destroy`: destroy a condition variable

- `pthread_cond_wait`: wait on a condition variable, unlocking mutex (so other threads can enter critical sections)

- `pthread_cond_broadcast`: wake up waiting threads because condition may have been enabled

BoundedQueue data type:

```
typedef struct {
  void **data;
  unsigned max_items, count, head, tail;
  pthread_mutex_t lock;
  pthread_cond_t not_empty, not_full;
} BoundedQueue;
```

Creating a BoundedQueue:

```
BoundedQueue *bqueue_create(unsigned max_items) {
  BoundedQueue *bq = malloc(sizeof(BoundedQueue));
  bq->data = malloc(max_items * sizeof(void *));
  bq->max_items = max_items;
  bq->count = bq->head = bq->tail = 0;
  pthread_mutex_init(&bq->lock, NULL);
  pthread_cond_init(&bq->not_full, NULL);
  pthread_cond_init(&bq->not_empty, NULL);
  return bq;
}
```

Bounded queue example

Enqueuing an item:

```
void bqueue_enqueue(BoundedQueue *bq, void *item) {
  pthread_mutex_lock(&bq->lock);

  while (bq->count >= bq->max_items) {
    pthread_cond_wait(&bq->not_full, &bq->lock);
  }

  bq->data[bq->head] = item;
  bq->head = (bq->head + 1) % bq->max_items;
  bq->count++;

  pthread_cond_broadcast(&bq->not_empty);

  pthread_mutex_unlock(&bq->lock);
}
```

Bounded queue example

Enqueuing an item:

```
void bqueue_enqueue(BoundedQueue *bq, void *item) {
  pthread_mutex_lock(&bq->lock);

  while (bq->count >= bq->max_items) {
    pthread_cond_wait(&bq->not_full, &bq->lock);
  }

  bq->data[bq->head] = item;
  bq->head = (bq->head + 1) % bq->max_items;
  bq->count++;

  pthread_cond_broadcast(&bq->not_empty);

  pthread_mutex_unlock(&bq->lock);
}
```

Acquire mutex

Enqueuing an item:

```
void bqueue_enqueue(BoundedQueue *bq, void *item) {
  pthread_mutex_lock(&bq->lock);

  while (bq->count >= bq->max_items) {
    pthread_cond_wait(&bq->not_full, &bq->lock);
  }

  bq->data[bq->head] = item;
  bq->head = (bq->head + 1) % bq->max_items;
  bq->count++;

  pthread_cond_broadcast(&bq->not_empty);

  pthread_mutex_unlock(&bq->lock);
}
```

Wait for queue to become non-full

Bounded queue example

Enqueuing an item:

```
void bqueue_enqueue(BoundedQueue *bq, void *item) {
  pthread_mutex_lock(&bq->lock);

  while (bq->count >= bq->max_items) {
    pthread_cond_wait(&bq->not_full, &bq->lock);
  }

  bq->data[bq->head] = item;
  bq->head = (bq->head + 1) % bq->max_items;
  bq->count++;

  pthread_cond_broadcast(&bq->not_empty);

  pthread_mutex_unlock(&bq->lock);
}
```

Add item to queue

Enqueuing an item:

```
void bqueue_enqueue(BoundedQueue *bq, void *item) {
  pthread_mutex_lock(&bq->lock);

  while (bq->count >= bq->max_items) {
    pthread_cond_wait(&bq->not_full, &bq->lock);
  }

  bq->data[bq->head] = item;
  bq->head = (bq->head + 1) % bq->max_items;
  bq->count++;

  pthread_cond_broadcast(&bq->not_empty);

  pthread_mutex_unlock(&bq->lock);
}
```

> Wake up threads
> waiting for queue
> to be non-empty

Enqueuing an item:

```
void bqueue_enqueue(BoundedQueue *bq, void *item) {
  pthread_mutex_lock(&bq->lock);

  while (bq->count >= bq->max_items) {
    pthread_cond_wait(&bq->not_full, &bq->lock);
  }

  bq->data[bq->head] = item;
  bq->head = (bq->head + 1) % bq->max_items;
  bq->count++;

  pthread_cond_broadcast(&bq->not_empty);

  pthread_mutex_unlock(&bq->lock);
}
```

Release mutex

Principles for using condition variables: ▌

- Each condition variable must be associated with a mutex ▌

- Multiple condition variables can be associated with the same mutex ▌

- The mutex must be locked when waiting on a condition variable

  - pthread_cond_wait releases the mutex, then reacquires it
    when the wait is ended (by another thread doing a broadcast) ▌

- pthread_cond_wait must be done in a loop!

  - Spurious wakeups are possible, so waited-for condition must
    be re-checked ▌

- Use pthread_cond_broadcast whenever a condition *might*
  have been enabled

Amdahl's Law

Let's say you're parallelizing a computation: goal is to make the computation complete as fast as possible

Say that t_s is the sequential running time, and t_p is the parallel running time

Speedup (denoted S) is t_s/t_p

E.g., say that t_s is 10 and t_p is 2, then $S = 10/2 = 5$

Let P be the number of processor cores

In theory, speedup S cannot be greater than P

So, in the ideal case,
$$S = P = t_s/t_p$$
implying that
$$t_p = t_s/P$$

Note that $\lim_{P \to \infty} t_s/P$ is 0

- Meaning that throwing an arbitrary number of cores at a computation should improve performance by an arbitrary factor

- That would be great!

When speedup $S = P$, we have perfect *scalability*

This is difficult to achieve in practice because parallel computations generally have some *sequential overhead* which cannot be (easily) parallelized:

- Divide up work

- Synchronization overhead

- Combining solutions to subproblems

- etc.

Say that, for some computational problem, the proportions of inherently sequential and parallelizable computation are w_s and w_p, respectively

Note that $w_s + w_p = 1$, so $w_p = 1 - w_s$

Normalized sequential execution time t_s:

$$t_s = 1 = w_s + w_p$$

Parallel execution time using P cores:

$$t_p = w_s + \frac{w_p}{P} = w_s + \frac{1 - w_s}{P}$$

Speedup using P cores:

$$S = \frac{t_s}{t_p} = \frac{1}{w_s + \frac{1-w_s}{P}}$$

As $P \to \infty$, $\frac{1-w_s}{P} \to 0$, so

$$S \to \frac{1}{w_s}$$

Let's say $w_s = .05$: maximum speedup is $1/.05 = 20$

- This is regardless of how many cores we use!

Amdahl's Law assumes that the proportion of inherently sequential computation (w_s) is independent of the problem size▮

Gustafson-Barsis's Law: for some important computations, the proportion of parallelizable computation scales with the problem size

- These are called *scalable* computations▮

- Such computations can realize speedups proportional to P for a large number of processors

Atomic machine instructions

Computer Systems Fundamentals: Concurrency issues

We noted previously that incrementing an integer variable (obj->count++) is not atomic

However, modern processors typically support *atomic machine instructions*

- These are atomic even when used on shared variables by multiple threads

Various ways to use these:

- Assembly language

- Compiler intrinsics

- Language support

Atomic machine instructions

Typical examples of atomic machine instructions:

- Increment

- Decrement

- Exchange (swap contents of two variables)

- Compare and swap (compare register and variable, if equal, swap variable's contents with another value)

- Load linked/store conditional (load from variable, store back to variable only if variable wasn't updated concurrently)

x86-64 memory instructions can have a *lock* prefix to guarantee atomicity, e.g.:

```
    .globl atomic_increment
  atomic_increment:
    lock; incl (%rdi)
    ret
```

Calling from C code:

```
  void atomic_increment(volatile int *p);

  ...

  atomic_increment(&obj->count);
```

See incr_atomic.c and atomic.S

gcc has a number of intrinsic functions for atomic operations

E.g., atomic increment:

```
__atomic_fetch_add(&obj->count, 1, __ATOMIC_ACQ_REL);
```

See incr_atomic2.c

Atomic increment using C11 _Atomic

The C11 standard introduces the _Atomic type qualifier

Defining shared counter type:

```
typedef struct {
    _Atomic int count;
} Shared;
```

Incrementing the shared counter:

```
obj->count++;
```

See incr_atomic3.c

Atomic machine instructions can be the basis for *lock-free data structures*

Basic ideas:

- Data structure must always be in a valid state!

- Transactional: mutators speculatively create a proposed update and attempt to commit it using compare-and-swap (or load linked/ store conditional)

  - Retry transaction if another thread committed an update concurrently, invalidating proposed update

Issue: waits and wake-ups are not really possible

- E.g., when trying to dequeue from an empty queue, can't easily wait for item to be available, calling thread must spin