Section 13: Intro to Distributed Systems

November 25, 2015

Contents

1 Warmup **2**
- 1.1 General's Paradox . . . 2

2 Problems **2**
- 2.1 General's Paradox Continued . . . 2
- 2.2 Message Passing With Threads . . . 2
  - 2.2.1 Message Inbox Datastructure . . . 2
  - 2.2.2 Inbox Initialization . . . 3
  - 2.2.3 Message Sending Function . . . 3
  - 2.2.4 Message Sending Function . . . 3
- 2.3 Baking with Messages . . . 4

1 Warmup

1.1 General's Paradox

Problem setup:

1. There are two generals, standing on opposite mountains.
2. They are communicating via messenger to plan an attack for the next morning.
3. The messengers who run between the two generals can be captured.

Let's say we're a spy in the valley, and we have seen the following exchange taking place:

```
General 1: Shall we attack at 11 AM tomorrow?
General 2: 11 AM is OK for me.
General 1: I shall have my troops ready at 11 AM then. Will you?
General 2: Yes, I shall.
```

Can we conclude that the two generals will attack at 11 AM on the morning after this exchange takes place?

No, we cannot conclude that they will attack. While we (a spy-in-the-middle) have seen the messages, we don't know if the acknowledgement from General 2 *did* make it back to General 1 successfully.

What does this example demonstrate about systems that require communication?

If the system that we use for communication is unreliable, then we cannot guarantee that the system can be used to synchronize two events.

2 Problems

2.1 General's Paradox Continued

How can we modify the protocol that the Generals use to ensure that both Generals attack at 11 AM the day after the exchange takes place? (Hint: can we change *how* they communicate?)

The reason we aren't sure whether the two Generals will be synchronized is because the messengers may be captured (the network is unreliable). To resolve this, we can allow a certain time window for a messenger to go from their base camp, to the base camp of the other general, and to then return. If the messenger doesn't return in this window, we will send a second (third, fourth, and so on...) messenger, who carries a replica of the message. By doing this until all messages go through and are acknowledged, we can turn an unreliable network into an approximately reliable network.

2.2 Message Passing With Threads

In lecture, we talked about message passing APIs, where threads/processes/computers communicate by sending and receiving messages. Let's come up with an example message passing implementation using the `pthread` or Pintos threading APIs.

2.2.1 Message Inbox Datastructure

```
struct inbox_t {
  pthread_mutex_t lock;
  pthread_cond_t cv;
  int msg_count;
  void ** msgs;
  int * msgs_size;
};
```

2.2.2 Inbox Initialization

```
void inbox_init(inbox_t * inbox) {
  pthread_mutex_init(&(inbox->lock));
  pthread_cond_init(&(inbox->cv));
  inbox->msg_count = 0;
}
```

2.2.3 Message Sending Function

```
int msg_send(inbox_t * inbox, void * msg, size_t msg_size) {
  // acquire lock on inbox
  pthread_mutex_lock(&(inbox->lock));

  // try to allocate larger message and size buffer
  inbox->msg_size = (int *) realloc((void *) inbox->msg_size,
                                    (inbox->msg_count + 1) * sizeof(int));
  inbox->msgs = (void **) realloc((void *) inbox->msgs,
                                  (inbox->msg_count + 1) * sizeof(void *));

  // did malloc fail? if so, return with -1
  if (inbox->msg_size == 0 || inbox->msgs == 0) {
    return -1;
  }

  // copy message info
  inbox->msg_size[inbox->msg_count] = msg_size;
  inbox->msgs[inbox->msg_count] = msg;
  inbox->msg_count++;

  // signal to wake waiting thread and unlock
  pthread_cond_signal(&(inbox->cv));
  pthread_mutex_unlock(&(inbox->lock));
}
```

2.2.4 Message Sending Function

```
int msg_recv(inbox_t * inbox, void * msg) {
  size_t msg_size;

  // acquire inbox lock
  pthread_mutex_lock(&(inbox->lock));
```

```
  // are there any messages in the inbox? if no, then wait.
  while(inbox->msgs == 0) {
    pthread_cond_wait(&(inbox->cv), &(inbox->lock));
  }

  // process a message
  inbox->msg_count--;
  msg_size = inbox->msg_size[inbox->msg_count];
  msg = inbox->msgs[inbox->msg_count];

  // reallocate message buffers to free newly unused memory
  // we do not need to check the return value of realloc, since a realloc
  // that decreases the size of the allocated block will never fail
  inbox->msg_size = (int *) realloc((void *) inbox->msg_size,
                                    inbox->msg_count * sizeof(int));
  inbox->msgs = (void **) realloc((void *) inbox->msgs,
                                  inbox->msg_count * sizeof(void *));

  // release inbox lock
  pthread_mutex_unlock(&(inbox->lock));
}
```

2.3 Baking with Messages

In a previous section, we went through a problem where we were trying to coordinate the baking of a cake between multiple people who only know a single skill (threads) using condition variables. Now, let's say that we are trying to solve the same problem, but our multiple people are spread across a network, and we are coordinating to cook the cake via messages. Let's use the message passing API we came up with in the last problem to build this.
A cake requires:

- 1 cake batter
- 2 eggs

Instructions:

1. Add ingredients to bowl
2. Heat bowl (it's oven-safe)
3. Eat the cake, clean out the bowl, and go back to step 1

Requirements:

- Don't start heating the cake in the oven unless there are exactly the right number of ingredients in the bowl.
- Don't add raw ingredients to a currently-baking cake or a finished cake.
- Don't eat the cake unless it's done baking.
- Given enough time, the code should bake an unbounded number of cakes, and should never stop.

In this problem, if you are allocating memory, you can assume that `malloc` always succeeds.

```
inbox_t egg_adder_inbox, batter_adder_inbox, bowl_heater_inbox, eater_inbox;

// add message struct definitions here, if necessary
typedef struct heater_msg {
  int eggs_added;
  int batter_added;
};

void batterAdder()
{
  void * msg;
  heater_msg * msg_to_send;
  while(1) {
    msg_recv(&batter_adder_inbox, msg);
    free(msg);
    msg_to_send = (heater_msg *) malloc(sizeof(heater_msg));
    msg_to_send->eggs_added = 0;
    msg_to_send->batter_added = 1;
    msg_send(&bowl_heater_inbox, (void *) msg_to_send, sizeof(heater_msg));
  }
}

void eggBreaker()
{
  void * msg;
  heater_msg * msg_to_send;
  while(1) {
    msg_recv(&egg_adder_inbox, msg);
    free(msg);
    msg_to_send = (heater_msg *) malloc(sizeof(heater_msg));
    msg_to_send->eggs_added = 1;
    msg_to_send->batter_added = 0;
    msg_send(&bowl_heater_inbox, (void *) msg_to_send, sizeof(heater_msg));
  }
}

void bowlHeater()
{
  int eggs = 0, batter = 0;
  heater_msg * msg;
  void * msg;
  while(1) {
    while(eggs < 2 && batter < 1) {
      msg_recv(&bowl_heater_inbox, (void *) msg);
      eggs += msg->eggs_added;
      batter += msg->batter_added;
      free(msg);
    }
    msg_to_send = malloc(sizeof(int));
    msg_send(&eater_inbox, msg_to_send, sizeof(int));
```

```
  }
}

void cakeEater()
{
  void * msg, msg_to_egg_1, msg_to_egg_2, msg_to_batter;
  while(1) {
    msg_recv(&eater_inbox, msg);
    free(msg);
    msg_to_egg_1 = malloc(sizeof(int));
    msg_to_egg_2 = malloc(sizeof(int));
    msg_to_batter = malloc(sizeof(int));
    msg_send(&egg_adder_inbox, msg_to_egg_1, sizeof(int));
    msg_send(&egg_adder_inbox, msg_to_egg_2, sizeof(int));
    msg_send(&batter_adder_inbox, msg_to_batter, sizeof(int));
  }
}

int main(int argc,char *argv[])
{
  // Initialize mailboxes
  // Starts threads that communicate via mailboxes:
  //   - 1 batterAdder
  //   - 2 eggBreakers
  //   - 1 bowlHeater
  //   - 1 cakeEater
  // main() sleeps forever
}
```