CS452 Train Control Demo 2

Jacob Parker, Kyle Spaans

Monday November 29, 2010

1 Administrivia

1.1 How to use our program

To load our code, type "l ARM/StrombolOS/t2; g -c" at the reboot prompt. Type "help" for assistance.

1.2 MD5 Sums

```
ea89f074802cfde603e71d79cfb1f543  Build.sh
4536d16dcdf9c8aa6e88e7a89d45e529  build/orex.ld
969d8339502a5ef8367bef58d0188080  bwio/Makefile
9313c3b8d3a2e1d6d6f1162555ea3392  bwio/bwio.c
db166ea08b307131dc8ef10348c680b2  docs/README
ab564c846fe68831cea6372a98356797  docs/Project\_Proposal.tex
40f4f3cd6e750c354c5e0669cc79afa0  include/ANSI.h
d32dda3f6cd59b210c03d1ed8332c581  include/bwio.h
8fe4bcbf13f60d8dfc91ecbdc0605734  include/debug.h
ba884180a37f346c0ba8547c148bd4da  include/lock.h
2c909d66f6df2de289b8358a300c6ea4  include/tids.h
396d47c513a8d1763ec8524ef7043df3  include/ts7200.h
427aed49b53f52ce6185ed277e1046cb  kernel/boot.c
af775bc4ccad76e014e77551160af12f  kernel/boot.h
5bba63afffefa69c48ca4ee9ae7fc4cd  kernel/kernel.c
70c9044b63f139fc82dfc9c98202cb33  kernel/switch.h
60da756ed07348e576848ad2db5b9103  kernel/syscalls/awaitevent.c
f1460ffe80756a10326ea1f543d56b66  kernel/syscalls/create.c
4c445abfc7bbc947523149aa4ff8db69  kernel/syscalls/exit.c
88de93ad80f82c0c19068db925aa9a7b  kernel/syscalls/ksyscall.h
1fc352c47457bcc3483baf673453ca47  kernel/syscalls/myparenttid.c
7aea0293ec3376f3b609444439e0ffd8  kernel/syscalls/mytid.c
02a046f81c0b15aae6672817ba9c8fd7  kernel/syscalls/pass.c
e6193ca65075c3dc851866dfb593b5d7  kernel/syscalls/receive.c
62cfc465c6f3a121e6a1fc7ab37611fc  kernel/syscalls/reply.c
ad583b8bfa980bcd78c2666c0d993662  kernel/syscalls/send.c
ca28262728ec68a324728334cde3d85e  kernel/tasks.c
cf322c58d2b0ffca7960adb1731fc034  kernel/tasks.h
a499e8d19ee5410832196b18d02604cc  ktests/tests.c
55ff8b30fa5d99a344fe21c4f04efa10  ktests/tests.h
63b2297f222ea1793a7645e5594b6b5d  servers/cali.c
868bcfef3e8a1e98fa7e2d9614076d32  servers/cali.h
84a16f5ae19e3fd84c10cfe2ce60af39  servers/calibration.h
```

```
e7394261773af98b40e1363d90fa8b1e  servers/clock.c
1d1832fa77490954b4c3462fc0053131  servers/lock.c
7d2b05ec0779ca90badce6913bb8abe4  servers/names.c
1b884ba7141569e43c5e611cd5ca2b37  servers/notifier_clock.c
1c31285b50f4d5d80396fb988ac672f9  servers/notifier_uart1rx.c
304d3518e6d323b70013aa816fc26a08  servers/notifier_uart1tx.c
73f601af47ba67c47b15ba051888c168  servers/notifier_uart2rx.c
843683fbdf9836dbe8454cfee822b646  servers/notifier_uart2tx.c
543b0ddeef7d66f66b7126efa9b58ea0  servers/rand.c
53eb72a04263b25510535132b3453e35  servers/rps_server.c
290809cef3d686a8b38eb476c452e660  servers/servers.h
4ae8e8ee2506107ab967fbc46196bacb  servers/t2c.py
ce231d3a84732dd148194087514ad94d  servers/track.c
f9c227aabc82ef7f425c68592cf9c1e9  servers/track.dot
4215bb59f117e068440caf560bcb3938  servers/track.h
22205ebd7eb10fbfd0414b0d0aff2205  servers/track_data.c
f1fa12526ca078848c5df08c4448be1f  servers/track_data.png
efa80a78194d72780c9d8f869f82bacf  servers/trains.c
f26834428e4c987502cf8734e60cab10  servers/trains.h
775f0fd0347788f42800f45778fa1208  servers/uart1.c
9b245bca020cc929b760e3d1e44318d4  servers/uart2.c
1fc7829e40ad9731b92880bd06e5ae6a  user/clock_client.c
0a4ecd61c17f8fab2701df0fc602f52b  user/clock_client.h
54c3506b4ecac217dd3c02c3e8468aad  user/lib.c
441bb18450155365328c5c06e56ab244  user/lib.h
bb26439fa9e96b4ad416cca1ffbee9d8  user/rps_client.c
5cdf0685d1b04b08223b59fe346c2a2b  user/timings.c
a66c2ece6a834d354855b5a349577f16  user/trains_ui.c
06d5b5abfea72aec6d8b2201fd56a111  user/user.c
2737fde1654b841a2a8273a03121a808  user/user.h
8e241c895f15b3f2e15a654aab0cddd3  user/usyscall.h
f7af323cc581879bd8e471a1ac6fec1a  user/wm.c
cbe6fec4a4afd27ab0585e230cdb07eb  user/wrappers.c
```

2 Documentation

2.1 Tracking

Our trains are managed by individual "agent" processes. Trains initially start off in the "completely lost" state. The next unattributable sensor reading becomes the location of the train.

During normal operation the train expects a specific sensor to be triggered in the future. A train tracks it's distance from the last witnessed sensor and updates the global trains application server of it's whereabouts.

The trains exist in a variety of states. These states and the transitions between them are explained in the (very)psuedo-code below.

The "lostfunc" takes the information about a train before it was lost(last known position and time of lostness) and tries to pick up a sensor reading that seems reasonable. This is to avoid the so-called "teleportation" problem. (NOTE: right now that code is actually not active; it still uses the "hope for the best" algorithm of picking the next unknown sensor reading. We got tired, sorry.) Sensor values that are older than about 3 seconds get thrown out as "stale", and only those that are older than about 0.5 seconds are able to be picked up by (either) lostfunc algorithm.

```
Loop {
  switch (state) {
    case "completely lost"
       if there is an unattributable sensor reading, acquire it and
          use it for position. now state = normal
        break
     case "lost"
        call the lostfunc function with last know position and time when we got lost
        if we get back a sensor event, use it. state = normal
        break
     case "reserve blocked"
        if we can reserve the next segments of track
            speed up, state = normal
        break
     case "accelerating"
         update velocity
         // don't break
      default:
          if we hit the next sensor {
               update position (dx=0 etc.) and find next expected sensor
               drop all reservations for this train
               if we fail to reserve the track segments in front of us then slow
down, state = "reserve blocked"
         }
   }
  update dx
}
```

2.2 Reservations

Reservations are tracked using an array of TIDs. If the TID is 0, then the section of track is not reserved. Otherwise the task with aforementioned TID is controlling a train that has reserved the section of track. Reservable sections of the track are currently sensors. This makes it easy to map a sensor ID to an element of the reservation array. In the future, graph edges will be subdivided to add more vertices (finer-grained reservations) without needing to change our current code that traverses the track graph.

Trains are aware of the distances required for them to decelerate from their current speed to a stop. They use this information to reserve enough segments of track in front of them to allow them to stop before touching track segments that they cannot reserve. This is accomplished by a wrapper, ReserveChunks(sensor, distance), that repeatedly reserves track segments until it has enough to satisfy the distance requirement. If a train cannot reserve a segment it will stop and re-attempt to reserve the segment. We do not currently have any code for dealing with deadlocked trains. In the future, two trains will detect deadlock with a random-length timeout, after which they will reverse direction.

2.3 Path Finding

This was not implemented in time for the demo. There is very little that needs to be implemented beyond what exist(s/ed) at the time of the demo to finish this part up.

We have the track as an undirected graph data structure with weight edges and additional support for things like switches and direction changing. Each time the train hits a new sensor the shortest path to it's current goal is computed via djikstra's algorithm and the first node is returned. If there is a turn-out ahead that needs to be switch and the train owns it (via the reservation system) and it is within a threshold distance the turn-out is switched. There is a lot of recompilation here but this easily allows for us to handle cases such as track segments being permanently unavailable (due to stopped trains for example) in a simple manner.