# CS107 lecture 15



```
int main()
{
  int numAgents = 10;
  int numTickets = 150;
  for (int agent = 1; agent <= numAgents; agent++) {
    SellTickets(agent, numTickets / numAgents);
  }
  return 0;
}

void SellTickets(int agentID, int numTicketsToSell)
{
  while (numTicketsToSell > 0) {
    printf("Agent %d sells a ticket \n", agentID);
    numTicketsToSell--;
  }
  printf("Agent %d: All done. \n", agentID);
}
```

Have thread library with functions on top of C's thread library

```
int main()                                  for new main
{
  int numAgents = 10;
  int numTickets = 150;          ← no debugging info
  InitThreadPackage(false);
  for (int agent; agent <= numAgent; agent++) {
    char name[32];
    sprintf(name, "Agent %d Thread", agent); // name of thread
    ThreadNew(name, SellTickets, 2, agent, numTickets/numAgents)
                                  # of args of SellTickets
  }
  RunAllThreads(); // start running all created threads
  return 0;
}
```

// add randomness to pause threads

```
void SellTickets (int agentID, int numTicketsToSell)
{
  while (numTicketsToSell > 0) {
    printf ("Agent %d sells a ticket. \n", agentID);
    numTicketsToSell --;
    if (RandomChance (0.1)) { ThreadSleep (1000); }   <- remove from processor
  }
  printf ("Agent %d : All done. \n", agentID);
}
```

Agent 1 sells a ticket

Agent 2 sells a ticket

Agent 8 : All done!

Agent 4 : All done!

Assembly instructions are atomic for purposes of thread execution and switching.

C instructions are not atomic.

```
// make all threads to sell from the same counter variable
void SellTickets (int agent, int *numTicketsp)
{
   while (*numTicketsp > 0) {
      (*numTickets)--;
   }
}

int main()
{
   int numAgents = 10;
   int numTickets = 150;
   InitThreadPackage (false);
   for (int agent; agent <= numAgent; agent++) {
      char name[32];
      sprintf (name, "Agent %d Thread", agent);
      ThreadNew (name, SellTickets, 2, agent, &numTickets);
   }
   RunAllThreads ();
   return 0;
}
```

critical region

Scenario:

- threads are removed from processor after passing the test at *numTicketsp = 1 and before decrementing

```
// address the critical region problem
int main ()
{
   int numAgents = 10;
   int numTickets = 150;
   Semaphore lock = SemaphoreNew (-, 1);
   InitThreadPackage (false);
   for (int agent; agent <= numAgents;
                   agent++) {
      char name[32];
      sprintf (name, "Agent %d Thread", agent);
      ThreadNew (name, SellTickets, 3, agent, &numTickets, lock);
   }
   RunAllThreads ();
   return 0;
}
```

with a global integer wrapped around Semaphore type which is a pointer type

non-negative

resource available to 1 thread

can be other int value

Semaphore functions emulating -- and ++ wrt. lock.

-- SemaphoreWait (lock);

++ SemaphoreSignal (lock);

as atomic operations.

Semaphore pointer

```
void SellTickets (int agent, int * numTicketsp, Semaphore lock)
{
    while (true) {                          <- brings lock 1 to 0
        SemaphoreWait (lock);
        if (* numTicketsp == 0) break;
        (*numTicketsp)--;
        printf("-----");                    <- 0 to 1
        SemaphoreSignal (lock);
    }
    SemaphoreSignal (lock);
}
```

atomic

need after break to allow other threads to exit when *numTicketsp == 0

<- if removed from processor here lock stays and other threads cannot access the region until the thread that was interrupted unlocks it; others threads can make other progress

when SemaphoreWait is called when lock == 0, if does not decrement it, it blocks the thread until lock is incremented by previously interrupted thread.