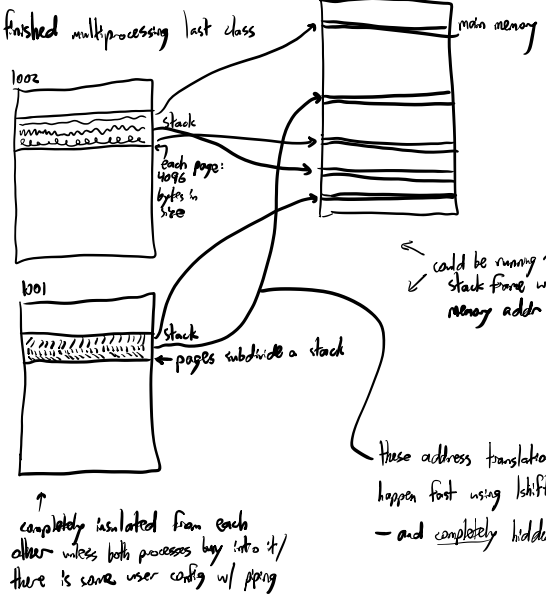

Today: Threads

thread: independent execution flow w/in process

each execution can agree where global variables are / update complex data structures together → run in same virtual address space → data integrity issues! → race conditions

processes's stack segment divided into miniature substacks

```
int main(-->) {
  cout << "Introverts example." << endl;
  thread introverts[6]; //unhandled thread at this pt:
  for(thread& t: introverts)
    t = thread(recharge); //constructor takes name of fxn to run in thread
  for(thread t: introverts)
    t.join(); //blocking method like wait/pid - halts until this thread is ready to "join back" w/ main execution thread
  return 0;
}
```

//contrived parallelism via threading directive

```
static void recharge() {
  cout << "I'm recharging." << endl;
}
```

↳ not multi-thread safe: doesn't acquire a lock on printing to terminal

at least running independent copies of this function

```
thread greeters[6];
for (size_t i=0; i<6; i++) {
  greeters[i] = thread(greet, i+1); //2nd arg to thread constructor!
}
```

```
static void greet(size_t id) {
  for(size_t i=0; i<id, i++){
    cout << oslock << "Greeter # " << id << " says hello!" << endl << osunlock;
  }
```

↑ bc cout and '<<' operator not thread safe

airline tickets example

`size_t numTickets = 250;` is the database - where workers consult whether there are tickets left to sell

`agents[i] = thread(agent, 101+i, ref(numTickets));`

↑ have to be explicit that it's passed by reference in thread constructor

In code from slides:

```
while (numTickets > 0) {
  handleCall();
  numTickets--;
}
```

separation between these 2 lines (checking & decrementing) is not thread safe - can sell "ghost tickets"

a "critical region" in multi-threading parlance

C++ statements aren't inherently atomic - result in multiple assembly code instructions

assembly code instructions are atomic