Implementing Go Channels

The Go philosophy is that goroutines should avoid reading & writing the same memory locations.

Instead goroutines should share data by exchanging messages.

Channels are objects that control the sharing of messages — can send to a channel, can receive from a channel.

Channels are typed, meaning a particular channel will always facilitate the exchange of the same kind of message. For us this means the length of the message sent or received by a particular channel will be the same.

Channels can contain a buffer with a fixed capacity — the maximum number of messages the channel can contain.

A channel with a non-zero capacity allows senders to possibly not block.
zero-capacity channel

sender must block until a
receiver is ready

receiver must block until a
sender is ready

if multiple senders are blocked,
or multiple receivers are blocked,
then they wait in a FIFO queue.

consider the case of the sender
arriving first:
sender will block — recording
its data address in its GCB block
by moving its GCB from
the ready list and inserting
it at the end of the queue
for the channel
and it will yield to the
next goroutine in the
ready list
if none, deadlock — panic
when receiver arrives, it
checks queue to see if there
is a waiting sender that copies the data from
the sender using the data address
in the sender's GCB (the data field needs to be a
field in the struct for the channel)
then it moves the sender's GCB to the end of the Ready list

the case of the receiver arrival
first is similar - when
the sender arrives, it will
copy its data to the receiver
using the data address field
of the receiver's GCB. And
then it will unblock the
receiver.

closing channels

a channel can be closed

need a waiting receivers should be unblocked
field in GCB for this purpose

return value of 0.
need a waiting senders should be unblocked and told to panic if/when they execute

nil channel
represented as a NULL channel handle

send to and receive from a nil channel could the goroutine to block forever

validating channel not zero!

put a "magic number" at the front of your struct that implements a channel

use this to do a rudimentary check that a handle is valid
if the check fails, panic
non-zero-capacity channels

does not contain

when allocating channel struct
must also allocate a buffer
must put fields in channel struct
to control the buffer:
  ptr to buffer
  number of data items currently
  in the buffer (the channel length)
when to insert the next data
item when to remove the next data item

send
check if waiting receive
  if so, buffer must be empty, so
  exchange is like for 0-capacity channel
check if room in buffer
  if so, insert into buffer and
do not block
otherwise must block again
  when garbage executes it should
  check if channel was closed
  when it was blocked
  if so, panic
Receive
if there is data client in the buffer, take it from buffer.
if there is a sender blocked, put its data client into the buffer & unblock the sender.
if there is a waiting sender, then this is a zero-capacity channel.
otherwise must block.
when some time elapses, it should check if channel was closed when it was blocked.
if so, return 0 instead of 1.

Other primitives

capChannel - return the channel capacity.

capBuffer - return the number of items in the channel buffer.

FreeChannel - free the memory for the channel.
close Goroutines — frees the memory used by all goroutines
note: does not free memory used by channels