A Garbage Collected Network Stack with CSP Threads

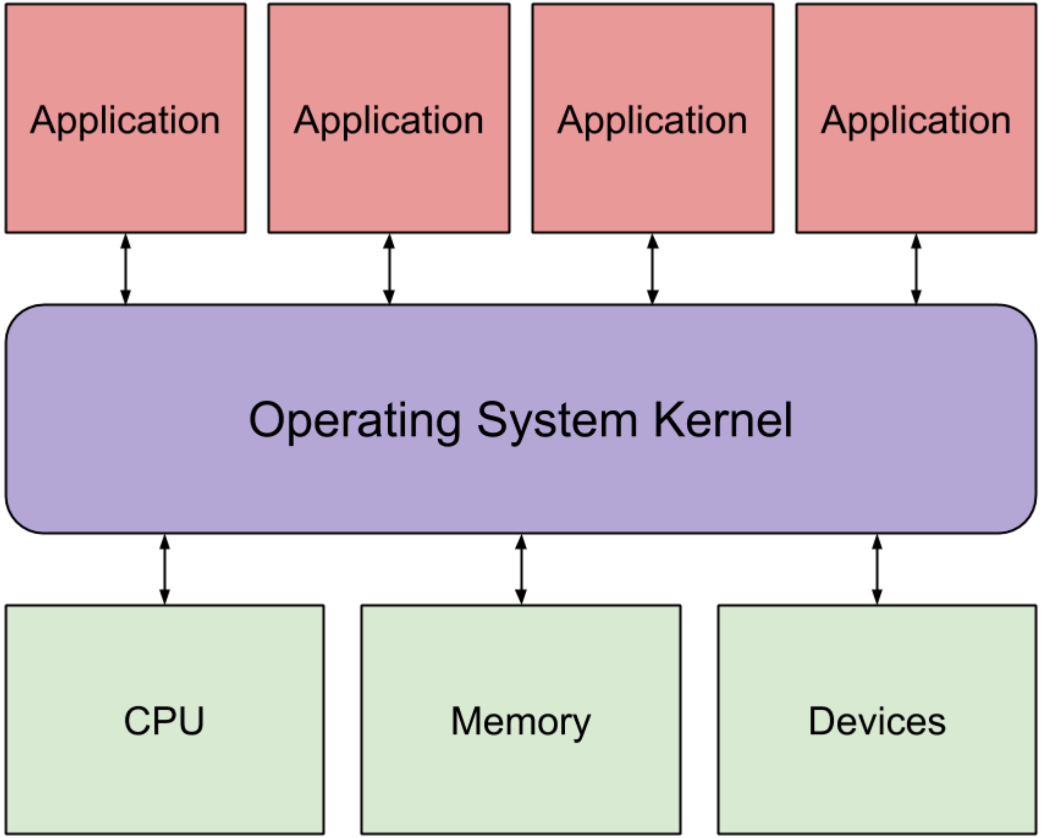
Harshal Sheth and Aashish Welling

Outline

- Problem
- Idea
  - Concerns
- Our Network Stack
  - Go
  - Code Comparisons
- Conclusion
  - Future Work

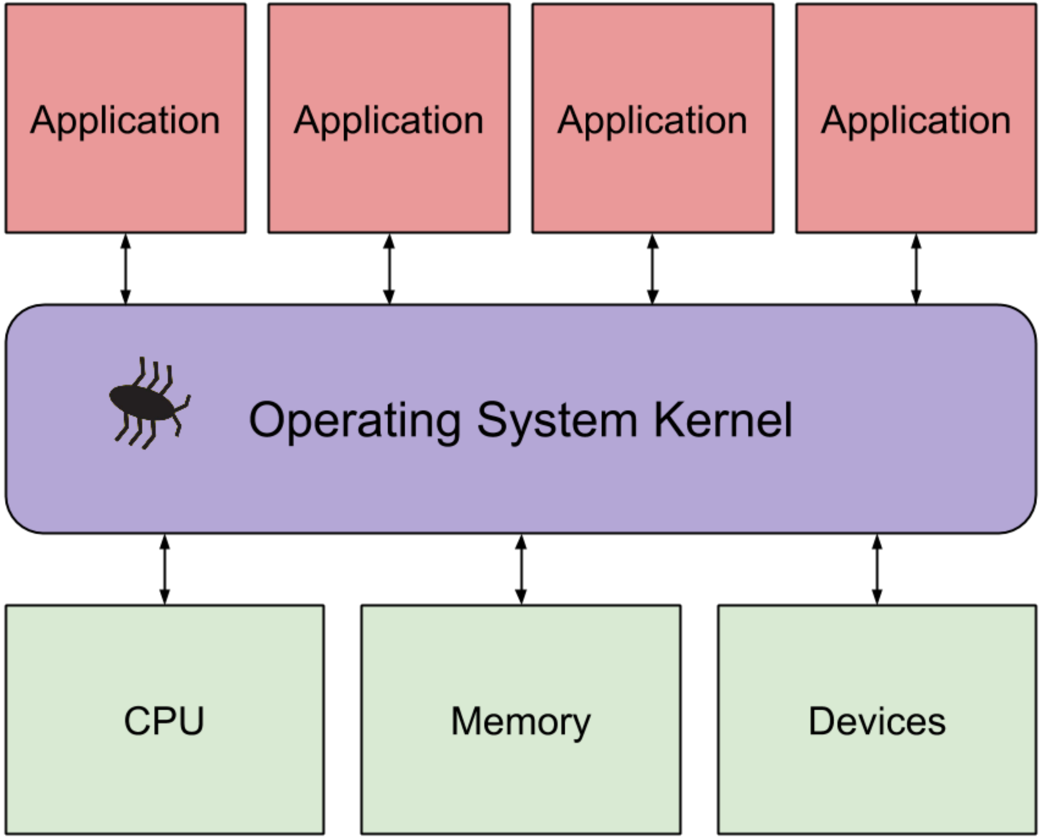
Background

The operating system runs all the software we use



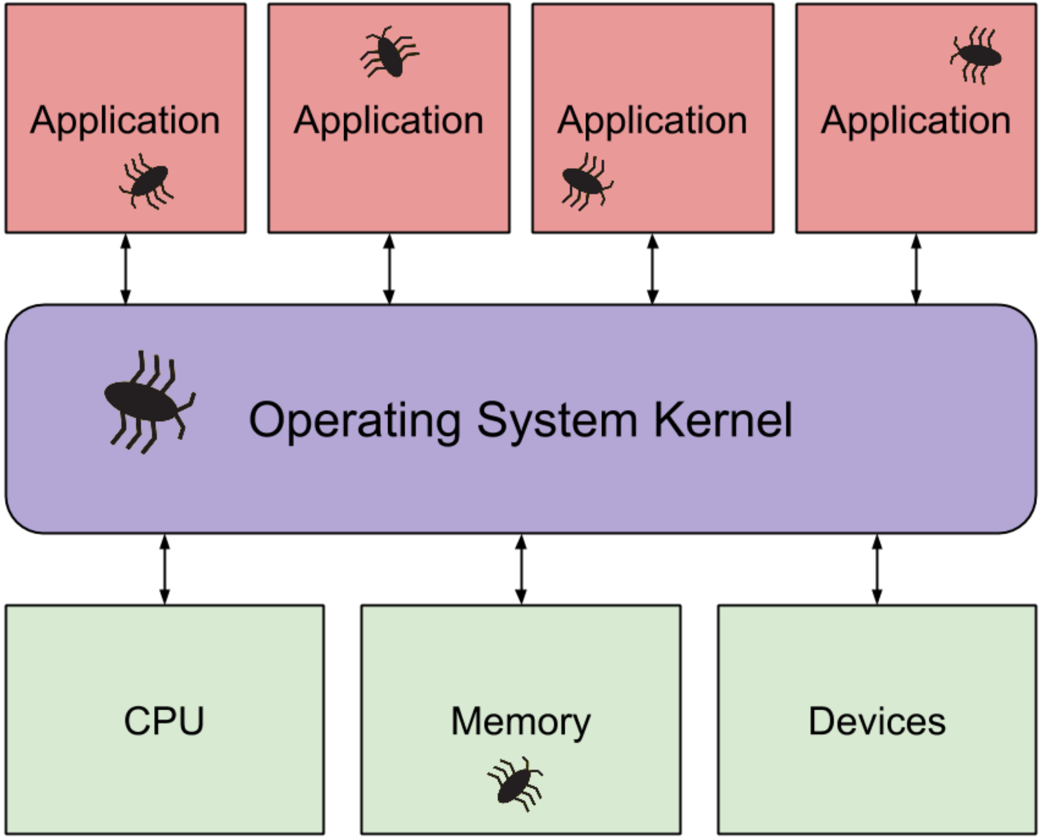
Problem

A bug in the kernel



Problem

A bug in the kernel may cause system crashes, memory corruptions, or abnormal behavior.



Problem

Commonly used kernels are written in C

C makes it challenging to write programs without bugs

- Memory leaks
- Memory corruption
- Deadlocks

Problem

The number of CPUs per computer is increasing.

It is challenging to write multithreaded programs correctly in C.

Our Approach

- Write the kernel in a high level language
  - Makes the code simpler, safer, and easier to maintain
- Use CSP style
  - Communicating Sequential Processes
  - Using many separate threads to accomplish tasks concurrently in the kernel
  - Passing data between concurrent tasks with channels

Potential Concerns

- Garbage collection can be costly
- Less control over memory usage

Our Project

- Evaluate the costs and benefits of a kernel in a high level language using CSP style by writing a network stack in the Go language.

- The network stack enables communication between computers (over networks).

Our Network Stack

- Simpler code
  - Less room for bugs and errors
- Parallelized
  - Bundles different tasks into their own threads
    - Automatically improves performance
  - Takes advantage of all available cores
- Modularized
  - Each thread maintains only its own state

Go: Reducing Bugs

- Strongly typed
  - No “void *” or other potentially dangerous types or casts
- Garbage collected
  - Provides memory safety
- Prevents memory corruption
  - Bounds checking
- First-class functions
  - Adds simplicity

Go: Promoting Concurrency

- Threads at the language level
  - Go threads are more lightweight than threads in C
- Communication between threads
  - Simplifies multithreaded code

- Go lends itself to CSP style

Code Comparisons

We display the C code of the lwIP network stack and the Go code of our network stack to compare simplicity.

C

```c
void *perform_work(void *argument) {
   int passed_in_value = *((int *) argument);
   printf("Worker thread with arg %d!\n", passed_in_value);
   sleep(2);
   return NULL;
}

int main(void) {
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int result_code, index;
   for (index = 0; index < NUM_THREADS; ++index) {
      thread_args[index] = index;
      result_code = pthread_create(&threads[index], NULL,
                 perform_work, (void *) &thread_args[index]);
      assert(0 == result_code);
   }
   for (index = 0; index < NUM_THREADS; ++index) {
      result_code = pthread_join(threads[index], NULL);
      assert(0 == result_code);
   }
   exit(EXIT_SUCCESS);
}
```

Takes .049 seconds of CPU time

Go

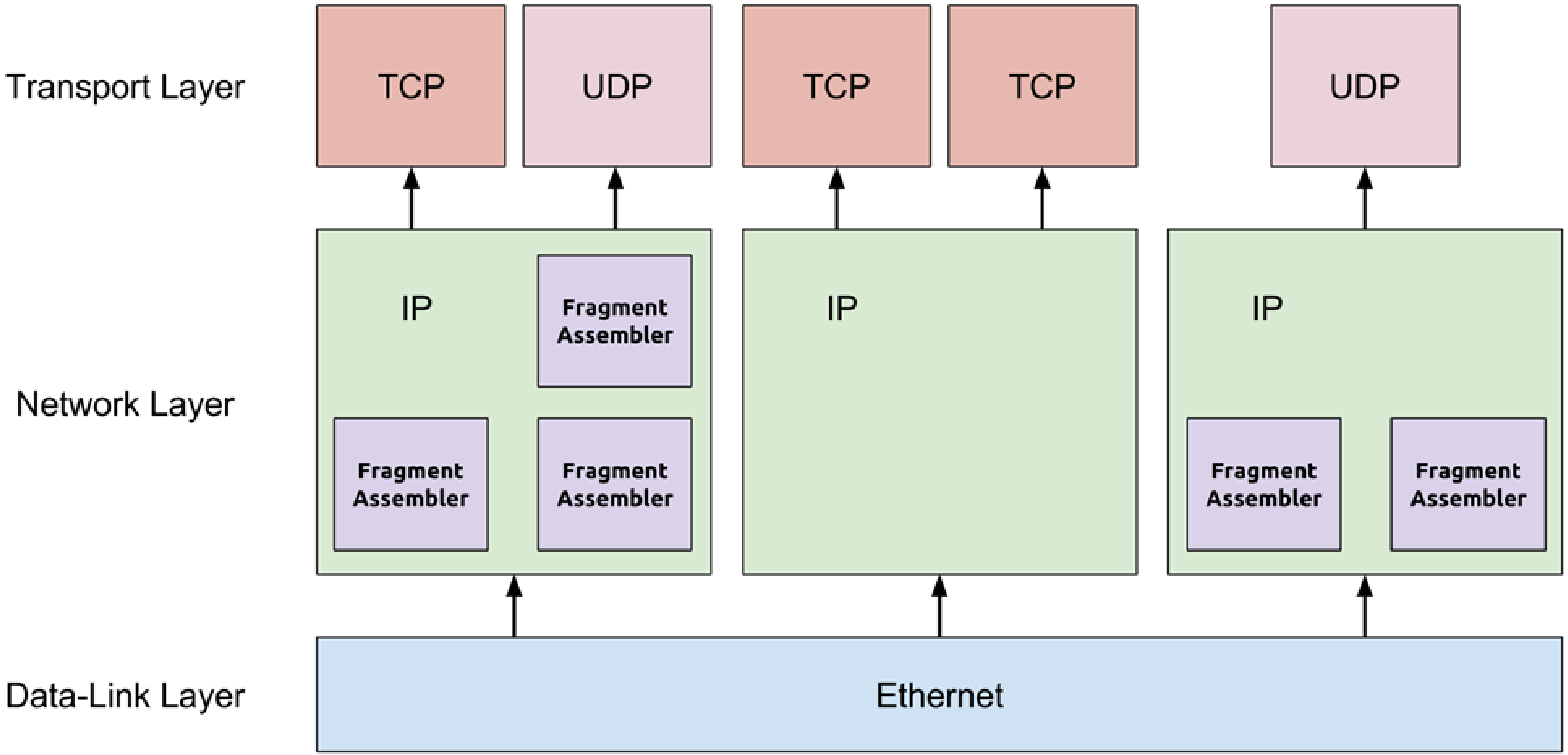
```go
func worker(arg int, done chan bool) {
    fmt.Println("Worker thread with argument", arg)
    time.Sleep(2 * time.Second)
    done <- true
}

func main() {
    finished := make(chan bool, NUM_THREADS)
    for index := 0; index < NUM_THREADS; index++ {
        go worker(index, finished)
    }
    for index := 0; index < NUM_THREADS; index++ {
        <- finished
    }
    fmt.Println("Finished")
}
```

Takes .007 seconds of CPU time

Parallelization

Each box represents a thread



C

```c
struct ip_reassdata* ipr;
ipr = (struct ip_reassdata *)memp_malloc(MEMP_REASSDATA);
if (ipr == NULL) {
 if (ip_reass_remove_oldest_datagram(fraghdr, clen) >= clen)
   ipr = (struct ip_reassdata *) memp_malloc(MEMP_REASSDATA);
 if (ipr == NULL)
   return NULL;
}
memset(ipr, 0, sizeof(struct ip_reassdata));
ipr->timer = IP_REASS_MAXAGE;
ipr->next = reassdatagrams;
reassdatagrams = ipr;
SMEMCPY(&(ipr->iphdr), fraghdr, IP_HLEN);
return ipr;

void ip_reass_tmr(void) {
  struct ip_reassdata *r, *prev = NULL;
  r = reassdatagrams;
  while (r != NULL) {
    if (r->timer > 0) {
      r->timer--;
      prev = r;
      r = r->next;
    } else {
      /* Deal with a timeout */
     }
   }
}
```

Go

```go
ipr.fragBuf[bufID] = make(chan []byte)
quit := make(chan bool)
done := make(chan bool)
didQuit := make(chan bool)

go ipr.fragmentAssembler(ipr.fragBuf[bufID], quit,
didQuit, ipr.incomingPackets, done)
go ipr.killFragmentAssembler(quit, didQuit, done,
bufID)

func (ipr* IP_Reader) killFragmentAssembler(
             quit chan<- bool, didQuit <-chan bool,
                 done <-chan bool, bufID string) {
  select {
    case <-time.After(FRAGMENT_TIMEOUT):
      quit <- true
      <-didQuit
    case <-done:
  }
  /* Deal with a timeout */
}
```

C

```c
for (q = ipr->p; q != NULL;) {
  iprh_tmp = (struct ip_reass_helper*)q->payload;
  if (iprh->start < iprh_tmp->start) {
    iprh->next_pbuf = q;
    if (iprh_prev != NULL) {
      iprh_prev->next_pbuf = new_p;
    } else {
      ipr->p = new_p;
    }
    break;
  } else if(iprh->start == iprh_tmp->start) {
    goto freepbuf;
  } else if (iprh_prev != NULL)
    if (iprh_prev->end != iprh_tmp->start)
      valid = 0;
  q = iprh_tmp->next_pbuf;
  iprh_prev = iprh_tmp;
}

if (q == NULL) {
  if (iprh_prev != NULL) {
    iprh_prev->next_pbuf = new_p;
    if (iprh_prev->end != iprh->start) {
      valid = 0;
    }
  } else {
    ipr->p = new_p;
  }
}
```

Go

```go
offset := 8 * (uint64(hdr[6]&0x1F)<<8 +
                uint64(hdr[7]))
extraFrags[offset] = p

for {
  if storedFrag, found :=
        extraFrags[uint64(len(payload))]; found {
    delete(extraFrags, uint64(len(payload)))
    payload = append(payload, storedFrag...)
  } else {
    break
  }
}
```

C

```c
ipr->datagram_len += IP_HLEN;

r = ((struct ip_reass_helper*)ipr->p->payload)->next_pbuf;

fraghdr = (struct ip_hdr*)(ipr->p->payload);
SMEMCPY(fraghdr, &ipr->iphdr, IP_HLEN);
IPH_LEN_SET(fraghdr, htons(ipr->datagram_len));
IPH_OFFSET_SET(fraghdr, 0);
IPH_CHKSUM_SET(fraghdr, inet_chksum(fraghdr, IP_HLEN));

p = ipr->p;

while(r != NULL) {
  iprh = (struct ip_reass_helper*)r->payload;

  pbuf_header(r, -IP_HLEN);
  pbuf_cat(p, r);
  r = iprh->next_pbuf;
}

if (ipr == reassdatagrams) {
  prev = NULL;
} else {
  for (prev = reassdatagrams; prev != NULL; prev = prev->next)
    if (prev->next == ipr)
      break;
}
```

Go

```go
fullPacketHdr := hdr
totalLen := uint16(fullPacketHdr[0]&0x0F)*4 +
             uint16(len(payload))
fullPacketHdr[2] = byte(totalLen >> 8)
fullPacketHdr[3] = byte(totalLen)
fullPacketHdr[6] = 0
fullPacketHdr[7] = 0
check := calculateChecksum(fullPacketHdr[:20])
fullPacketHdr[10] = byte(check >> 8)
fullPacketHdr[11] = byte(check)

go func() {
    finished <- append(fullPacketHdr, payload...)
}()
done <- true
```

Performance

- Drop rate of approximately 0.002%
- Only two times slower than Linux kernel
  - Unfair comparison

Conclusions

- Higher level languages allow for simpler and cleaner code
- Programming in CSP style enables simple network stack design

Future Work

- Implement additional protocols
  - ARP, ICMP, etc.
- Optimize the network stack implementation
- Support IPv6
- Implement other sections of the kernel

Acknowledgements

We would like to thank:

- Our mentor **Cody Cutler** for guiding us throughout the project
- **Prof. Frans Kaashoek** for suggesting this project
- **MIT PRIMES** for this opportunity
- Our parents