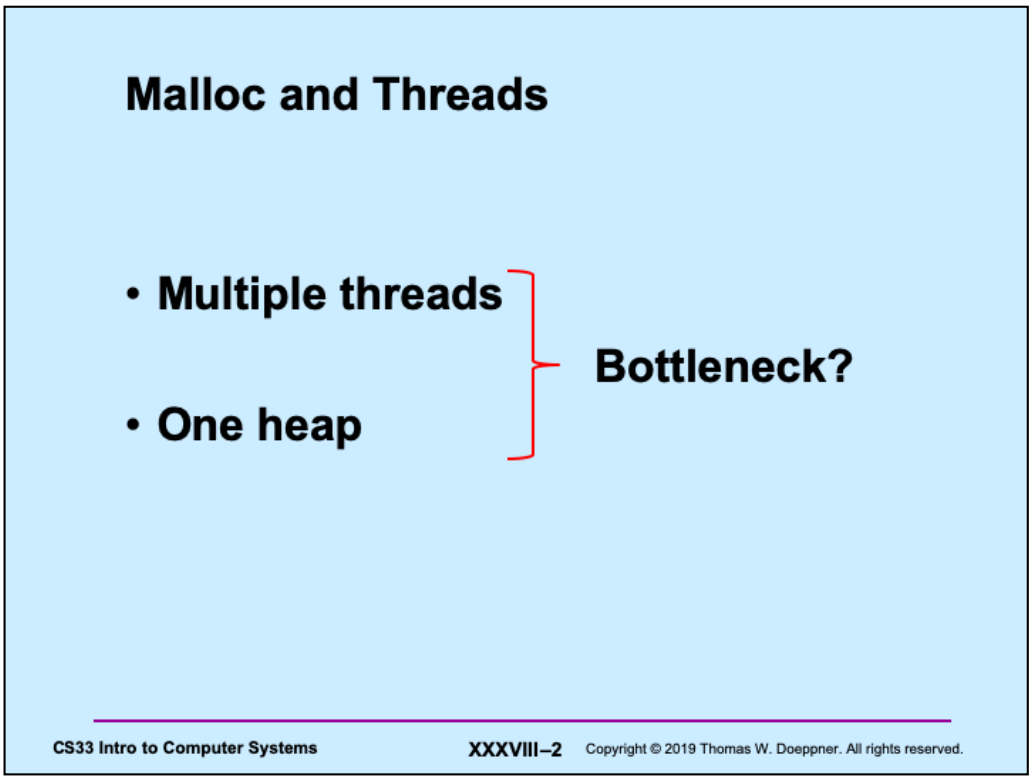
CS 33

Final Words



In a naïve multithreaded implementation of malloc/free, there is one mutex protecting the heap, resulting in a bottleneck.

Solution 1

- Divvy up the heap among the threads
  - each thread has its own heap
  - no mutexes required
  - no bottleneck
- How much heap does each thread get?
- What if one thread mallocs something and another thread frees it?

Solution 2

- Multiple “arenas”
  - each with its own mutex
  - thread allocates from the first one it can find whose mutex was unlocked
    - » if none, then creates new one
  - deallocations go back to original arena

Solution 3

- Global heap plus per-thread heaps
  - threads pull storage from global heap
  - freed storage goes to per-thread heap
    - » unless things are imbalanced
      - then thread moves storage back to global heap
  - mutex on only the global heap
- What if one thread allocates and another frees storage?

Malloc/Free Implementations

- ptmalloc
  - based on solution 2
  - in glibc (i.e., used by default)
- tcmalloc
  - based on solution 3
  - from Google
- Which is best?

Test Program

```
const unsigned int N=64, nthreads=32, iters=10000000;
int main() {
  void *tfunc(void *);
  pthread_t thread[nthreads];
  for (int i=0; i<nthreads; i++) {
    pthread_create(&thread[i], 0, tfunc, (void *)i);
    pthread_detach(thread[i]);
  }
  pthread_exit(0);
}
void *tfunc(void *arg) {
  long i;
  for (i=0; i<iters; i++) {
    long *p = (long *)malloc(sizeof(long)*((i%N)+1));
    free(p);
  }
  return 0;
}
```

Compiling It …

```
% gcc -o ptalloc alloc.cc -lpthread
% gcc -o tcalloc alloc.cc -lpthread -ltcmalloc
```


Running It (2014) …

```
$ time ./ptalloc
real    0m5.142s
user    0m20.501s
sys     0m0.024s
$ time ./tcalloc
real    0m1.889s
user    0m7.492s
sys     0m0.008s
```


The code was run on an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz.

What's Going On?

```
$ strace -c -f ./ptalloc
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.040002          13      3007       520 futex
 …


$ strace -c -f ./tcalloc
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 …
  0.00    0.000000           0        59        13 futex
 …
```


strace is a system facility that supplies information about the system calls a process uses. The –c flag tell is to print the cumulative statistics after the process terminates. The –f flag tells it to include information on all threads and child processes.

Test Program 2, part 1

```
#define N 64
#define npairs 16
#define allocsPerIter 1024
const long iters = 8*1024*1024/allocsPerIter;
#define BufSize 10240
typedef struct buffer {
  int *buf[BufSize];
  unsigned int nextin;
  unsigned int nextout;
  sem_t empty;
  sem_t occupied;
  pthread_t pthread;
  pthread_t cthread;
} buffer_t;
```


This program creates pairs of threads: one thread allocates storage, the other deallocates storage. They communicate using producer-consumer communication.

Test Program 2, part 2

```
int main() {
  long i;
  buffer_t b[npairs];
  for (i=0; i<npairs; i++) {
    b[i].nextin = 0;
    b[i].nextout = 0;
    sem_init(&b[i].empty, 0, BufSize/allocsPerIter);
    sem_init(&b[i].occupied, 0, 0);
    pthread_create(&b[i].pthread, 0, prod, &b[i]);
    pthread_create(&b[i].cthread, 0, cons, &b[i]);
  }
  for (i=0; i<npairs; i++) {
    pthread_join(b[i].pthread, 0);
    pthread_join(b[i].cthread, 0);
  }
  return 0;
}
```


The main function creates *npairs* (16) of communicating pairs of threads.

Test Program 2, part 3

```
void *prod(void *arg) {
  long i, j;
  buffer_t *b = (buffer_t *)arg;
  for (i = 0; i<iters; i++) {
    sem_wait(&b->empty);
    for (j = 0; j<allocsPerIter; j++) {
      b->buf[b->nextin] = malloc(sizeof(int)*((j%N)+1));
      if (++b->nextin >= BufSize)
        b->nextin = 0;
    }
    sem_post(&b->occupied);
  }
  return 0;
}
```


To reduce the number of calls to *sem_wait* and *sem_post*, at each iteration the thread calls malloc *allocsPerIter* (1024) times.

Test Program 2, part 4

```
void *cons(void *arg) {
  long i, j;
  buffer_t *b = (buffer_t *)arg;
  for (i = 0; i<iters; i++) {
    sem_wait(&b->occupied);
    for (j = 0; j<allocsPerIter; j++) {
      free(b->buf[b->nextout]);
      if (++b->nextout >= BufSize)
        b->nextout = 0;
    }
    sem_post(&b->empty);
  }
  return 0;
}
```


Running It (2014) …

```
$ time ./ptalloc2
real    0m1.087s
user    0m3.744s
sys     0m0.204s
$ time ./tcalloc2
real    0m3.535s
user    0m11.361s
sys     0m2.112s
```


The code was run on a SunLab machine (an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz).

What's Going On?

```
$ strace -c -f ./ptalloc2
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 94.96    2.347314          44     53653     14030 futex
…
$ strace -c -f ./tcalloc2
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 93.86    6.604632          36    185731     45222 futex
 …
```


Running it (2015) ...

```
sphere $ time ./ptalloc

real    0m2.373s
user    0m9.152s
sys     0m0.008s
sphere $ time ./tcalloc

real    0m4.868s
user    0m19.444s
sys     0m0.020s
```


Running it (2015) ...

```
kui $ time ./ptalloc

real    0m2.787s
user    0m11.045s
sys     0m0.004s
kui $ time ./tcalloc

real    0m1.701s
user    0m6.584s
sys     0m0.004s
```


Running it (2015) ...

```
cslab0a $ time ./ptalloc

real    0m2.234s
user    0m8.468s
sys     0m0.000s
cslab0a $ time ./tcalloc

real    0m4.938s
user    0m19.584s
sys     0m0.000s
```


What's Going On?

- On kui:
  - `libtcmalloc.so -> libtcmalloc.so.4.1.0`
- On other machines:
  - `libtcmalloc.so -> libtcmalloc.so.4.2.2`

However (2015) ...

```
cslab0a $ time ./ptalloc2

real    0m0.466s
user    0m1.504s
sys     0m0.212s
cslab0a $ time ./tcalloc2

real    0m1.516s
user    0m5.212s
sys     0m0.328s
```


It's 2019

- tcmalloc no longer exists
  - no explanation from Google, it's simply gone
- ptmalloc continues to improve

Thread Scheduling

- The OS multiplexes threads on the available processors/cores
  - share the processors equally
    - » time slicing: each thread gets a fixed amount of time before it's forced to yield the processor to another thread (if there is one)
  - some threads are more important than others
    - » priorities: higher-priority threads get the processor in preference to lower-priority threads

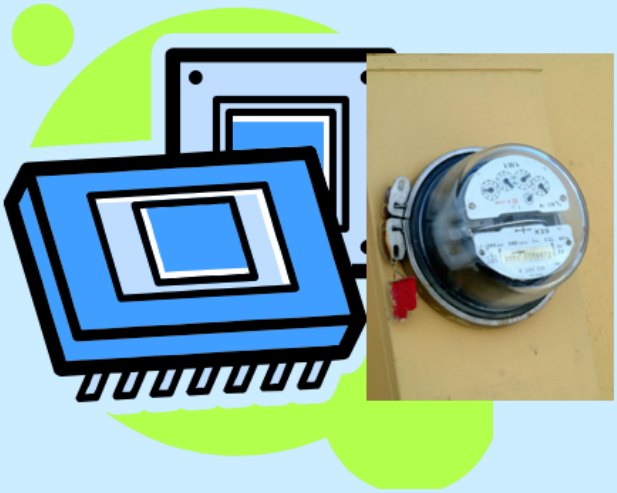
A Scheduling Issue

- You and four friends each contribute $1000 towards a server
  - you, rightfully, feel you own 20% of it
- Your friends are into threads, you're not
  - they run 5-threaded programs
  - you run a 1-threaded program
- The scheduler treats all threads equally
- Their programs each get 5/21 of the processor
- Your programs get 1/21 of the processor
  - (you should have paid more attention to the fractal threads lab)

Lottery Scheduling

- 25 lottery tickets are distributed equally to you and your four friends
  - you give 5 tickets to your one thread
  - they give one ticket each to their threads
- A lottery is held for every scheduling decision
  - your thread is 5 times more likely to win than the others

Metered Processors



To measure the usage of a processor, let's assume the existence of a meter.

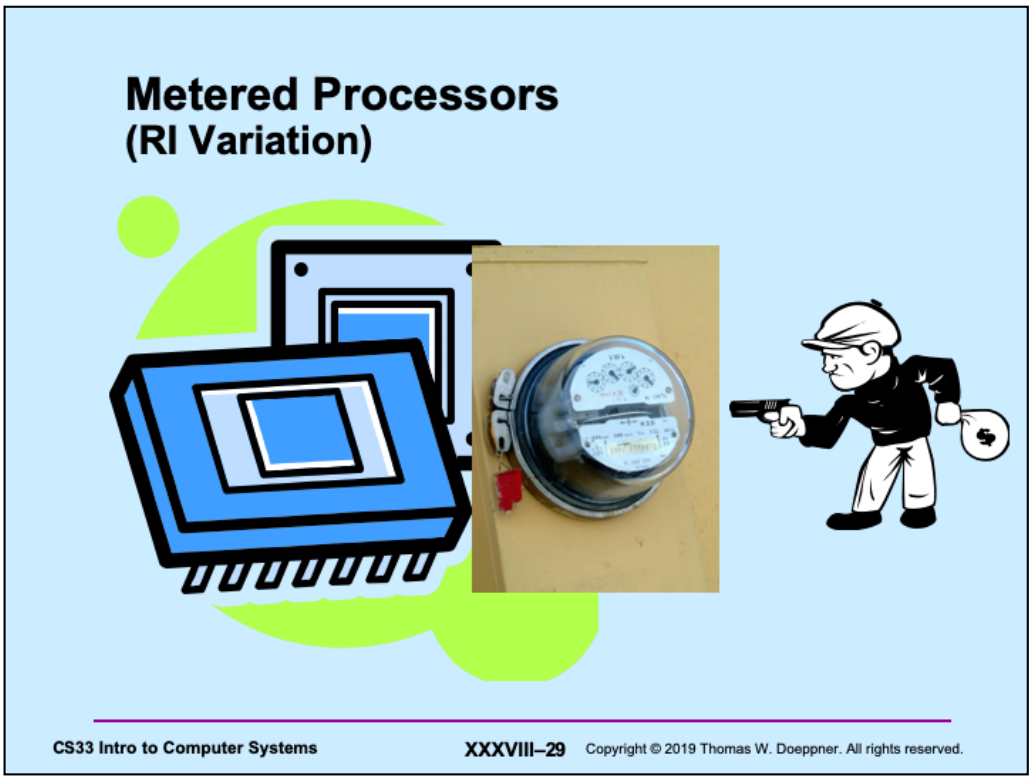
Algorithm

- Each thread has a meter, which runs only when the thread is running on the processor
- At every clock tick
  - give processor to thread that's had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins

Assuming all threads are equal, all started at the same time, and all run forever, the intent is to share the processor equitably. Note that has the time between clock ticks approaches zero, each thread gets 1/n of total processor time, where n is the number of threads.

Issue

- Some threads may be more important than others



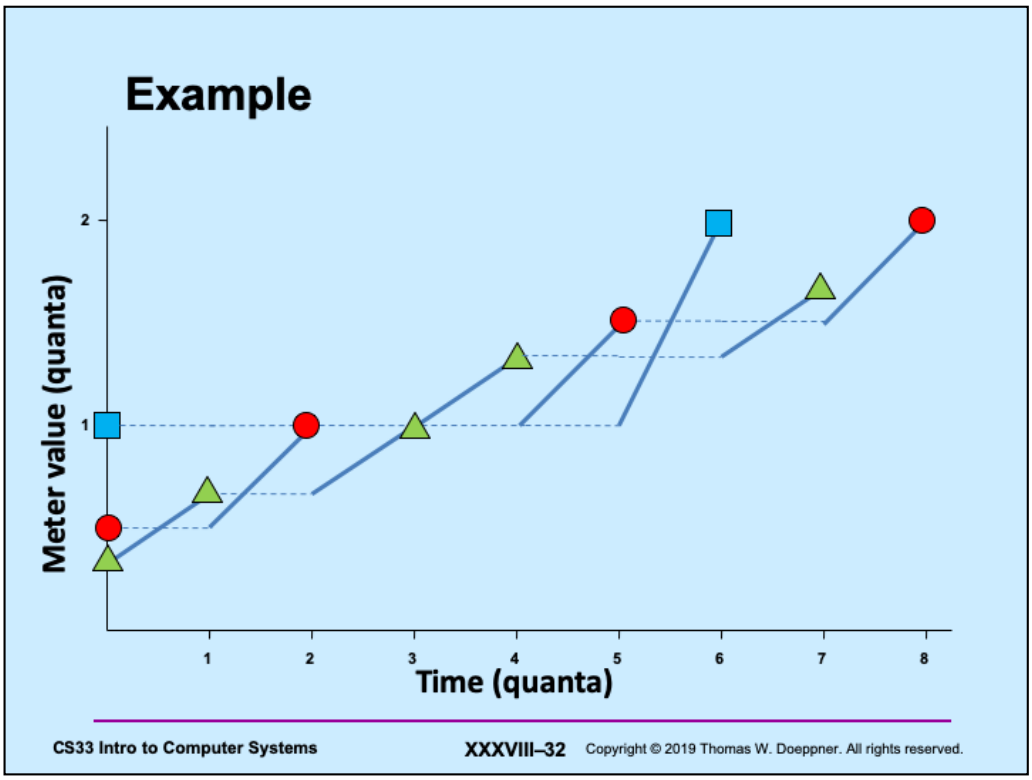
Let's now assume that meters can be "fixed" so that they run more slowly than they should. Thus a thread with a fixed meter gets charged for less processor time than it has actually used.

Details …

- Each thread pays a bribe
  - the greater the bribe, the slower the meter runs
  - to simplify bribing, you buy "tickets"
    - » one ticket is required to get a fair meter
    - » two tickets get a meter running at half speed
    - » three tickets get a meter running at 1/3 speed
    - » etc.

New Algorithm

- Each thread has a *(possibly crooked)* meter, which runs only when the thread is running on the processor
- At every clock tick
  - give processor to thread that's had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins



The slide illustrates the execution of three threads using stride scheduling. Thread 1 (labeled with a triangle) has a paid a bribe of three tickets. Thread 2 (labeled with a circle) has paid a bribe of two tickets, and thread three (labeled with a square) had paid only one ticket. The thicker lines indicate when a thread is running. Their slopes are proportional to the meter rates (and inversely proportional to the bribe). Note that meter values on the y axis are twice as far apart as ticks on the x axis.

In this example, a total bribe of six tickets has been paid. After six clock ticks, each thread's meter has been increased by 1.

In general, if the clock ticks once per second and the total bribe is B, then after B seconds, each thread's meter has increased by exactly 1. To see this, assume that each thread t_i starts with a meter reading of the reciprocal of its bribe b_i. To make this easier, let's assume that each thread has paid a different bribe. Suppose thread t_1 paid the largest bribe, b_1. After some period of time its meter will have increased by 1, requiring b_1 seconds of actual execution. Since it's the thread that paid the largest bribe, its meter will be increased by 1 before that of any other thread. It of course won't run again until its meter has the lowest value. Thread t_2, which paid the second largest bribe, will be the second thread to have its meter increased by 1, requiring b_2 seconds of actual execution. It also won't run again until its meter has the lowest value. Similar arguments can be made for the remaining threads, through t_n. Once t_n's meter has been increased by 1, t_1 again has the lowest meter value and the cycle starts again. The total amount of time required to get to this point is $b_1 + b_2 + \ldots + b_n$, i.e., the total bribe.

You'll Soon Finish CS 33 …

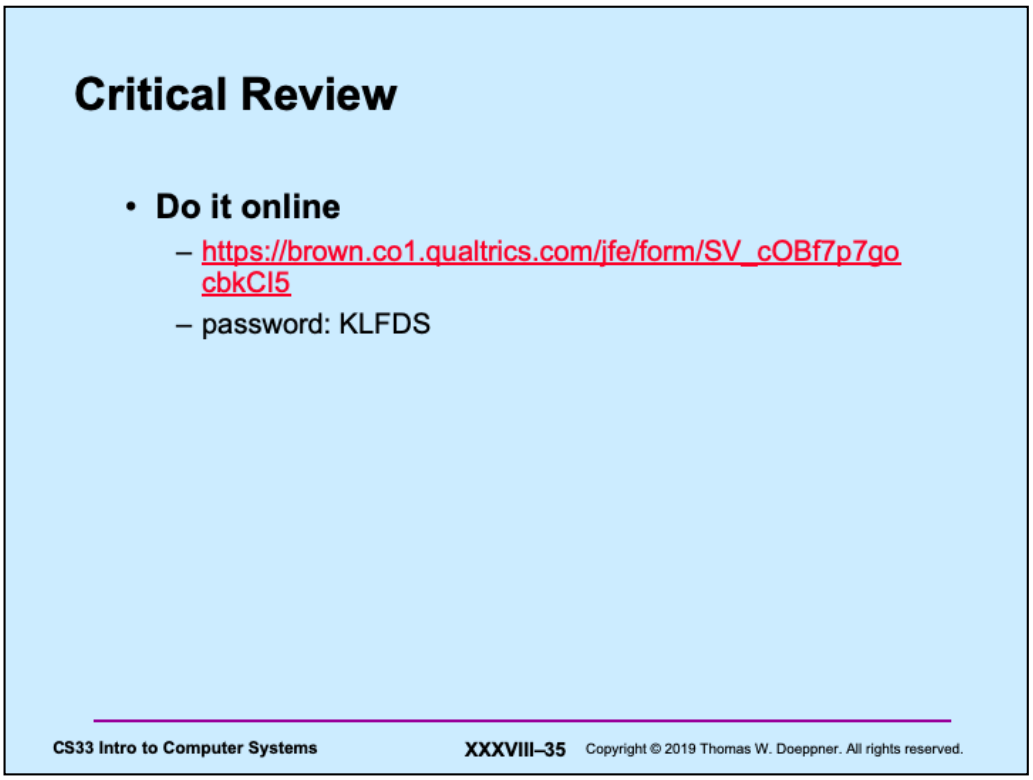
- You might
  - celebrate

  - take another systems course
    - » 32
    - » 131
    - » 138
    - » 166
    - » 167

  - become a 33 TA

Systems Courses Next Semester

- CS 32 (Intro to Software Engineering)
  - you've mastered low-level systems programming
  - now do things at a higher level
  - learn software-engineering techniques using Java, XML, etc.
- CS 131 (Fundamentals of Computer Systems)
  - an overview of how computer systems work
- CS 138 (Distributed Systems)
  - you now know how things work on one computer
  - what if you've got lots of computers?
  - some may have crashed, others may have been taken over by your worst (and smartest) enemy
- CS 166 (Computer Systems Security)
  - liked buffer?
  - you'll really like 166
- CS 167/169 (Operating Systems)
  - still mystified about what the OS does?
  - write your own!



The URL for the critical review is https://brown.co1.qualtrics.com/jfe/form/SV_cOBf7p7gocbkCI5.

The password is KLFDS.

The End

Well, not quite …

Database is due on 12/13.

The TAs and I will hold hours all this week.

I'll hold hours 3-4 today, 2-4 Wednesday, 2-5 Friday

Happy coding and happy holidays!

