Condition Variables

Glenn Bruns
CSUMB

Lecture Objectives

After this lecture, you should be able to:

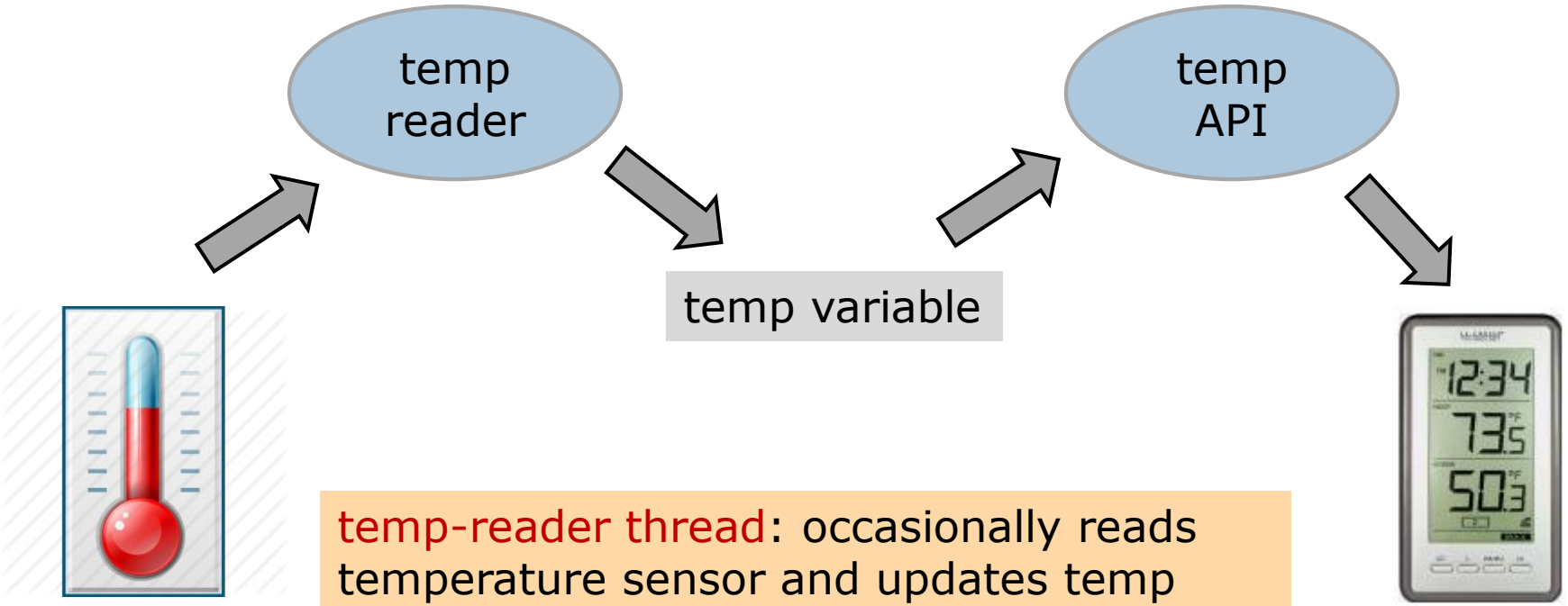
- Explain why condition variables are used
- Build a bounded buffer with condition variables
- Appreciate how concurrency bugs are subtle

Recap

- We're learning to write concurrent programs
- We're using C with the pthreads library
- We create concurrent programs using synchronization primitives like locks
- locks provide for mutually exclusive access to shared data
- We've seen how to build thread-safe data structures with locks

Do we need more than locks?

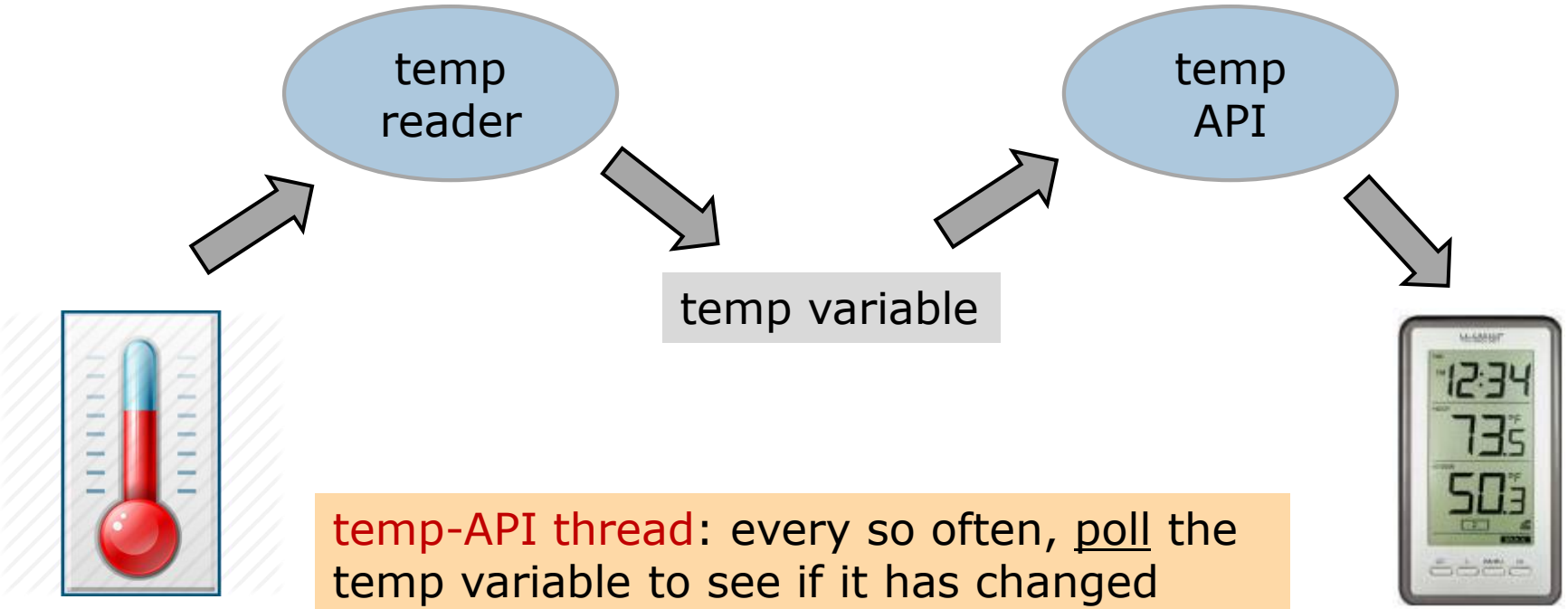
Example: temperature monitor



temp-reader thread: occasionally reads temperature sensor and updates temp variable

temp-API thread: updates temperature API when the temperature variable changes

Polling solution



temp-API thread: every so often, poll the temp variable to see if it has changed

Polling solution using pthreads

```
static int temp = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
void *read_sensor(void *threadid) {
  int sensed_temp;
  while (1) {
    // get reading from sensor
    sensed_temp = 70 + rand() % 5;

    // update shared temp variable
    pthread_mutex_lock(&lock);
    temp = sensed_temp;
    pthread_mutex_unlock(&lock);
    sleep(1);
  }
}
```

```
void *temp_api(void *threadid) {
  int old_temp = 0;
  int new_temp;
  // poll the temp variable
  while (1) {
    pthread_mutex_lock(&lock);
    new_temp = temp;
    pthread_mutex_unlock(&lock);
    if (new_temp != old_temp) {
      printf("current temp: %d\n",
              new_temp);
      old_temp = new_temp;
    }
  }
}
```

Problems with polling solution

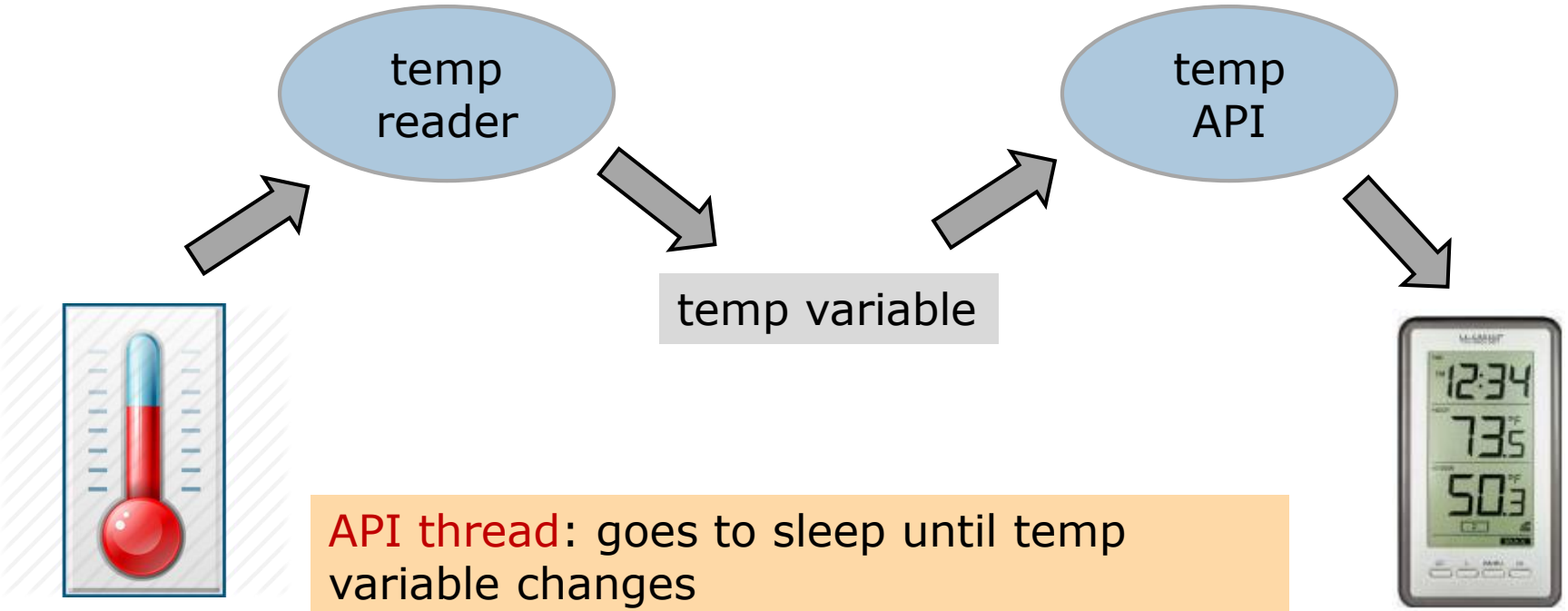
Q: What are the problems with this solution?

- inefficient: when the API thread runs, it usually accomplishes nothing
- inefficient: when the API thread is running, the sensor thread can't run

Polling is like checking for new email every 10 seconds.

It would be better if the sensor thread notified the API thread when the temperature changed.

Notification solution



API thread: goes to sleep until temp variable changes

reader thread: wakes up API thread when a new temperature value is written

Notification solution using pthreads

```
static int temp = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t    cv = PTHREAD_COND_INITIALIZER;
```

```
void *read_sensor(void *threadid) {
  int sensed_temp;
  while (1) {
    sensed_temp = 70 + rand() % 5;
    pthread_mutex_lock(&lock);
    temp = sensed_temp;
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&lock);
    sleep(1);
  }
}
```

```
void *temp_api(void *threadid) {
  int old_temp = 0;
  while (1) {
    pthread_mutex_lock(&lock);
    while (temp == old_temp) {
      pthread_cond_wait(&cv, &lock);
    }
    printf("current temp: %d\n",
           temp);
    old_temp = temp;
    pthread_mutex_unlock(&lock);
  }
}
```

Condition variable

A condition variable is a synchronization variable that lets a thread efficiently wait for a change to shared state

`pthread_cond_wait(cond, lock)`

- releases lock, suspends execution of calling thread, puts calling thread on cond. variable's waiting list. The lock is re-acquired before the thread returns from wait.

`pthread_cond_signal(cond)`

- takes a thread off the waiting list and marks it as "ready" (does nothing if waiting list empty)

`pthread_cond_broadcast(cond)`

- takes all threads off waiting list and marks them as ready

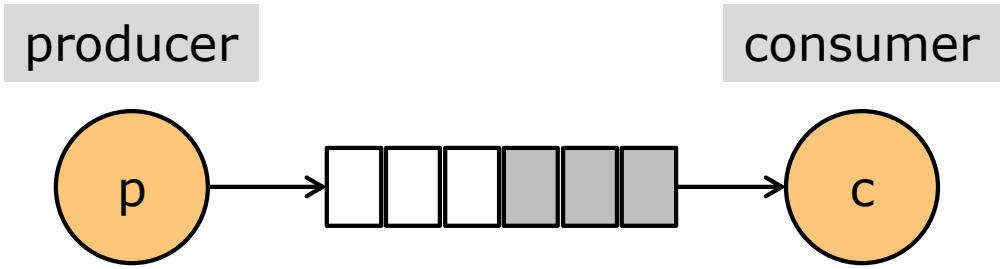
API details

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

A lock is used with wait() but not signal().

The bounded buffer problem

- A classic concurrency problem
- One thread waits for buffer to be non-full before writing
- Another thread waits for buffer to be non-empty before reading
- Linux pipes use a bounded buffer



Baby bounded buffer

```
int buffer;
int count = 0; // init. empty

void put(int value) {
  assert(count == 0);
  count = 1;
  buffer = value;
}

int get() {
  assert(count == 1);
  count = 0;
  return buffer;
}
```

buffer put and get functions

```
void *producer(void *arg) {
  int loops = (int)arg;
  int i;
  for (i = 0; i < loops; i++) {
    put(i);
  }
}

void *consumer(void *arg) {
  int i;
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

producer/consumer threads

(code from Operating Systems: Three Easy Pieces, Arpaci-Dusseau et al)

Condition variable design pattern

```
method_that_signals() {
  Pthread_mutex_lock(&mutex);

  // read/write shared state here

  // if state has changed in a way
  // that would allow another thread
  // to progress, signal

  Pthread_cond_signal(&cond);

  Pthread_mutex_unlock(&mutex);
}
```

```
method_that_waits() {
  Pthread_mutex_lock(&mutex);

  // read/write shared state here

  while (!test_on_shared_state()) {
    Pthread_cond_wait(&cond, &mutex);
  }
  assert(test_on_shared_state());

  // read/write shared state here

  Pthread_mutex_unlock(&mutex);
}
```

Remember:

- always lock before calling signal or wait
- the call to wait automatically releases the lock and puts the thread on a waiting list
- when a waiting thread is re-enabled through a signal, it may not run immediately

(code from Operating Systems: Principles and Practice, Anderson & Dahlin)

Summary

- Condition variables are an efficient way for a thread to wait for another thread to do something.
- Care has to be taken in using condition variables correctly.