CSSE2310 — 8.2

Threads and Synchronization continued

Passing values into threads

So far we've passed in strings (`char*`).

- `char*` → `void*` → `char*`

What about `int`?

See `thread4.c`

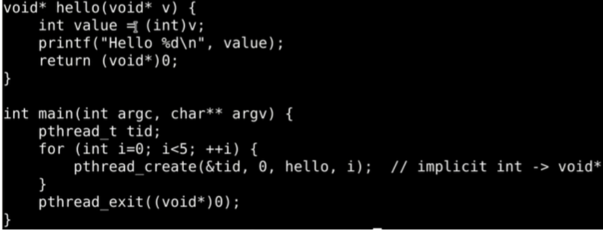
Why bother with `malloc()`ing?

See `thread5.c`

This is an example of a "race condition".

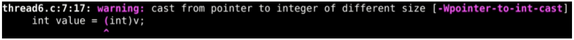
Abusing pointers

ints fit in void* right?



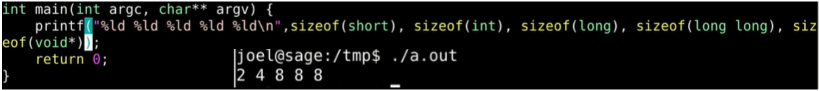
```
void* hello(void* v) {
    int value = (int)v;
    printf("Hello %d\n", value);
    return (void*)0;
}

int main(int argc, char** argv) {
    pthread_t tid;
    for (int i=0; i<5; ++i) {
        pthread_create(&tid, 0, hello, i);  // implicit int -> void*
    }
    pthread_exit((void*)0);
}
```

What about thread6.c?



```
thread6.c:7:17: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
     int value = (int)v;
                 ^
```

- We could pass long instead to get around the size issue.
- The real problem[1] here is that it works
    - . . . in this case.
    - it encourages people to ignore warnings



```
int main(int argc, char** argv) {
    printf("%ld %ld %ld %ld %ld\n",sizeof(short), sizeof(int), sizeof(long), sizeof(long long), siz
eof(void*));
    return 0;
}
```

```
joel@sage:/tmp$ ./a.out
2 4 8 8 8
```

[1] from a teaching point of view

Multiple parameters

Suppose we have a function which we would like to run in a thread:

```c
void do_things(char** items, char* s, int limits);
```

It doesn't match the required signature.

Multiple parameters

Declare a struct type to hold all the values:

```
struct Params {
    char** items;
    char* s;
    int limits;
};
```

Add a wrapper function

```
void* do_things_wrapper(void* v) {
    struct Params* p =(struct Params*)v;
    do_things(p->items, p->s, p->limits);
    free(v);    // can't free earlier without copying the struct
}
```

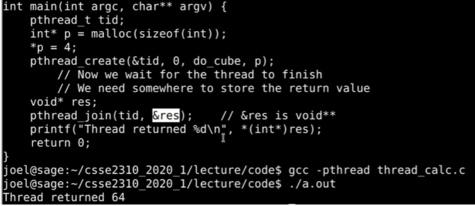
Elsewhere:

```
    struct Params* p = malloc(sizeof(struct Params));
    p->items = items1;
    p->s = "target";
    p->limits = 10;
    pthread_create(&tid, 0, do_things_wrapper, p);
```

What's a thread id?

- `<pthread.h>` declares a `pthread_t` type.
- It is an *opaque* type
  - Makes printing it in debugging awkward
  - (On `moss` it turns out to be `unsigned long` but you can't rely on that)

Where does thread return go?

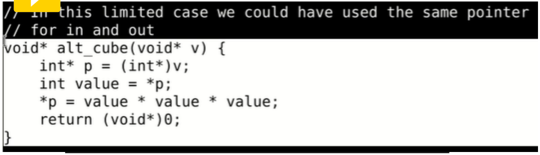


```
int main(int argc, char** argv) {
    pthread_t tid;
    int* p = malloc(sizeof(int));
    *p = 4;
    pthread_create(&tid, 0, do_cube, p);
        // Now we wait for the thread to finish
        // We need somewhere to store the return value
    void* res;
    pthread_join(tid, &res);    // &res is void**
    printf("Thread returned %d\n", *(int*)res);
    return 0;
}
joel@sage:~/csse2310_2020_1/lecture/code$ gcc -pthread thread_calc.c
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
Thread returned 64
```

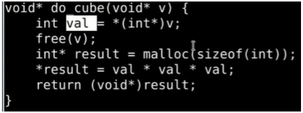
See `thread_calc.c`

Note:

- Since the thread function returns `void*`, to allow `pthread_join` to modify a variable, you need to pass in `void**`.
- Most pthread functions[2] return error codes, which you should check.
- `pthread_exit(V)` ends the thread and sets its result to `V`;



```
// In this limited case we could have used the same pointer
// for in and out
void* alt_cube(void* v) {
    int* p = (int*)v;
    int value = *p;
    *p = value * value * value;
    return (void*)0;
}
```



```
void* do_cube(void* v) {
    int val = *(int*)v;
    free(v);
    int* result = malloc(sizeof(int));
    *result = val * val * val;
    return (void*)result;
}
```

[2] in particular `pthread_create()` and `pthread_join()`

Zombie threads?

- Yes, the doco indicates these exist.
- `pthread_join()` deals with them in a similar way to `wait()` reaping zombie processes.
- Threads don't terminate independently due to signals
  - Unhandled signals will take out the whole process.
- `pthread_detach(tid)` tells the system to clean up zombie threads automatically.
  - You can use the second argument of `pthread_create` to start a thread detached.

Killing threads?

Can you "kill" threads?

- (as previously noted) not using signals.
- `pthread_cancel(tid)` exists.

In general, it is difficult to do safely.
A "safer" approach is to use a shared variable that the thread checks.

```
void* thread_fn(void* v) {
    bool* stop=...       // get this some how
        // other things
    while (!*stop) {
        // do some work
    }
}
```

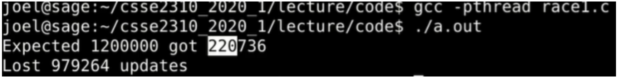
So, yes you can but you shouldn't.

pthread_self()

```
pthread_t pthread_self(void);
```

Returns the thread id of the current thread.

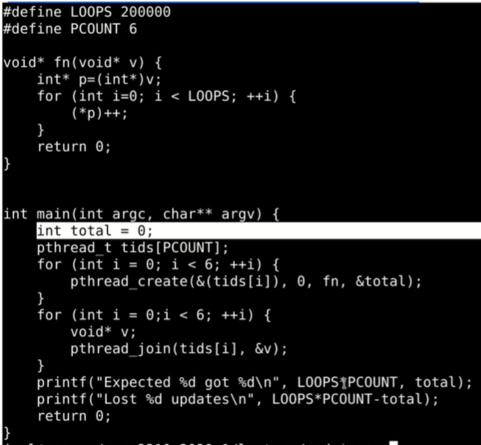
Racing



```
joel@sage:~/csse2310_2020_1/lecture/code$ gcc -pthread race1.c
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
Expected 1200000 got 220736
Lost 979264 updates
```

See `race1.c`

Problem: `value++` is not atomic[3]



```
#define LOOPS 200000
#define PCOUNT 6

void* fn(void* v) {
    int* p=(int*)v;
    for (int i=0; i < LOOPS; ++i) {
        (*p)++;
    }
    return 0;
}


int main(int argc, char** argv) {
    int total = 0;
    pthread_t tids[PCOUNT];
    for (int i = 0; i < 6; ++i) {
        pthread_create(&(tids[i]), 0, fn, &total);
    }
    for (int i = 0;i < 6; ++i) {
        void* v;
        pthread_join(tids[i], &v);
    }
    printf("Expected %d got %d\n", LOOPS*PCOUNT, total);
    printf("Lost %d updates\n", LOOPS*PCOUNT-total);
    return 0;
}
```

[3] ie indivisible (greek atomos)

Racing

Let's use a lock.

See `race2.c`

Hmm. . . Our lock code is not atomic either.

Running `race3.c` shows that locks can be made to work.

Thread coordination

Problems

The `race2` implementation fails to do two desirable things:

1. Ensure mutual exclusion
   - Much simpler with hardware assistance.
   - Normally access this via library calls.
2. Avoid busy waiting

Note: For a mutex algorithm that doesn't rely on[4] hardware assistance see Peterson's algorithm.

[4] much

Semaphores

An opaque type which represents an integer value.

Two atomic operations :

- `sem_wait()`
  - If the value is > 0 decrement it by 1.
  - If the value $== 0$, stop the process until value > 0, then attempt to decrement.
- `sem_post()`
  - Increment the value.

If the p threads are waiting on the semaphore and a `post()` occurs, only one of the threads unblocks, the others stay blocked.

Does everyone get a turn eventually?

- “Starvation” is beyond the scope of this course

Semaphores in action

See race3.c

- sem_init() — set the initial value of the semaphore
- sem_wait() — decrement the semaphore
- sem_post() — increment the semaphore
- sem_destroy() --- clean up

Note: Always pass pointers to semaphores, do not copy the value itself.