EC 440 – Introduction to Operating Systems Project 4 – Discussion

Manuel Egele

Department of Electrical & Computer Engineering

Boston University

Goals – Thread Local Storage

1. Provide protected memory regions for threads
2. Understand the basic concepts of memory management

Why Protected Memory for Threads?

- By default, all threads share the same address space
- Easy sharing of information
- But no protection from misbehaving threads
  – Thus, let's implement this protection
- Similar to Threads themselves, can be implemented in user space or kernel space
  – We'll implement it in user space

Thread Local Storage (TLS) Library

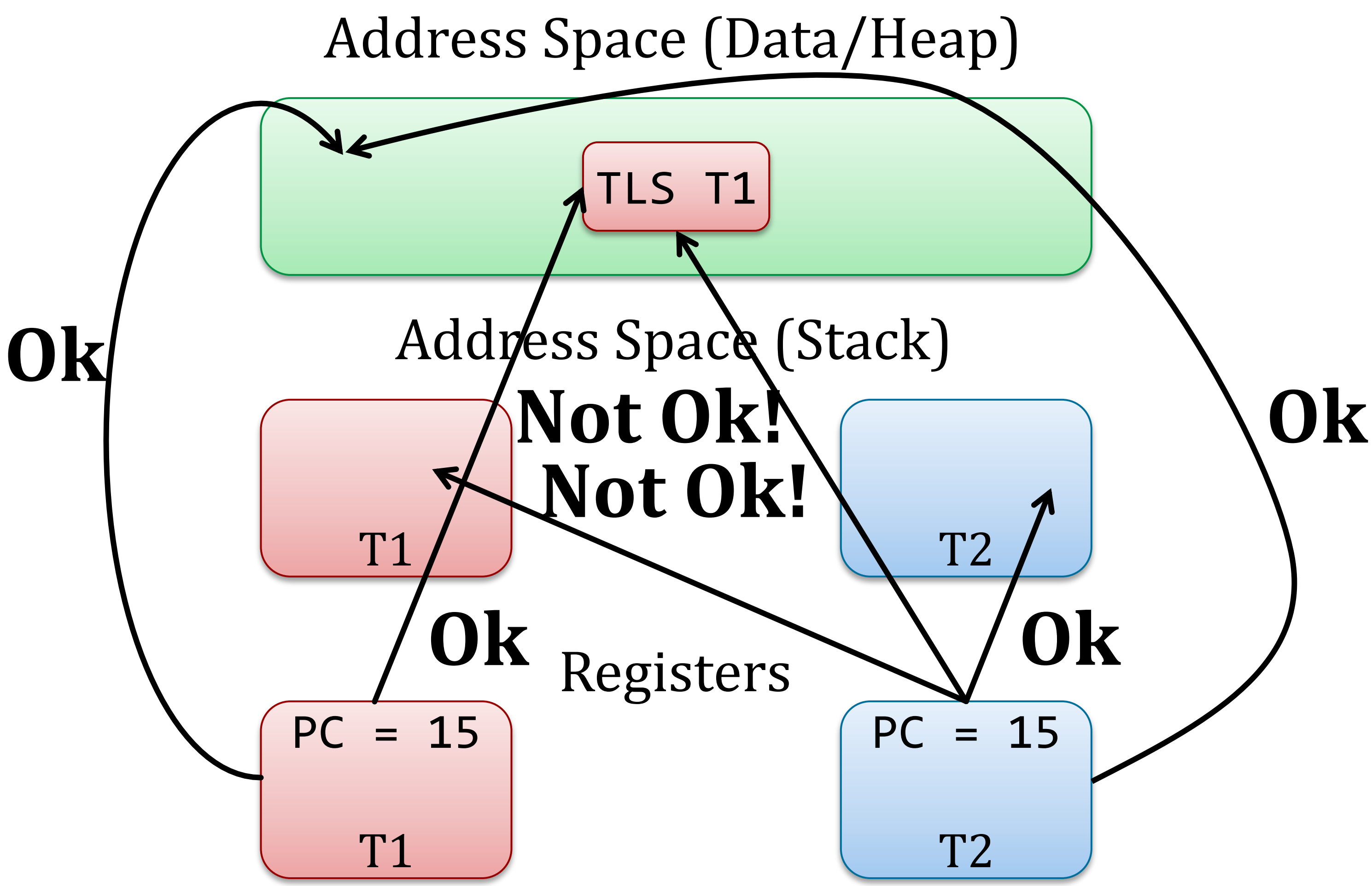
Threads
- Were the topic of projects 2 & 3 … i.e., should not be a matter of debate anymore

Storage
- An area of memory where data can be written to & read from

Local
- Each storage area is *local* (i.e., private) to one thread
- i.e., Thread T1 cannot read/write the TLS area of Thread T2

Copy-on-Write (COW) Semantics
- … more on that later …

Threads – TLS



Address Space (Data/Heap)
TLS T1
Address Space (Stack)
Ok
Not Ok!
Not Ok!
Ok
T1
T2
Ok
Registers
Ok
PC = 15
T1
PC = 15
T2

TLS – Local Storage

Local Storage – Protection

- Protect data tampering from other threads (i.e., no other thread can write to my TLS)
- Protect data "stealing" from other threads (i.e., no other thread can read from my TLS)

What if thread violates the protection?

- Terminate the offending thread!

How can we detect protection violations?

Protection, Violation, & Detection

Need to protect against read & write

– Remember the page-permission bits from the lecture, esp. R(ead) and W(rite)

How can violations occur?

– If R (W) bit is cleared reading (writing) from (to) the corresponding memory area will trigger a segmentation fault

– But: segfault kills the process

How to detect violations?

– Segfault is just another signal, i.e., we can catch it with a signal handler

Enabling Protection

- All TLS sections have R/W bits cleared unless they're actively in use:
  - (i.e., only during calls to tls_read, tls_write)
- We need memory for the TLS sections. How do we allocate that memory?
  - use mmap()!
  - why not simply malloc()?
    - Granularity of protection bits is per virtual memory page (e.g., 4k)
    - malloc() allocates memory w/o regard for page boundaries and might put two different TLSs into the same page
    - mmap() allocates memory with page granularity and aligned to page boundaries (i.e., exactly what we need!)
    - All TLS areas are rounded up to the next page-size

Shades of Segfaults

A segfault happened, now what?

Two cases:

1. Thread Ti (i != 1) accesses T1's TLS
    a. Kill Ti (pthread_exit())
2. A regular segfault, Ti tries to access memory that's not a TLS but the access is inconsistent with page permission settings
    a. Raise segfault to the process (i.e., process will die)

How do we know which case happened?

Which Thread Caused the SEGV For What Address?

Caused a SEGV

- Our signal handler for SIGSEGV is invoked

Which thread?

- pthread_self()

What address?

- Signal handler (man sigaction, esp. fields in siginfo_t)
- `sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
- ```
  struct sigaction { …
    void (*sa_sigaction)(int, siginfo_t *, void *)
  … }
  ```
  ```

siginfo_t struct with these fields

```
siginfo_t {
    int      si_signo;     /* Signal number */
    int      si_errno;     /* An errno value */
    int      si_code;      /* Signal code */
    int      si_trapno;    /* Trap number that caused hardware-generated signal (unused on most architectures) */
    pid_t    si_pid;       /* Sending process ID */
    uid_t    si_uid;       /* Real user ID of sending process */
    int      si_status;    /* Exit value or signal */
    clock_t  si_utime;     /* User time consumed */
    clock_t  si_stime;     /* System time consumed */
    sigval_t si_value;     /* Signal value */
    int      si_int;       /* POSIX.1b signal */
    void    *si_ptr;       /* POSIX.1b signal */
    int      si_overrun;   /* Timer overrun count; POSIX.1b timers */
    int      si_timerid;   /* Timer ID; POSIX.1b timers */
    void    *si_addr;      /* Memory location which caused fault */
    long     si_band;      /* Band event (was int in glibc 2.3.2 and earlier) */
    int      si_fd;        /* File descriptor */
    short    si_addr_lsb;  /* Least significant bit of address (since Linux 2.6.32) */
    void    *si_lower;     /* Lower bound when address violation occurred (since Linux 3.19) */
    void    *si_upper;     /* Upper bound when address violation occurred (since Linux 3.19) */
    int      si_pkey;      /* Protection key on PTE that caused fault (since Linux 4.6) */
    void    *si_call_addr; /* Address of system call instruction (since Linux 3.5) */
    int      si_syscall;   /* Number of attempted system call (since Linux 3.5) */
    unsigned int si_arch;  /* Architecture of attempted system call (since Linux 3.5) */
}
```

API

Create/Destroy TLS

```
int tls_create(unsigned int size);
int tls_destroy();
int tls_clone(pthread_t tid);
```
… later

Write to a TLS

```
int tls_write(unsigned int offset,
    unsigned int length,
    char *buffer);
```

Read from a TLS

```
int tls_read(unsigned int offset,
    unsigned int length,
    char *buffer);
```

tls_create

`int tls_create(unsigned int size)`

- Creates a local storage area for the **current** thread of a given size
- Returns 0 on success
- Error: return -1
  - if current thread already has a LSA
  - size <= 0

tls_write

```
int tls_write(unsigned int offset,
    unsigned int length,
    char *buffer);
```

- Reads `length` bytes from the memory location pointed to by `buffer` and writes them into the local storage area of the currently executing thread, starting at position `offset`.
- Returns 0 on success
- Error: return -1
  - if current thread does not have an LSA
  - if offset+length > size of LSA

tls_read

```
int tls_read(unsigned int offset,
     unsigned int length,
     char *buffer);
```

- Reads `length` bytes from the LSA of the currently executing thread, starting at position `offset` and writes into memory location pointed to by `buffer`.
- Returns 0 on success
- Error: return -1
  - if current thread does not have an LSA
  - if offset+length > size of LSA

tls_destroy

```
int tls_destroy();
```

– Frees local storage area for **current** thread.

– Returns 0 on success

– Error: return -1

  - if current thread does not have an LSA

tls_clone

`int tls_clone(pthread_t tid);`

- Clones the LSA of the target thread `tid` as CoW.
- Copy on Write (CoW):
  - Storage areas of both threads initially refer to the same memory pages
  - Only when one thread writes to a shared region (i.e., using tls_write()), then the TLS library creates a private copy of the region that is written
  - Other areas must remain shared!
- Returns 0 on success
- Error: return -1
  - if target thread does not have a LSA
  - if current thread already has a LSA

Assumptions

1. Whenever a thread calls tls_read or tls_write, you can temporarily unprotect this thread's local storage area
   – i.e., race condition where other threads can tamper with TLS w/o getting punished
2. We will work with page (vs. byte) granularity
   a. If T2 clones T1's TLS, and T2 writes one byte to it's own (CoW) TLS
   b. Instead of copying one byte, we make a copy of the entire page that contains the target byte

One complexity

It is possible for more than two threads to share the same LSA

- Example
  T1.tls_create(8192)
  T2.clone(T1)
  T3.clone(T1)
- T1, T2, and T3 share the same local storage area
- CoW applies to all three threads.

Useful Library Functions

`mmap(2)`

- Helps to create local storage that cannot be accessed directly by thread
- No read/write permission to thread
- Memory obtained is aligned to start of page
- Allocates memory in multiples of page size

`mprotect(2)`

- Threads cannot access memory assigned by mmap
- Use mprotect to unprotect the memory before read/write
- Re-protect memory when the operation is complete

Need Some Data-structures

```
typedef struct thread_local_storage
{
  pthread_t tid;
  unsigned int size;      /* size in bytes               */
  unsigned int page_num;  /* number of pages             */
  struct page **pages;    /* array of pointers to pages  */
} TLS;


struct page {
  unsigned int address;   /* start address of page       */
  int ref_count;          /* counter for shared pages    */
};
```

Mapping of a Thread to a TLS

Need a global data structure to keep this mapping (e.g., fixed-sized array (limited number of concurrent threads inherited from HW2), linked list, hash map, etc.)

```
struct hash_element
{
  pthread_t tid;
  TLS *tls;
  struct hash_element *next;
};

struct hash_element* hash_table[HASH_SIZE];
```

Initialize on First Run

```
void tls_init()
{
  struct sigaction sigact;

  /* get the size of a page */
  page_size = getpagesize();

  /* install the signal handler for page faults (SIGSEGV, SIGBUS) */
  sigemptyset(&sigact.sa_mask);
  sigact.sa_flags = SA_SIGINFO; /* use extended signal handling */
  sigact.sa_sigaction = tls_handle_page_fault;

  sigaction(SIGBUS, &sigact, NULL);
  sigaction(SIGSEGV, &sigact, NULL);

  initialized = 1;
}
```

Difference between SIGSEGV and SIGBUS?

Handle SIGSEGV

Does what, exactly?

```
void tls_handle_page_fault(int sig, siginfo_t *si, void *context)
{
  p_fault = ((unsigned int) si->si_addr) & ~(page_size - 1);
```

1. check whether it is a "real" segfault or because a thread has touched forbidden memory

   a. make a brute force scan through all allocated TLS regions

   b. for each page:

```
      if (page->address == p_fault) {
        pthread_exit(NULL);
      }
```

2. a normal fault - install default handler and re-raise signal

```
    signal(SIGSEGV, SIG_DFL);
    signal(SIGBUS, SIG_DFL);
    raise(sig);
}
```

tls_create

```
int tls_create(unsigned int size) {
  if (!initialized)
    tls_init();
```

1. Error handling:
   – check if current thread already has a LSA
   – check size > 0 or not
2. Allocate TLS using calloc
3. Initialize TLS,
   – add TLS->threadid,
   – TLS->size and
   – TLS->page_num

tls_create

4. Allocate TLS->pages, array of pointers using calloc

5. Allocate all pages for this TLS

```
for each page i:
    struct page *p;
    p->address = (unsigned int)mmap(0,
        PAGESIZE, 0, MAP_ANON | MAP_PRIVATE,
        0, 0);
    p->ref_count = 1
    tls->pages[i] = p;
```

6. Add this threadid and TLS mapping to global data structure

}

tls_destroy

```
int tls_destroy(){
```

1. Error handling:
   - check if current thread has LSA
2. Clean up all pages

   Check each page whether it's shared
   a) If not shared, free the page
   b) If shared, can't free as other threads still rely on it. But…?
3. Clean up TLS
4. Remove the mapping form global structure

```
}
```

Helper Function: tls_protect()

```
void tls_protect(struct page *p)
{
  if (mprotect((void *) p->address, page_size, 0)) {
    fprintf(stderr, "tls_protect: could not protect page\n");
    exit(1);
  }
}
```

Helper Function: tls_unprotect()

```
void tls_unprotect(struct page *p)
{
  if (mprotect((void *) p->address, page_size, \
        PROT_READ | PROT_WRITE)) {
    fprintf(stderr, "tls_unprotect: could not unprotect page\n");
    exit(1);
  }
}
```

tls_read()

```
int tls_read(unsigned int offset,
unsigned int length, char *buffer)
```

{

1. Error handling:
   – check if current thread has a LSA
   – check if offset+length > size
2. Unprotect all pages belonging to thread’s TLS
3. Perform read operation
4. Reprotect all pages belonging to thread’s TLS

}

Read Operation

```
 for (cnt = 0, idx = offset; idx < (offset +
length); ++cnt, ++idx) {
    struct page *p;
    unsigned int pn, poff;

    pn = idx / page_size;
    poff = idx % page_size;

    p = tls->pages[pn];
    src = ((char *) p->address) + poff;

    buffer[cnt] = *src;
  }
```

Terribly inefficient,
feel free to optimize.

tls_write()

```
int tls_write(unsigned int offset,
unsigned int length, char *buffer) {
```

1. Error handling:
   – check if current thread has a LSA
   – check if offset+length > size
2. Unprotect all pages belonging to thread's TLS
3. Perform write operation (next slide)
4. Reprotect all pages belonging to thread's TLS

}

Write Operation

```
/* perform the write operation */
 for (cnt = 0, idx = offset; idx < (offset + length); ++cnt, ++idx) {
   struct page *p, *copy;
   unsigned int pn, poff;
   pn = idx / page_size;
   poff = idx % page_size;
   p = tls->pages[pn];
   if (p->ref_count > 1) {

     /* this page is shared, create a private copy (COW) */
       (next slide)
   }

   dst = ((char *) p->address) + poff;
   *dst = buffer[cnt];
 }
```

Terribly inefficient,
feel free to optimize.

CoW Implementation

```
{

    /* this page is shared, create a private copy (COW) */
    copy = (struct page *) calloc(1, sizeof(struct page));
    copy->address = (unsigned int) mmap(0,
      page_size, PROT_WRITE,
      MAP_ANON | MAP_PRIVATE, 0, 0);
    copy->ref_count = 1;
    tls->pages[pn] = copy;

    /* update original page */
    p->ref_count--;
    tls_protect(p);
    p = copy;
  }
```

tls_clone

`int tls_clone(pthread_t tid) {`

1. Error handling
   – check if current thread already has a LSA
   – check whether target thread has a LSA
2. Do Cloning, allocate TLS
3. Copy pages (not contents!), adjust reference counts
   – Note, per proj. description and CoW semantics do not create a copy of the data itself!
   – Make cloned' page entries point to original data-pages
   – CoW is handled in tls_write
4. Add this thread/TLS mapping to global structure

Questions?