Engineering &
Computer Science
Syracuse University

CIS 657 – Principles of Operating Systems

Topic: Concurrency – intro + thread API

Endadul Hoque

Acknowledgement

- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

Thread

- A new abstraction for a single running process

- Multi-threaded program
  - A multi-threaded program has more than one point of execution.
  - Multiple PCs (Program Counters)
  - They share the same address space.

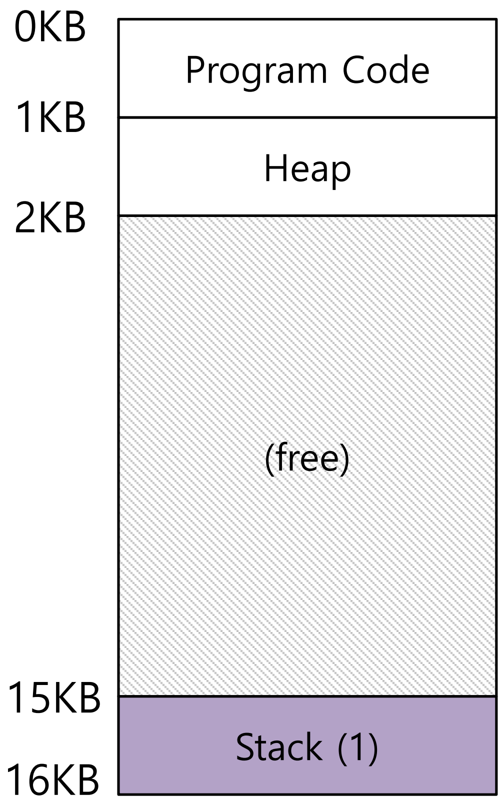
Context switch between threads

- Each thread has its ***own*** program counter and set of registers.
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread.

- When switching from running one (T1) to running the other (T2),
  - The register state of T1 will be saved.
  - The register state of T2 will be restored.
  - The address space remains the same.

Address Spaces

- There will be one stack per thread (**thread-local** storage)

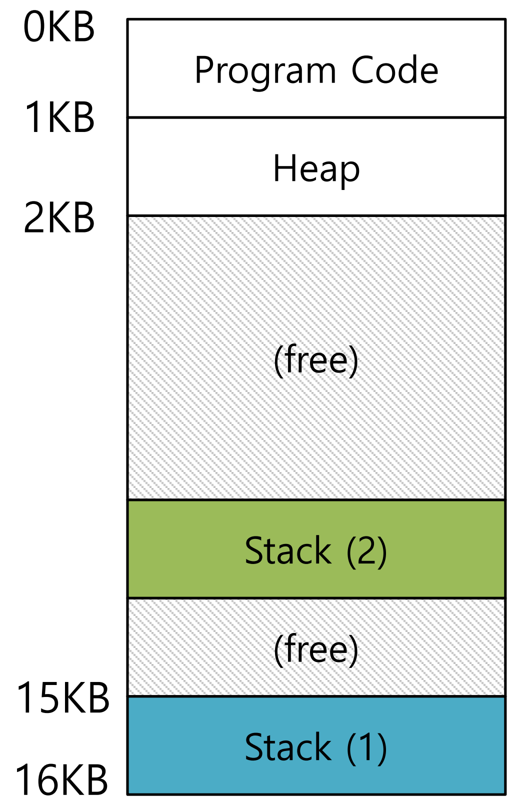


A Single-Threaded Address Space

The code segment:
where instructions live

The heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

(it grows upward)
The stack segment:
contains local variables
arguments to routines,
return values, etc.



Two threaded Address Space

Why use Threads?

- Two major reasons
  - **Parallelism**:
    - A thread per CPU to speed up the process
  - **Avoid blocking** of progress due to slow I/O:
    - threading enables overlap of I/O with other activities within a single process
- Why not use multi-processes?
  - Threads share an address space and thus make it easy to share data
  - Using multi-processes is <u>suitable for logically separate</u> tasks, with little to no data sharing involved.

Example: Thread Traces

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: **Simple Thread Creation Code (`t0.c`)**

Example: Thread Traces

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Example: Thread Traces

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Example: Thread Traces

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Sharing Data: Race Condition

- Example with two threads
  - **`counter = counter + 1`** (default is **50**)
  - We expect the result to be **52**. However,

```
100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c
```

OS	Thread1	Thread2	(after instruction) PC	%eax	counter

Sharing Data: Race Condition

- Example with two threads
  - **`counter = counter + 1`** (default is **50**)
  - We expect the result to be **52**. However,

```
100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c
```

OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	`mov 0x8049a1c, %eax`		105	50	50
	`add $0x1, %eax`		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		`mov 0x8049a1c, %eax`	105	50	50
		`add $0x1, %eax`	108	51	50
		`mov %eax, 0x8049a1c`	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	`mov %eax, 0x8049a1c`		113	51	$\boxed{51}$

Sharing Data: Race Condition

- Example with two threads
  - **`counter = counter + 1`** (default is **50**)
  - We expect the result to be **52**. However,

```
100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c
```

(after instruction)

Race conditions (aka data race) can not only produce wrong results, but also produce different results across different runs – giving rise to **indeterminate** behavior

interrupt

save T2's state

restore T1's state

	108	51	50
`mov %eax, 0x8049a1c`	113	51	$\boxed{51}$

Critical section

- A piece of code that accesses a shared variable (aka resources) and must **not be concurrently executed** by more than one thread.
  - Multiple threads executing critical section can result in a race condition.
  - Desired property: **mutual exclusion** guarantee for critical sections
  - One possible solution: **Atomic** execution of critical sections
    - Atomicity notion: "**all or none**"
- Based on a hardware synchronization primitives (i.e., special instructions), we can build complex abstractions to enable multi-threaded code produce correct results reliably.

Some interesting lines from Chapter 26

- It is a wonderful and hard problem, and should make your mind hurt (a bit).
- If it doesn't, then you don't understand!
- Keep working until your head hurts; you then **know** you're headed **in the right direction**.

Thread API

Thread Creation

- How to create and control threads?

```
#include <pthread.h>

int
pthread_create(      pthread_t*       thread,
               const pthread_attr_t* attr,
                     void*            (*start_routine)(void*),
                     void*            arg);
```

  - `thread`: Used to interact with this thread.
  - `attr`: Used to specify any attributes this thread might have.
    - Stack size, Scheduling priority, …
  - `start_routine`: pointer to the function this thread will start executing from.
  - `arg`: the argument to be passed to the function (`start_routine`)
    - *a **void pointer*** allows us to pass in ***any type of*** argument.

Thread Creation

```
#include <pthread.h>

typedef struct __myarg_t {
         int a;
         int b;
} myarg_t;

void *mythread(void *arg) {
         myarg_t *m = (myarg_t *) arg;
         printf(“%d %d\n”, m->a, m->b);
         return NULL;
}

int main(int argc, char *argv[]) {
         pthread_t p;
         int rc;

         myarg_t args;
         args.a = 10;
         args.b = 20;
         rc = pthread_create(&p, NULL, mythread, &args);
         …
}
```

Wait for a thread to complete

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- `thread`: Specify which thread *to wait for*
- `value_ptr`: A pointer to capture the <u>return value</u>
  - Because `pthread_join()` routine changes the value, you need to pass in a pointer to that value.

Wait for a thread to complete

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Wait for a thread to complete

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);   // this thread has been
                                  // waiting inside of the
                                        // pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

```
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Example: Dangerous code

- Be careful with how values are returned from a thread.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

  - When the variable `r` returns, it is automatically de-allocated.

Example: Simpler Argument Passing to a Thread

- Just passing in a single value

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf(“%d\n”, m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf(“returned %d\n”, m);
    return 0;
}
```

Locks

- Provide mutual exclusion to a critical section
  - Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - No other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Locks

- All locks must be properly initialized.
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks

- Check errors code when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- These two calls are used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - `trylock`: return failure if the lock is already held
  - `timedlock`: return after a timeout

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads.

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

  – `pthread_cond_wait`:
    - Put the calling thread to sleep.
    - Wait for some other thread to signal it.
  – `pthread_cond_signal`:
    - Notify at least one of the threads that are blocked on the condition variable

Condition Variables

- A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (ready == 0)
        pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);
```

  - The wait call releases the lock when putting said caller to sleep.
  - Before returning after being woken, the wait call re-acquire the lock.

- A thread calling signal routine:

```
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

Condition Variables

- The waiting thread **re-checks** the condition in a while loop, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (ready == 0)
        pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);
```

  – Without rechecking, the waiting thread will continue thinking that the condition has changed <u>*even though it has not*</u>.

Condition Variables

- Poor way to do this.
  - A thread calling wait routine:

```
while(ready == 0)
        ; // spin
```

  - A thread calling signal routine:

```
ready = 1;
```

  - It performs poorly in many cases. → just wastes CPU cycles.
  - It is error prone.

Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the pthreads library, by adding the `-pthread` flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

  - For more information,

```
man -k pthread
```

Reading Material

- **Chapter 26-27** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)
  http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf
  http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf

Questions?