Concurrency: An Introduction

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 Counter)
  - They share the same address space.

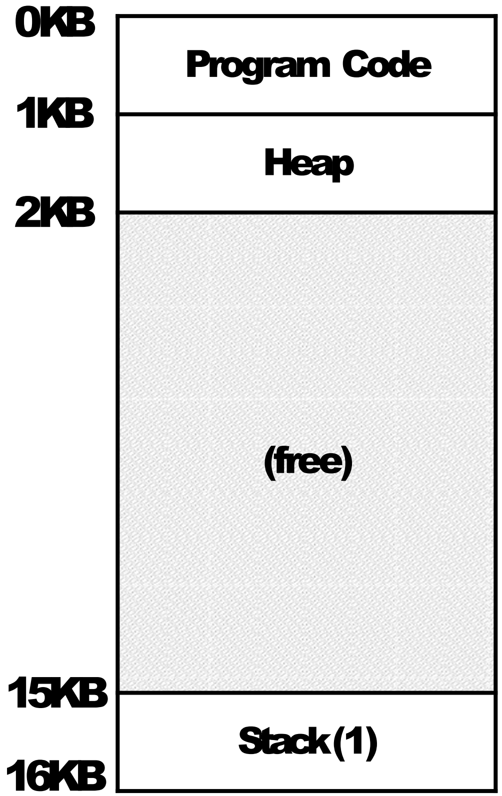
Context switch between threads

- **Each thread has its own <u>program counter</u> and <u>set of registers</u>.**
  - **One or more** thread control blocks(TCBs) **are needed to store the state of each thread.**
  - **All of then within a common PCB**

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

The stack of the relevant thread

- There will be **one stack per thread**.

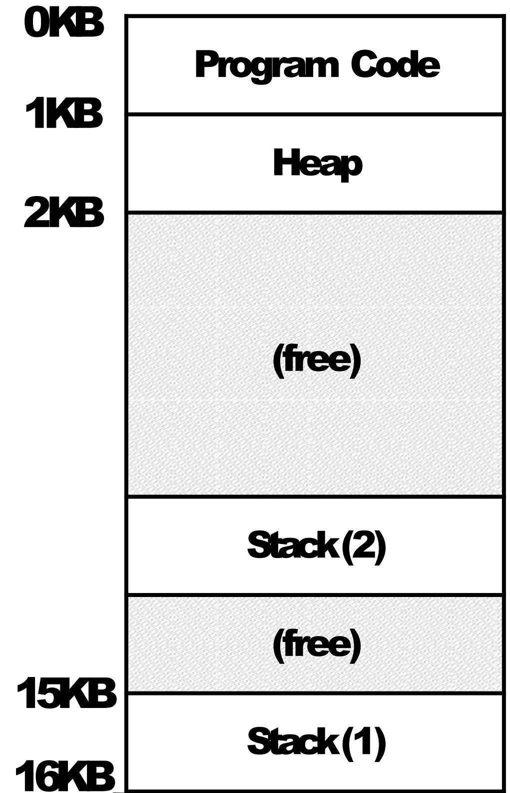


A Single-Threaded Address Space

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.**



Thread- local storage

Two threaded Address Space

Why threads?

- **Performance**
  - **Parallelism is the only way to use translate multiple cores into performance**
  - **Parallelization: from single-threaded programs to multi-threaded**
- **Convenience**
  - **Way to overlap I/O with useful work: approach of server-base applications such as web-servers, DBMS, etc..**
- **Why threads and not processes?**
  - **In threads is much easier to share data**
  - **Less pressure over the memory**
  - **Processes when the task are separated with little (to none) sharing**

Example

```
#include <stdio.h>
#include <assert.h>
#include <pthread.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");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

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

Possible outcomes

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”		

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”		

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”		

Uh Oh

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    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: done with both (counter = %d)\n", counter);
    return 0;
}
```

Possible outcomes

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

The heart of the problem: : Uncontrolled Scheduling

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

			(after instruction)		
OS	**Thread1**	**Thread2**	**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 (TCB)					
restore T2's state (TCB)			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	50
	`mov %eax, 0x8049a1c`		113	51	**51**

The wish for atomicity

- **Do the read and modification of the memory in a single step**
  - **i.e. "all or nothing"!**

- **How ho handle complex data? (v.gr. a b-tree)**
  - **Use some atomic hardware support (called synchronization primitives) to construct OS support**

- **A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.**
  - **Multiple threads executing critical section can result in a race condition.**
  - **Need to support** atomicity **for critical sections (**mutual exclusion**)**

Locks

- **Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).**

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

One more problem: Waiting for another/s

- **Some times the thread interaction is wait for another thread**
  - **V.gr. When a thread should wait to another that had issued a I/O**
  - **Need to be slept until the other thread receives the I/O end**
- **Some times the action of multiple threads should be synchronous**
  - **V.gr. Many threads are performing in parallel a iteration in a numerical problem**
  - **All threads should start the next iteration at once (barrier)**
- **This sleeping/waking cycle will be controlled by** condition variables

Interlude: 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 (OUT).
- `attr`: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, … (IN)
- `start_routine`: the function this thread start running in (IN)
- `arg`: the argument to be passed to the function (`start routine`) (IN/OUT)
  - *a void pointer* allows us to pass in *any type of* argument.
- Returns 0 if went good (a error code otherwise: EAGAIN, EINVAL, EPERM)

Thread Creation

Thread Creation (Cont.)

- If `start_routine` instead required another type argument, the declaration would look like this (example):
  - An integer argument:

```
int
pthread_create(…, // first two args are the same
                    void*  (*start_routine)(int),
                    int     arg);
```

  - Input is anything (usually a pointer to struct for multiple arguments or even internal returns), return an integer:

```
int
pthread_create(…, // first two args are the same
                    int  (*start_routine)(void*),
                    void*     arg);
```

Example: Creating a Thread

```
#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 we want to put the return value of the start routine (ouch!)
  - Because `pthread_join()` routine changes the value, you need to pass in a pointer to that value.
- Returns 0 if good, or EINVAL, ESRCH if err

Example: Waiting for Thread Completion

```
#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;
}

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);
printf("returned %d %d\n", m->x, m->y);
return 0;
}
```

Example: Dangerous code

- Be careful with <u>how values are returned</u> 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.
- Don't malloc here! (memory leak prone) [bad example before]
  - Better to be consistent a allocate and free in parent

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;
}
```

From a practical perspective
using threads this way is pointless!
(just do a procedure call)

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 (Cont.)

Locks (Cont.)

- All locks must be properly initialized (i.e. unlocked value).
  - 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 && "Error in mutex init");
```

Locks (Cont.)

- 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 && "Error in acquire");
}
```

- 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
  - timelock: return after a timeout

Not a bad idea to define a wrapper: much cleaner code

```c
#ifndef __MYTHREADS_h__
#define __MYTHREADS_h__

#include <pthread.h>
#include <assert.h>
#include <sched.h>

void
Pthread_mutex_lock(pthread_mutex_t *m)
{
    int rc = pthread_mutex_lock(m);
    assert(rc == 0);
}

void
Pthread_mutex_unlock(pthread_mutex_t *m)
{
    int rc = pthread_mutex_unlock(m);
    assert(rc == 0);
}

void
Pthread_create(pthread_t *thread, const pthread_attr_t *attr,
               void *(*start_routine)(void*), void *arg)
{
    int rc = pthread_create(thread, attr, start_routine, arg);
    assert(rc == 0);
}

void
Pthread_join(pthread_t thread, void **value_ptr)
{
    int rc = pthread_join(thread, value_ptr);
    assert(rc == 0);
}

#endif // __MYTHREADS_h__
```

Locks (Cont.)

Locks (Cont.)

- These two calls are also 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
  - **`timelock`**: return after a timeout or after acquiring the lock

Condition Variables

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`:
  - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables (Cont.)

- A thread calling wait routine:

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

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &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);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Condition Variables (Cont.)

- 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 init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not*.
  - For example if multiple threads are waiting and only one should grab the data (producer-consumer)

Condition Variables (Cont.)

Condition Variables (Cont.)

- Don't ever to this.
  - A thread calling wait routine:

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

  - A thread calling signal routine:

```
initialized = 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
```

Thread API Use Guidelines

- Keep it simple
  - Tricky thread interactions lead to (hard to find) bugs
- Minimize thread interaction
  - Limits scalability
- Initialize mutex and cond vars
- Check always return codes
- Be careful how to pass arguments and get values:
  - A good practice is to allocate/free memory in the calling thread
  - Be careful with heap
- Each thread has his own stack
- Always use cond. variables to signal between threads
- **Read the man pages**