Concurrency

Piece #2

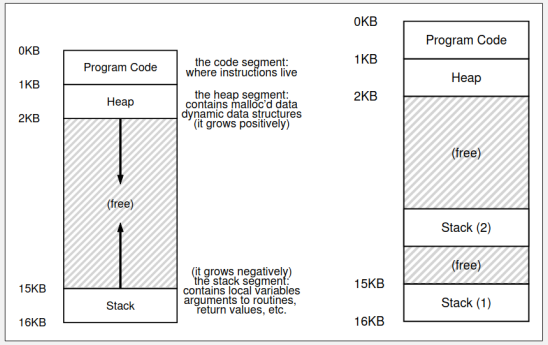
- virtualization – CPU, memory
- **concurrency**
- persistence

Threads

- a new abstraction: a *thread* is a single point of execution
- threads vs processes
  - a *process* is a running program with its own address space
  - a *thread* is a single path of execution within a process
    - each thread has its own stack and registers but the address space is shared
      - multiple threads within a single program can access the same data
    - switching between threads requires a context switch
      - need a *thread control block* (TCB) to store the state of each thread
      - thread context switch requires save/restore of registers but not changing the address space (page table)

Single vs Multi-threaded Address Spaces

single-threaded address space



multi-threaded address space

- each point of execution requires its own stack for parameters, local variables, return values
  - a multi-threaded process has multiple stacks

Advantages of Threads

- parallelism
  - multiple threads can be scheduled on multiple CPUs
- responsiveness
  - other threads can run while one is blocked waiting for I/O
  - allows overlap of I/O and other activities within a program
- shared address space
  - threads can communicate through shared memory
  - lighter weight than processes – cheaper to create and switch between
- use threads when tasks need shared memory
- use processes for logically separate tasks

Concurrency Issues

- shared memory introduces concurrency problems
  - a *critical section* is a piece of code that accesses a shared resource
  - a *race condition* occurs if multiple threads enter a critical section at the same time and the results depend of the exact timing of the code's execution
  - *mutual exclusion* is required to limit a critical section to one thread at a time

Concurrency Issues

```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;
}
```

- threads are scheduled independently so there is no guaranteed order of execution of steps across threads

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"		


```c
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;
}
```

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

- this can be a problem when shared data is involved

Concurrency Issues

- this problem occurs because a step like

```
counter = counter + 1;
```

turns into multiple assembly language instructions

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

and a context switch can occur at any point

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Synchronization Primitives

- need to ensure *atomicity* for critical sections
  - either all of the instructions happen without interruption, or none of them do
- also commonly need for one thread to *wait* for another to finish a particular task
  - e.g. need to wait for I/O to complete before continuing
- hardware + OS support will allow building a set of *synchronization primitives* so concurrency can be utilize safely

Relevance to OS

- the OS has had to have been concerned with concurrency as long as multiprogramming has existed
  - issues are not limited to shared memory
    - e.g. two processes both attempt to append data to a file – may need to allocate a new block, record its address, and update the file size
- OS provides tools for the user to support multi-threaded programs

POSIX Threads

- POSIX Threads (pthreads) is a standard C API for threads
  - defined by IEEE standard 1003.1c-1995
  - available on many systems (Unix flavors, Linux, macOS, Windows)
- `#include <pthread.h>`
- includes routines for
  - thread management
  - mutexes
  - condition variables
  - synchronization between threads
  - spinlocks
- semaphores are provided by a related API

Thread Creation

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

- `thread` is the structure representing the thread
  - allocated prior to calling `pthread_create()`
  - initialized by `pthread_create()`
- `attr` specifies thread attributes such as the stack size
  - initialized by `pthread_attr_init()`
  - to use defaults, pass `NULL`
- `start_routine` is a *function pointer* to a function which takes a single parameter of type `void *` and returns `void *`
  - the thread runs this function
  - `void *` allows for any type of parameter and return value (cast as needed)
- `arg` is the value passed for the single parameter
  - can be `NULL` if no arguments

Thread Creation

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

- `thread` is the structure representing the thread
  - allocated prior to calling `pthread_create()`
  - initialized by `pthread_create()`
- `attr` specifies thread attributes such as the stack size
  - initialized by `pthread_attr_init()`
  - to use defaults, pass `NULL`
- `start_routine` is a *function pointer* to a function which takes a single parameter of type `void *` and returns `void *`
  - the thread runs this function
  - `void *` allows for any type of parameter and return value (cast as needed)
- `arg` is the value passed for the single parameter

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

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

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

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

user-defined type (to pass two `int`s to the thread)

cast the parameter to the expected type

create structures for the thread and args

pass pointers to the thread and args structures

Thread Completion

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

- `thread` is the structure representing the thread
- `value_ptr` is a pointer to where the return value should be put
  - can be `NULL` if there is no return value or the return value is not desired

Thread Completion

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

- `thread` is the structure representing the thread
- `value_ptr` is a pointer to where the return value should be put
  - can be `NULL` if there is no return value or the return value is not desired

```
void *mythread(void *arg) {
    long long int value = (long long int) arg;
    printf("%lld\n", value);
    return (void *) (value + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    long long int rvalue;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &rvalue);
    printf("returned %lld\n", rvalue);
    return 0;
}
```

cast the parameter to the expected type

cast the return type to match the declaration

capital-letter version refers to `#defines` that check for success (return 0)

Thread Completion

```
typedef struct { int a; int b; } myarg_t;
typedef struct { int x; int y; } myret_t;

void *mythread(void *arg) {
    myret_t *rvals = Malloc(sizeof(myret_t));
    rvals->x = 1;
    rvals->y = 2;
    return (void *) rvals;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myret_t *rvals;
    myarg_t args = { 10, 20 };
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &rvals);
    printf("returned %d %d\n", rvals->x, rvals->y);
    free(rvals);
    return 0;
}
```

user-defined types (to pass two ints to the thread and to get two `int`s back)

cast the return type to match the declaration

capital-letter version refers to #defines that check for success (return 0)

`free` matching the `malloc` that the thread executed

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

never return a reference to memory allocated on the thread's stack

Compiling

- must explicitly link with the pthreads library

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