Concurrency Review

CMPU 334 – Operating Systems
Jason Waterman

Thread

- A new abstraction for **a single running process**

- Multi-threaded program
  - A multi-threaded program has **more than one point of execution**
    - Multiple program counters, one for each thread
    - Also known as **multiple threads of execution**
  - Threads in a process **share the same address space**
  - Each thread has its own stack
  - Each thread has its own private set of registers

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

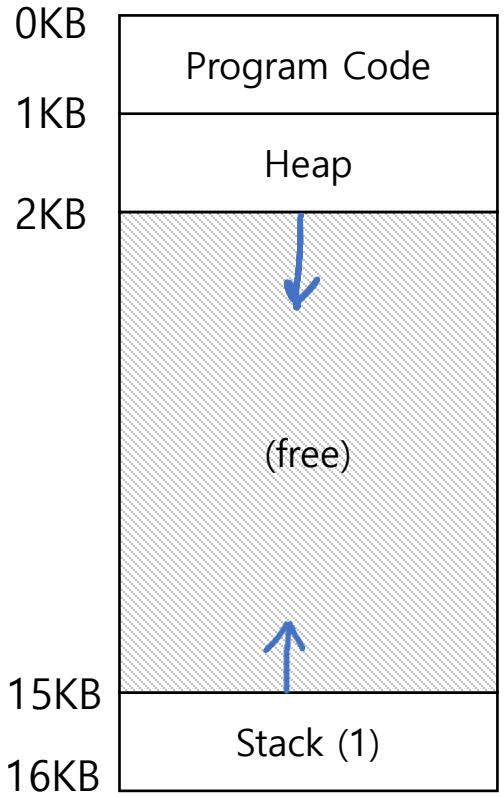
- When switching from running one thread (T1) to running the another (T2):
  - The register state of T1 be saved
  - The register state of T2 restored
  - The **address space remains** the same

Why Use Threads?

- Two major reasons
  1. Parallelism
     - Divide a task among several threads
     - On a system with multiple processors threads can work in parallel with each other
  2. Overlap of I/O with other activities **within** a single program
     - When one thread requests I/O from the system, switch to another thread ready to work
     - Similar to multiprogramming **across** different processes
     - The cost to switch to another thread is less than the cost of a context switch between processes

The stack of the relevant thread

- There is **one stack per thread** of execution

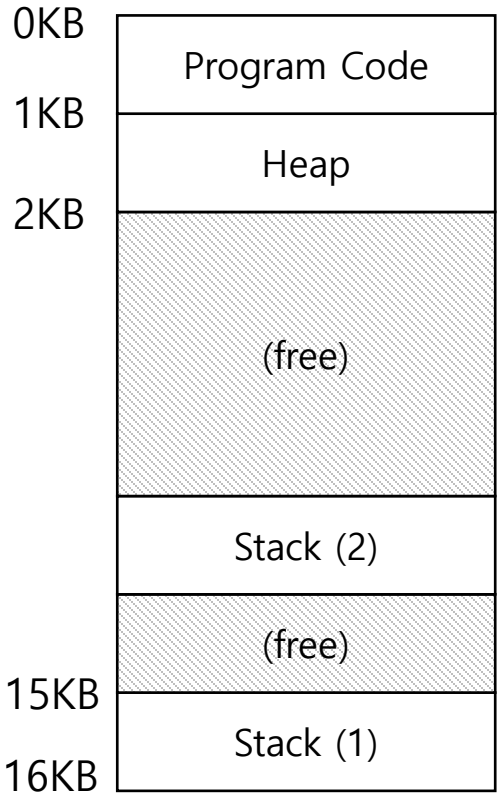


A Single-Threaded Address Space

The code segment: where instructions live

The heap segment: contains malloc'd data dynamic data structures. It grows towards increasing memory addresses

The stack segment: contains local variables arguments to routines, return values, etc. It grows towards decreasing memory addresses



Two threaded Address Space

Example: Creating a Thread

```
#include <pthread.h>
void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}


int main(int argc, char *argv[]) {
    if (argc != 1) {
        fprintf(stderr, "usage: main\n");
    exit(1);
    }

    pthread_t p1, p2;
    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;
}
```

Thread Creation

- How to create and control threads?

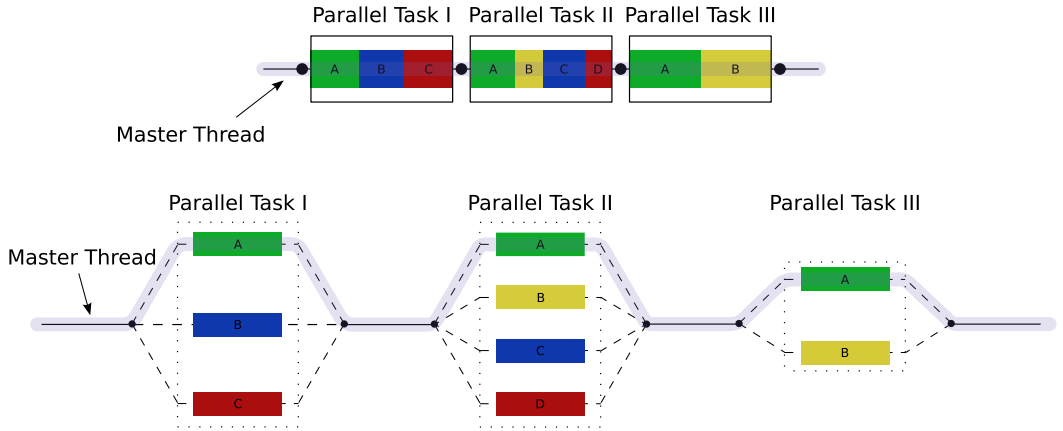
```
#include <pthread.h>

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, etc. – pass in `NULL` if not changing any attributes
  - **`start_routine`**: the function the thread executes when first started
  - **`arg`**: the argument to be passed to the function (`start_routine`)
    - *a void pointer* allows us to pass in *any type of* argument
  - **return value**: on success, returns 0; on error, it returns an error number, and the contents of `thread` are undefined

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 the **return value**
  - Because `pthread_join()` routine changes the value, you need to **pass in a pointer** to the return value (which is of type void *)
  - If you don't care about the return value of the thread, pass in NULL
- The name join comes from the fork-join model of executing parallel programs



By Wikipedia user A1 - w:en:File:Fork_join.svg, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=32004077

Example: Creating a Thread with Arguments

```
#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);
        …
}
```

Example: Returning Data from a Thread

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #include <assert.h>
4   #include <stdlib.h>
5
6   typedef struct __myarg_t {
7       int a;
8       int b;
9   } myarg_t;
10
11  typedef struct __myret_t {
12      int x;
13      int y;
14  } myret_t;
15
16  void *mythread(void *arg) {
17      myarg_t *m = (myarg_t *) arg;
18      printf("%d %d\n", m->a, m->b);
19      myret_t *r = malloc(sizeof(myret_t));
20      r->x = 1;
21      r->y = 2;
22      return (void *) r;
23  }
24
```

```
25  int main(int argc, char *argv[]) {
26      int rc;
27      pthread_t p;
28      myret_t *m;
29
30      myarg_t args;
31      args.a = 10;
32      args.b = 20;
33      pthread_create(&p, NULL, mythread, &args);
34      pthread_join(p, (void **) &m);
35      // this thread has been
36      // waiting inside of the
37      // pthread_join() routine.
38      printf("returned %d %d\n", m->x, m->y);
39      return 0;
40  }
```

Threading: Shared Variables

```
static volatile int counter = 0; // shared variable global to the .c file

// 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");
    Pthread_join(p1, NULL); // join waits for the threads to finish
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Race condition

- Example with two threads and counter = 50
  - counter = counter + 1 (runs twice, once for each thread)
  - We expect the result to be 52

```
100 lw   a0,-608(gp)#<counter>
104 addi a0,a0,1
108 sw   a0,-608(gp)#<counter>
```

OS	Thread1	Thread2	PC (after instruction)	a0	Counter (0x13788)
			100	0	50
	lw a0,-608(gp)		104	50	50
	addi a0,a0,1		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		lw a0,-608(gp)	104	50	50
		addi a0,a0,1	108	51	50
		sw a0,-608(gp)	112	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	sw a0,-608(gp)		112	51	**51**

Critical section

- In the previous example, there are three actions that take place to increase the counter
  - **Read** the current value from memory
  - **Modify** the value
  - **Write** the new value to memory
- These three operations must happen **atomically**
  - Once the read takes place, the thread should not be interrupted by another thread that uses the shared variable until after the modify and write occur
- A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread is called a **critical section**
  - Multiple threads executing critical section can result in a **race condition**
  - Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

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

```
1     lock_t mutex;
2     . . .
3     lock(&mutex);
4     counter = counter + 1;
5     unlock(&mutex);
```

Critical section

Locks

- Provide **mutual exclusion** to a critical section
  - Interface – two operations **lock** and **unlock**

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

  - Usage (without lock initialization and error checking)

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

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

Lock Initialization

- 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!
```

Lock Error Checking

- Always check the return value for errors 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);
}
```

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
```

Evaluating locks – Basic criteria

- **Mutual exclusion**
  - Does the lock work, preventing multiple threads from entering a **critical section**?
- **Fairness**
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free? (i.e., starvation free)
- **Performance**
  - The time overheads added by using the lock
  - Locks must provide mutual exclusion at low cost
- Building a lock needs help from the **hardware** and the **OS**

Controlling Interrupts

- **Disable Interrupts** for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for **single-processor** systems

```
1   void lock() {
2       DisableInterrupts();
3   }
4   void unlock() {
5       EnableInterrupts();
6   }
```

  - Problems
    - Requires too much *trust* in applications
      - Greedy (or malicious) program could monopolize the processor
    - Does not work on **multiprocessors**
    - Code that masks or unmasks interrupts is executed *slowly* by modern CPUs

Why is hardware support needed?

- **First attempt**: Using a *flag* denoting whether the lock is held or not
  - The lock function below has problems

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 → lock is available, 1 → held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10                ;  // spin-wait (do nothing)
11      mutex->flag = 1;  // now SET it !
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

Why hardware support needed? (Cont.)

- **Problem 1**: No Mutual Exclusion (assume `flag=0` to begin)

Thread1	Thread2
call `lock()`	
`while (mutex->flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (mutex->flag == 1)`
	`mutex->flag = 1;`
	interrupt: switch to Thread 1
`flag = 1; // set flag to 1 (too!)`	

- **Problem 2**: Spin-waiting wastes time waiting for another thread
- So, we need an atomic instruction supported by **hardware**
  - *test-and-set* instruction, also known as *atomic exchange*

Test And Set (Atomic Exchange)

- An instruction to support the creation of simple locks

```
1    int TestAndSet(int *ptr, int new) {
2        int old = *ptr;  // fetch old value at ptr location in memory
3        *ptr = new;      // store 'new' into ptr location in memory
4        return old;      // return the old value
5    }
```

  - **return** (test) old value pointed to by the `ptr`
  - *Simultaneously* **update** (set) said value to `new`
  - This sequence of operations is **performed atomically**

A Simple Spin Lock using test-and-set

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0 indicates that lock is available
7       // 1 that it is held
8       lock->flag = 0;
9   }
10
11  void lock(lock_t *lock) {
12      while (TestAndSet(&lock->flag, 1) == 1)
13              ;       // spin-wait
14  }
15
16  void unlock(lock_t *lock) {
17      lock->flag = 0;
18  }
```

- **Note**: To work correctly on *a single processor*, it requires a preemptive scheduler
  - Why?

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to entry the critical section
- **Fairness**: no
  - Spin locks don't provide any fairness guarantees
  - Indeed, a thread spinning may spin *forever*
- **Performance**:
  - For a single CPU, performance overheads can be quite *painful*
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*

Compare-And-Swap

- Test whether the value at the address(`ptr`) is equal to `expected`
  - *If so*, **update** the memory location pointed to by `ptr` with the `new` value
  - *In either case*, **return** the actual value at that memory location

```
1    int CompareAndSwap(int *ptr, int expected, int new) {
2        int actual = *ptr;
3        if (actual == expected)
4                *ptr = new;
5        return actual;
6    }
```

Compare-and-Swap hardware atomic instruction (C-style)

```
1    void lock(lock_t *lock) {
2        while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3                ; // spin
4    }
```

Spin lock with compare-and-swap

Fetch-And-Add

- **Atomically increment** a value while returning the old value at a particular address

```
1    int FetchAndAdd(int *ptr) {
2        int old = *ptr;
3        *ptr = old + 1;
4        return old;
5    }
```

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock

- **Ticket lock** can be built with fetch-and add
  - Ensure progress for all threads → **fairness**

```
1   typedef struct __lock_t {
2       int ticket;
3       int turn;
4   } lock_t;
5
6   void lock_init(lock_t *lock) {
7       lock->ticket = 0;
8       lock->turn = 0;
9   }
10
11  void lock(lock_t *lock) {
12      int myturn = FetchAndAdd(&lock->ticket);
13      while (lock->turn != myturn)
14              ; // spin
15  }
16  void unlock(lock_t *lock) {
17      FetchAndAdd(&lock->turn);
18  }
```

So Much Spinning

- Hardware-based spin locks are **simple** and they work

- In some cases, these solutions can be quite **inefficient**
  - Any time a thread gets caught *spinning*, it **wastes an entire time slice** doing nothing but checking a value

How To Avoid *Spinning*?
We'll need OS Support!

A Simple Approach: Just Yield

- When you are going to spin, **give up the CPU** to another thread
  - OS system call moves the caller from the *running state* to the *ready state*
  - The cost of a **context switch** can be substantial and the **starvation** problem still exists

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

- **Queue** to keep track of which threads are **waiting** to enter the lock
- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`

Using Queues: Sleeping Instead of Spinning

```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q; } lock_t;
5
6   void lock_init(lock_t *m) {
7       m->flag = 0;
8       m->guard = 0;
9       queue_init(m->q);
10  }
11
12  void lock(lock_t *m) {
13      while (TestAndSet(&m->guard, 1) == 1)
14          ; // acquire guard lock by spinning
15      if (m->flag == 0) {
16          m->flag = 1; // lock is acquired
17          m->guard = 0;
18      } else {
19          queue_add(m->q, gettid());
20          m->guard = 0;
21          park();
22      }
23  }
```

```
24  void unlock(lock_t *m) {
25      while (TestAndSet(&m->guard, 1) == 1)
26          ; // acquire guard lock by spinning
27      if (queue_empty(m->q))
28          // let go of lock; no one wants it
29          m->flag = 0;
30      else
31          // hold lock (for next thread!
32          unpark(queue_remove(m->q));
33      m->guard = 0;
34  }
```

- Potential race condition
  - Thread A holds lock
  - Thread B tries to get lock; fails
  - About to call park; switch B -> A
  - Thread A releases lock; switch A -> B
  - Thread B calls park; no thread will wakeup B!

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues: Sleeping Instead of Spinning

```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q; } lock_t;
5
6   void lock_init(lock_t *m) {
7       m->flag = 0;
8       m->guard = 0;
9       queue_init(m->q);
10  }
11
12  void lock(lock_t *m) {
13      while (TestAndSet(&m->guard, 1) == 1)
14          ; // acquire guard lock by spinning
15      if (m->flag == 0) {
16          m->flag = 1; // lock is acquired
17          m->guard = 0;
18      } else {
19          queue_add(m->q, gettid());
20          setpark() // declare intent to park
21          m->guard = 0;
22          park();
23      }
24  }
```

```
24  void unlock(lock_t *m) {
25      while (TestAndSet(&m->guard, 1) == 1)
26          ; // acquire guard lock by spinning
27      if (queue_empty(m->q))
28          // let go of lock; no one wants it
29          m->flag = 0;
30      else
31          // hold lock (for next thread!
32          unpark(queue_remove(m->q));
33      m->guard = 0;
34  }
```

- **Solaris** solves this problem by adding a third system call: `setpark()`
  - By calling this routine, a thread can indicate it *is about to* `park`
  - If the thread happens to be interrupted and the lock is freed before `park` is actually called, the subsequent `park` returns immediately instead of sleeping

Lock With Queues, Test-and-set, Yield, And Wakeup

Two-Phase Locks

- A two-phase lock realizes that spinning can be useful if the lock is about to be released
  - First phase
    - The lock spins for a while, hoping that it can acquire the lock
    - If the lock is not acquired during the first spin phase, a second phase is entered,
  - Second phase
    - The caller is put to sleep
    - The caller is only woken up when the lock becomes free later
- Another example of a **hybrid** approach