CS 111 week 4
Project 2a: Races and Synchronization

Discussion 1B

Tianxiang Li

Race Conditions

thread 1	*thread 2*
counter++;	counter++;

Q: The initial value of counter is 0, what would be the final value of the counter?

A: 1 or 2.

thread 1	*thread 2*
eax $\leftarrow$ counter;	eax $\leftarrow$ counter;
eax $\leftarrow$ eax + 1;	eax $\leftarrow$ eax + 1;
counter $\leftarrow$ eax;	counter $\leftarrow$ eax;

Project Overview

Part A: Concurrent adds to a shared global variable
Part B: Concurrent accesses and modifications to a shared, sorted, doubly linked list.

1)Demonstrate the Race Conditions
2)Apply mechanisms to prevent the Race Conditions
3)Measure the performance overhead of different prevention mechanisms and draw figure.

Part A: Adds to a shared variable

Write a test driver program (lab2_add)

- Options
  - **--threads**=*th_num*, default 1
  - **--iterations**=*it_num*, default 1
  - **--yield**, sets opt_yield to 1
- Variables
  - **Counter**: init zero (long long)
  - **Starting time**: *clock_gettime(3)*
- Start *th_num* of threads, each run add function to:
  - **Add 1** to counter *it_num* times
  - **Add -1** to counter *it_num* times
  - Exit to re-join parent thread
- Wait for all threads to complete
  - note the ending time for the run
  - Print to Stdout csv recording

Start with a basic add routine:

```
void add(long long *pointer, long long value) {
        long long sum = *pointer + value;
        *pointer = sum;
}
```

Pthread APIs

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

thread: returns the **thread descriptor** of the newly created thread

attr: thread attribute (which stack/CPU the thread runs), **NULL** for default

start_routine: function pointer to the **function** the new thread will execute

arg: **argument** to the start_routine

int **pthread_join**(pthread_t thread, void **retval);

***block** and **wait** for the specified thread to finish*

retval: If not NULL, stores the return value of start_routine.

Pthread APIs: example

```
long sum = 0;
void * thread_worker(void *arg) {
        unsigned long iter = *((unsigned long*) arg),  i  = 0; //iter = 2047;
        for (i = 0; i < iter; i++) sum++;
         printf(“sum is %d \n”,sum);
}

int main(void) {
        pthread_t th;
        unsigned long all= 2047;
        pthread_create(&th, NULL, thread_worker, &all); //create a new thread
        pthread_join(th, NULL); //join waits for thread to finish before main thread continues
        printf(“main: end\n”);
        return 0;
}
```

Thread API Example

```c
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");
    //create two threads
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    /* join waits for the two threads to finish,
       before finally allowing the main thread to run again*/
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    /* after two threads finish,
    main thread prints “main: end” and exit*/
    printf("main: end\n");
    return 0;
}
```

What are the possible execution orders?

Thread API Example

```c
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");
    //create two threads
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    /* join waits for the two threads to finish,
       before finally allowing the main thread to run again*/
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    /* after two threads finish,
    main thread prints “main: end” and exit*/
    printf("main: end\n");
    return 0;
}
```

What are the possible execution orders?

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”		

system creates a new thread of execution for the routine that is being called, and it runs independently of the caller

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”		

B can also print before A, etc.

Part A Overview

```
long long sum = 0;
void * thread_worker(iteration) {
        for (i = 0; i < iteration; i++)  add(&sum, 1);
        for (i = 0; i < iteration; i++)  add(&sum, -1);
}
int main(argc, argv)  {
        threads, iterations = process_arg(argc, argv);
        start = clock_gettime() // get the time when starts
        for (i = 0; i < threads; i++) thread[i] = pthread_create(.., thread_worker,.., iteration);
        for (i = 0; i < threads; i++) pthread_join(thread[i],..);
        end = clock_gettime() // get the time when ends
        total_runtime = end - start;
        print_out(name, threads, iterations, total_runtime, sum);
}
```

Part A: Demonstrate the Race Conditions

```
unsigned long long sum = 0;
void * thread_worker(iterations) {
        for (i = 0; i < iterations; i++)  add(&sum, 1);
        for (i = 0; i < iterations; i++)  add(&sum, -1);
}
int main(argc, argv)  {
        threads, iterations = process_arg(argc, argv);
        start = clock_gettime() // get the time when starts
        for (i = 0; i < threads; i++) thread[i] = pthread_create(thread_worker, iterations);
        for (i = 0; i < threads; i++) pthread_join(thread[i]);
        end = clock_gettime() // get the time when ends
        total_runtime = end - start;
        print_out(name, threads, iterations, total_runtime, sum);
}
```

threads: 2

iterations: 100

Will the final value of sum not 0?

→ Possible, but highly unlikely

- → Execution interval/thread number too small
- → The buggy interleaving is unlikely to occur

Reason: uncontrolled scheduling

- compiler code sequence to update the counter
  - **mov** 0x8049a1c, %eax — *counter is located at address 0x8049a1c, mov instruction is used to get the memory value at the address and put it into register eax*
  - **add** $0x1, %eax — *add is performed, adding 1 (0x1) to the contents of the eax register*
  - **mov** %eax, 0x8049a1c — *contents of eax are stored back into memory at the same address*

Reason: uncontrolled scheduling

			(after instruction)		
OS	**Thread 1**	**Thread 2**	**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**

Finding inconsistent sum

- Run your program for ranges of iterations (100, 1000, 10000, 100000)
  - capture the output, and note how many threads and iterations it takes to (fairly consistently) result in a failure (non-zero sum).

```
./lab2_add --iterations=10000 --threads=10
```

```
add-none,10,10000,200000,3304519,16,2828
```

Name of test, thread#, itera#, operation#, runtime, avg t/oper, total

Yield to exacerbate race condition

Race conditions only occur during certain thread interleaving

→ Hard to reproduce

→ Can we do something (e.g. add a random sleep to the code) to make it easier to reproduce?

sched_yield(): system call for the thread to give up its current time slice (give up the CPU), and let another thread run.

On Linux, default: 100ms.

```
int opt_yield; //get from command line arguments
void add(long long *pointer, long long value) {
        long long sum = *pointer + value;
        if (opt_yield)
            sched_yield();
        *pointer = sum;
    }
```

Part A: Prevent Race Condition

Implement three new versions of add function:

Using Locks

- Spin-lock
  - *__sync_lock_test_and_set*
  - *__sync_lock_release*
- pthread_mutex
  - *pthread_mutex_init*
  - *phread_mutex_lock*
  - *pthread_mutex_unlock*
- Compare and Swap
  - *__sync_val_compare_and_swap*

Race Prevention: Using lock

Lock: only enable one thread to execute the code at one time.

Example critical section

balance = balance + 1;

To use lock, we add some code around the critical section like this:

lock_t mutex; // some globally-allocated lock 'mutex' –> mutual exclusion object

...

lock(&mutex); //acquire the lock

balance = balance + 1; // *critical section*

unlock(&mutex); //free the lock

Lock variable has two states: locked or free

*A mutual exclusion object (**mutex**) is a program object that allows multiple program threads to share the same resource, but not simultaneously.*

Lock APIs: pthread_mutex

Init:

pthread_mutex_t mutex;

int **pthread_mutex_init**(pthread_mutex_t * mutex, pthread_mutexattr_t *restrict attr);

attr: NULL for default.

----

pthread_mutex_t mutex;

pthread_mutex_init(&mutex, NULL);

Lock:

pthread_mutex_lock(&mutex);

Unlock:

pthread_mutex_unlock(&mutex);

Pthread APIs: example

```
long sum = 0;
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
void * thread_worker(void *arg) {
        unsigned long iter = *((unsigned long*) arg),  i  = 0; //iter = 2047
        pthread_mutex_lock(&mutex);
        for (i = 0; i < iter; i++) sum++;
        pthread_mutex_unlock(&mutex);
}
int main(void) {
        pthread_t th;
        unsigned long all= 2047;
        pthread_create(&th, NULL, thread_worker, &all);
        pthread_join(th, NULL);
        return 0;
}
```

Spin Lock

- When you don’t get the lock
  - Keep trying again (spinning)
- Good point
  - Enforce access to critical sections
  - Simple to program
- Dangers
  - Wasteful
  - Spin locking takes up processor cycles
  - May delay freeing of required resource

Spin Lock

- While(lock==1) *//If lock =1 (locked by other threads), we keep looping until lock is released (lock=0)*
- Lock=1 *//acquire the lock*

  …
- Lock=0 *//release the lock*

Spin Lock problem

Thread1	Thread2	Thread3
Lock=0		
	While(lock==1)	
		While(lock==1)
	Lock=1	
		Lock=1

Test and Set (Atomic instruction)

- The core operation of acquiring a lock (when it's free) requires:
  1. Check that no one else has it
  2. Change something so others know we have it

```
//Pseudo code for test and set (machine language instructions)
long TS(long*lock, long value){
        long old = *lock;
        *lock = value;          //set lock to new value
        return old;             //return old value
}

If (TS(&lock,1)==0){//other threads have returned the lock, we set it to 1, and continue to execute critical section

        /* We have control of the critical section!*/
}
```

Lock APIs: spinlock

init:

```
long lock = 0;
```

Lock:

```
while (__sync_lock_test_and_set (&lock, 1));
//If lock =1 (locked by other threads), we keep looping until lock is released (lock=0)
//if lock=0, we acquire the lock(lock=1) and jump out of the loop
```

Unlock:

```
__sync_lock_release(&lock);
```

Example code

```
long sum = 0;
void * thread_worker(void *arg) {
        unsigned long iter = *((unsigned long*) arg),  i  = 0; //iter = 2047
        for (i = 0; i < iter; i++) sum++;
}

int main(void) {
        pthread_t th;
        unsigned long all= 2047;
        pthread_create(&th, NULL, thread_worker, &all);
        pthread_join(th, NULL);
        return 0;
}
```

Where do we add spinlock code?

spinlock: example

```
long sum = 0;
long lock = 0;
void * thread_worker(void *arg) {
        unsigned long iter = *((unsigned long*) arg),  i  = 0; //iter = 2047
        while (__sync_lock_test_and_set (&lock, 1));
        for (i = 0; i < iter; i++) sum++;
        __sync_lock_release(&lock);
}

int main(void) {
        pthread_t th;
        unsigned long all= 2047;
        pthread_create(&th, NULL, thread_worker, &all);
        pthread_join(th, NULL);
        return 0;
}
```

spinlock vs pthread_mutex_lock

What to do if failed to acquire the lock:

1. Keep spinning: spinlock

   Drawback: waste CPU cycles → ideal for low contention, short critical sections

2. Yield: the thread is put into asleep

   Drawback: large overhead to sleep/wake up threads →

   ideal for high contention, large critical sections.

Ideally: Transparently select appropriate operation when failed to acquire the lock for different critical sections

pthread_mutex_lock: middle ground:

spins on the lock for a threshold (e.g. 1000 times), if not acquired the lock, put into asleep.

Compare and Swap for atomic adds

```
void add(long long *pointer, long long value) {
        long long sum = *pointer + value;
        *pointer = sum;
    }
```

Insight: essentially just needs an *atomic add operation* →

atomic_add(pointer, value);

Goal: Implement **atomic_add** with **compare and swap**

Compare and Swap (Atomic Instruction)

Again, a C description of machine instruction

```
bool compare_and_swap( int *p, int old, int new ) {
  if (*p == old) {    /* see if value has been changed      */
      *p = new;       /* if not, set it to new value        */
      return( TRUE);  /* tell caller he succeeded           */
  } else              /* someone else changed *p            */
    return( FALSE);   /* tell caller he failed              */
}

if (compare_and_swap(flag,UNUSED,IN_USE) {
      /* I got the critical section! */
} else {
      /* I didn't get it.  */
}
```

Compare and swap implementation for atomic add

NOTE: This is just pseudo code for compare and swap, implementation is done by hardware

```
bool sync_bool_compare_and_swap(long long *p, long long old, long long new )  {
        if (*p == old) { /* see if value has been changed */
                *p = new; /* if not, set it to new value */
                 return true; /* tell caller he succeeded */
        }
        else /* someone else changed *p */
                return false; /* tell caller he failed */
}

void atomic_add(void * pointer, unsigned long value) {
        long long prev, sum;
        do {
                prev = *pointer;
                sum = prev + value;
        } while(__sync_bool_compare_and_swap (pointer, prev, sum) == false);
}
```

Part A: Measure Performance

Part A Overview

```
long long sum = 0;
void * thread_worker(iteration) {
        for (i = 0; i < iteration; i++)  add(&sum, 1);
        for (i = 0; i < iteration; i++)  add(&sum, -1);
}
int main(argc, argv)  {
        threads, iterations = process_arg(argc, argv);
        start = clock_gettime() // get the time when starts
        for (i = 0; i < threads; i++) thread[i] = pthread_create(thread_worker, iteration);
        for (i = 0; i < threads; i++) pthread_join(thread[i]);
        end = clock_gettime() // get the time when ends
        total_runtime = end - start;
        print_out(name, threads, iterations, total_runtime, sum);
}
```

clock_gettime API

```
struct timespec {
        time_t   tv_sec; // seconds: number of whole seconds elapsed since some staring point
        long     tv_nsec; //nanoseconds: number of nanoseconds elapsed since tv_sec value
    };
int clock_gettime(clockid_t clock_id, struct timespec *tp);
```

clockid argument is the identifier of the particular clock on which to act

clock_id can be:

- CLOCK_REALTIME (Real World time, adjust based on time server)
- **CLOCK_MONOTONIC (Elapsed time since some unspecified starting point, increasing)**
- CLOCK_PROCESS_CPUTIME_ID (CPUTIME of the current processes)
- CLOCK_THREAD_CPUTIME_ID (CPUTIME of the current threads)

clock_gettime API examples

```
static inline unsigned long get_nanosec_from_timespec(struct timespec * spec)
{
        unsigned long ret= spec->tv_sec; //seconds
        ret = ret * 1000000000 + spec->tv_nsec; //nanoseconds
        return ret;
}

int main(void) {
        struct timespec begin, end;
        unsigned long diff = 0;
        clock_gettime(CLOCK_MONOTONIC, &begin);
        do_something;
        clock_gettime(CLOCK_MONOTONIC, &end);

        //diff stores the execution time in ns
        diff =  get_nanosec_from_timespec(&end) - get_nanosec_from_timespec(&begin);
}
```

Generate Figures

Within Makefile (note the below makefile code does not consider the proper decencies for targets)

```
tests:
	rm -rf lab2_add.csv
	./lab2_add --threads=2  --iterations=100    >> lab2_add.csv
	./lab2_add --threads=2  --iterations=1000   >> lab2_add.csv
	./lab2_add --threads=2  --iterations=10000  >> lab2_add.csv
	...
	./lab2_add --threads=12 --iterations=10000 --sync=s >> lab2_add.csv

graphs:
	gnuplot ./lab2_add.gp
```

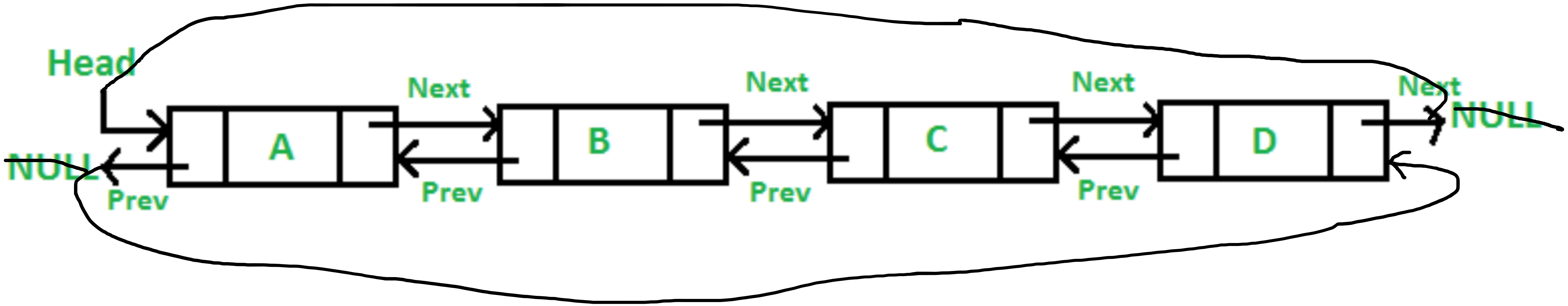
Part B

Doubly linked list

```
struct SortedListElement {
    struct SortedListElement *prev;
    struct SortedListElement *next;
    const char *key;
};
typedef struct SortedListElement SortedList_t;
typedef struct SortedListElement SortedListElement_t;
```

We will be using a circular doubly linked list in project2A

The next pointer in the head points at the first (lowest valued) element in the list. The prev pointer in the list head points at the last (highest valued) element in the list. Thus, the list is circular.



Sortedlist.h functions

- **sortedList_insert**(list, element);
  - insert an element into a sorted list
- **sortedList_delete**(element);
  - remove an element from a sorted list
- **sortedList_lookup**(list, key);
  - search sorted list for a key
  - specified list will be searched for an element with the specified key
- **sortedList_length**(list)
  - count elements in a sorted list while enumerating list
  - checks all prev/next pointers

```
SortedListElement_t *listhead, * pool;
void * thread_worker(threadNum) {
        startIndex = threadNum * iteration;
        for (i = startIndex; i < startIndex + iteration; i++) SortedList_insert(listhead, pool[i]); //insert element
        SortedList_length(listhead); //check length
        for (i = startIndex; i < startIndex + iteration; i++) {
                e = SortedList_lookup(listhead, pool[i]->key); //lookup inserted element
                SortedList_delete(listhead, e); //remove inserted element
        }
}
int main(argc, argv)  {
        threads, iterations = process_arg(argc, argv);
        pool = malloc(threads * iterations * sizeof(SortedListElement_t));
        for (i = 0; i < threads * iterations; i++) init_elem(&pool[i]);
        start = clock_gettime() // get the time when starts
        for (i = 0; i < threads; i++) thread[i] = pthread_create(thread_worker, i);
        for (i = 0; i < threads; i++) pthread_join(thread[i]);
        end = clock_gettime() // get the time when ends
        total_runtime = end - start;
        print_out(name, threads, iterations, total_runtime, sum);
}
```

Thread #

Iteration #	0	1	2
1	ele01	ele11	ele21
2	ele02	ele12	ele22
3	ele03	ele12	ele23

pool of elements

Part B: Demonstrate the Race Conditions

Note: The code in this slide cannot be directly applied to project.

```
void list_remove(node *element) {
        node*n = element->next; //say n = 0x1234
```

Interrupted by another process trying to remove n

```
                                        list_remove(0x1234);
```

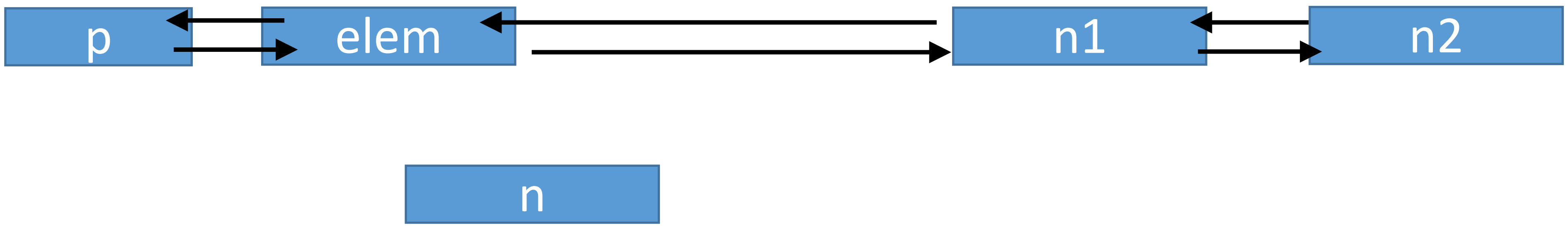
```
        node*p = element->prev;
        n->prev = p;
        p->next = n;
        element->next = NULL;
        element->prev = NULL;
}
```

Note: The code in this slide cannot be directly applied to project.

```
void list_remove(node *element) {
        node*n = element->next; //say n = 0x1234
        node*p = element->prev;
        n->prev = p;
        p->next = n;
        element->next = NULL;
        element->prev = NULL;
}
```

Interrupted by another process trying to remove n

list_remove(0x1234);



Note: The code in this slide cannot be directly applied to project.

```
void list_remove(node *element) {
        node*n = element->next; //say n = 0x1234
                                        Interrupted by another process trying to remove n
                                        list_remove(0x1234);
        node*p = element->prev;
        n->prev = p;
        p->next = n;
        element->next = NULL;
        element->prev = NULL;
}
```

PART B: sorted, doubly-linked, list

- implement all four methods in sortedList.c.
  - Identify the critical section in each of four methods
  - Add calls to pthread_yield or sched_yield
    - in SortedList_insert if opt_yield & INSERT_YIELD
    - in SortedList_delete if opt_yield & DELETE_YIELD
    - in SortedList_lookup if opt_yield & LOOKUP_YIELD
    - in SortedList_length if opt_yield & LOOKUP_YIELD

```
extern int opt_yield;
#define INSERT_YIELD    0x01    // yield in insert critical section
#define DELETE_YIELD    0x02    // yield in delete critical section
#define LOOKUP_YIELD    0x04    // yield in lookup/length critical esction
```

End of Discussion

- Q&A