

Concurrency Discussion Review

CS 537: 03/12/25

Outline

Locks

Condition Variables

Examples and Group Work

- Reader/Writer Lock using Condition Variables

- Designing Thread Safe Stack

- Identifying Deadlock Practice

Outline

Basics

Locks

Condition Variables

Application

Examples and Group Work

Reader/Writer Lock using Condition Variables

Designing Thread Safe Stack

Identifying Deadlock Practice

Quick Note: Threads Vs. Processes

What is the difference?

Why use one over the other?

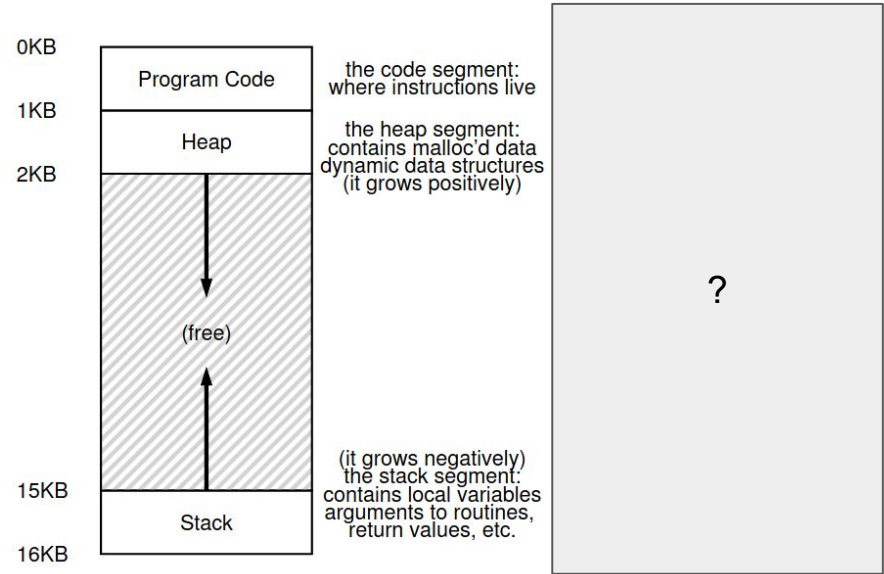


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

Quick Note: Threads Vs. Processes

What is the difference?

Why use one over the other?

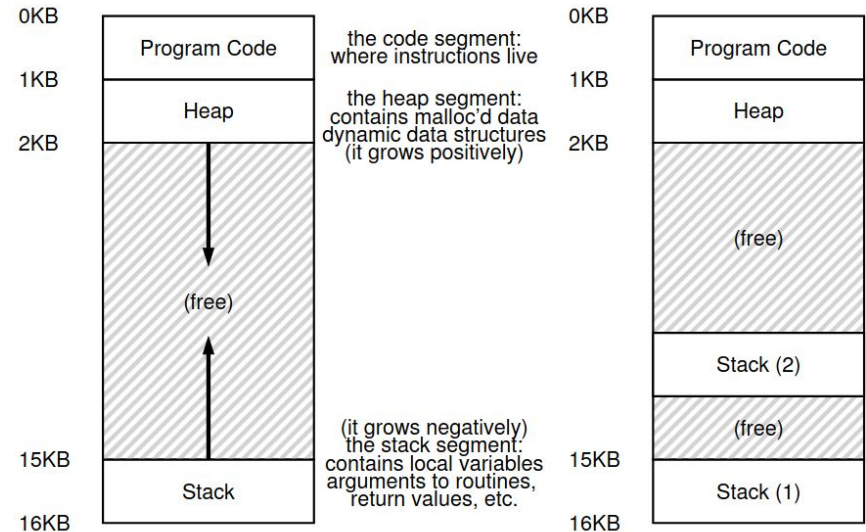


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

Creating Threads

POSIX Threads (pthreads)

Declare pthread vars

Join (like wait)

Function Thread Executes

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4  #include "common.h"
5  #include "common_threads.h"
6
7  void *mythread(void *arg) {
8      printf("%s\n", (char *) arg);
9      return NULL;
10 }
11
12 int
13 main(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     int rc;
16     printf("main: begin\n");
17     Pthread_create(&p1, NULL, mythread, "A");
18     Pthread_create(&p2, NULL, mythread, "B");
19     // join waits for the threads to finish
20     Pthread_join(p1, NULL);
21     Pthread_join(p2, NULL);
22     printf("main: end\n");
23     return 0;
24 }
```

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

Creating Threads

POSIX Threads (pthreads)

Declare pthread vars

Join (like wait)

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4  #include "common.h"
5  #include "common_threads.h"
6
7  void *mythread(void *arg) {
8      printf("%s\n", (char *) arg);
9      return NULL;
10 }
11
12 int
13 main(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     int rc;
16     printf("main: begin\n");
17     Pthread_create(&p1, NULL, mythread, "A");
18     Pthread_create(&p2, NULL, mythread, "B");
19     // join waits for the threads to finish
20     Pthread_join(p1, NULL);
21     Pthread_join(p2, NULL);
22     printf("main: end\n");
23     return 0;
24 }
```

Function Thread Executes

Print order not guaranteed!

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

Concurrency Primitives

Locks

Why use locks?

```
5
6 #include <pthread.h>
7 #include <stdio.h>
8
9 int sum = 0;
10
11 struct arg_struct{
12     int val;
13     int n;
14 };
15
16 void * thread_func(void *args)
17 {
18     struct arg_struct *arguments = (struct arg_struct *) args;
19
20     for (int i = 0; i < arguments->n; i++)
21     {
22         sum += arguments->val;
23     }
24
25     return NULL;
26 }
27
28 int main(int argc, char ** argv)
29 {
30     pthread_t p1, p2;
31
32     struct arg_struct p1_arg = {10, 10000};
33     struct arg_struct p2_arg = {1, 10000};
34
35     pthread_create(&p1, NULL, thread_func, &p1_arg);
36     pthread_create(&p2, NULL, thread_func, &p2_arg);
37
38     pthread_join(p1, NULL);
39     pthread_join(p2, NULL);
40
41     int exp = p1_arg.n * p1_arg.val \
42             + p2_arg.n * p2_arg.val;
43
44     printf("Sum: %d, Expected: %d\n", sum, exp);
45     printf("Total %: %f\n", 100 * sum * 1.0 / exp);
46
47     return 0;
48 }
```

Locks

Why use locks?

```
6 #include <pthread.h>
7 #include <stdio.h>
8
9 int sum = 0;
10
11 struct arg_struct{
12     int val;
13     int n;
14 };
15
16 void * thread_func(void *args)
17 {
18     struct arg_struct *arguments = (struct arg_struct *) args;
19
20     for (int i = 0; i < arguments->n; i++)
21     {
22         sum += arguments->val;
23     }
24
25     return NULL;
26 }
27
28 int main(int argc, char ** argv)
29 {
30     pthread_t p1, p2;
31
32     struct arg_struct p1_arg = {10, 10000};
33     struct arg_struct p2_arg = {1, 100000};
34
35     pthread_create(&p1, NULL, thread_func, &p1_arg);
36     pthread_create(&p2, NULL, thread_func, &p2_arg);
37
38     pthread_join(p1, NULL);
39     pthread_join(p2, NULL);
40
41     int exp = p1_arg.n * p1_arg.val \
42             + p2_arg.n * p2_arg.val;
43
44     printf("Sum: %d, Expected: %d\n", sum, exp);
45     printf("Total %: %f\n", 100 * sum * 1.0 / exp);
46
47     return 0;
48 }
```

Three different runs!

Sum: 91290, Expected: 110000
Total %: 82.990909



Sum: 10000, Expected: 110000
Total %: 9.090909



Sum: 110000, Expected: 110000
Total %: 100.000000

Locks

```
16 pthread_mutex_t lock;  
17 pthread_mutex_t lock;  
18 void * thread_func(void *args)  
19 {  
20     struct arg_struct *arguments = (struct arg_struct *) args;  
21  
22     for (int i = 0; i < arguments->n; i++)  
23     {  
24         pthread_mutex_lock(&lock);  
25         sum += arguments->val;  
26         pthread_mutex_unlock(&lock);  
27     }  
28  
29     return NULL;  
30 }
```

Sum: 110000, Expected: 110000
Total %: 100.000000

Ex. Types of Lock Mechanism (Pros/Cons of each?)

Spin Lock

```
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

Spin Lock with Yield

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

Figure 28.8: Lock With Test-and-set And Yield

Blocking Lock

```
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

Figure 28.9: Lock With Queues, Test-and-set, Yield, And Wakeup

Condition Variables

Allows threads to wait on a specific condition.

```
1  cond_t  empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == 1)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.12: Producer/Consumer: Two CVs And While

Condition Variables

Allows threads to wait on a specific condition.

Why two cond_t?

Why use while() and not if()?

```
1  cond_t  empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == 1)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.12: Producer/Consumer: Two CVs And While

Semaphores

Wait: Check value, if greater than 0
(Linux) decrement and proceed.

Post: Increment value.

Can be used for both lock/condition
variable.

Can be used for thread throttling.

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

Figure 31.17: Implementing Zemaphores With Locks And CVs

“Zemaphore” is a semaphore! Read chapter 31 for more.

Examples

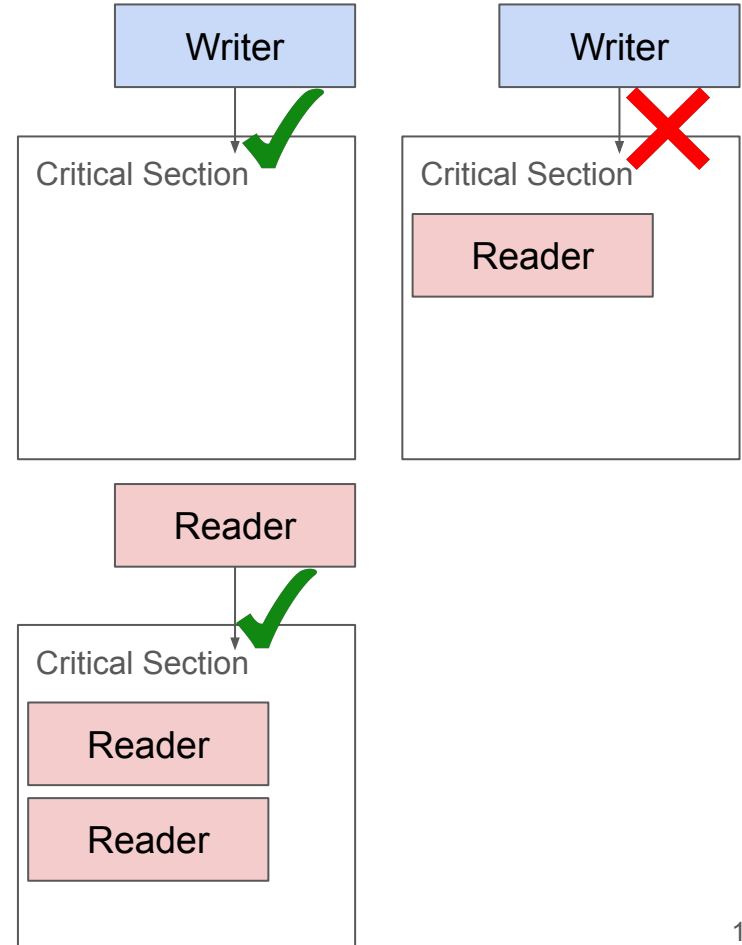
Reader/Writer Lock

Mutual exclusion is inefficient with many readers, few writers.

Readers could execute in parallel.

ReaderWriter (RW) Lock

- If no one holds lock, writer may enter.
- If no one other than a reader holds lock, a reader may enter.



Reader/Writer Lock

```
int NReaders, Nwriters;  
Cond_t CanRead, CanWrite;  
Mutex_t lock;
```

```
Void BeginWrite() {  
    pthread_mutex_lock(&lock)  
    while(NWriters == 1 || NReaders > 0) {  
        cond_wait(CanWrite,&lock);  
    }  
    NWriters = 1;  
    pthread_mutex_unlock(&lock)  
}
```

```
Void EndWrite() {  
    pthread_mutex_lock(&lock)  
    NWriters = 0;  
    Signal(CanRead);  
    Signal(CanWrite);  
    pthread_mutex_unlock(&lock)  
}
```

```
Void BeginRead() {  
    pthread_mutex_lock(&lock)  
    while(NWriters == 1) {  
        cond_wait(CanRead,&lock);  
    }  
    ++NReaders;  
    Signal(CanRead);  
    pthread_mutex_unlock(&lock)  
}
```

```
Void EndRead() {  
    pthread_mutex_lock(&lock)  
    if(--NReaders == 0)  
        Signal(CanWrite);  
    pthread_mutex_unlock(&lock)  
}
```

Reader/Writer Lock

```
int NReaders, Nwriters;  
Cond_t CanRead, CanWrite;  
Mutex_t lock;
```

What happens with
multiple readers and
writers waiting?

```
Void BeginWrite() {  
    pthread_mutex_lock(&lock)  
    while(NWriters == 1 || NReaders > 0) {  
        cond_wait(CanWrite,&lock);  
    }  
    NWriters = 1;  
    pthread_mutex_unlock(&lock)  
}
```

```
Void EndWrite() {  
    pthread_mutex_lock(&lock)  
    NWriters = 0;  
    Signal(CanRead);  
    Signal(CanWrite);  
    pthread_mutex_unlock(&lock)  
}
```

```
Void BeginRead() {  
    pthread_mutex_lock(&lock)  
    while(NWriters == 1) {  
        cond_wait(CanRead,&lock);  
    }  
    ++NReaders;  
    Signal(CanRead);  
    pthread_mutex_unlock(&lock)  
}
```

```
Void EndRead() {  
    pthread_mutex_lock(&lock)  
    if(--NReaders == 0)  
        Signal(CanWrite);  
    pthread_mutex_unlock(&lock)  
}
```

Reader/Writer Lock : Fair Edition

```
int WaitingWriters,  
    WaitingReaders,  
    NReaders, N Writers;  
Cond_t CanRead, CanWrite;  
Mutex_t lock;
```

```
Void BeginWrite() {  
    pthread_mutex_lock(&lock)  
    while(NWriters == 1 || NReaders > 0) {  
        ++WaitingWriters;  
        cond_wait(CanWrite, &lock);  
        --WaitingWriters;  
    }  
    NWriters = 1;  
    pthread_mutex_unlock(&lock)  
}
```

```
Void EndWrite() {  
    pthread_mutex_lock(&lock)  
    NWriters = 0;  
    if(WaitingReaders)  
        Signal(CanRead);  
    else Signal(CanWrite);  
    pthread_mutex_unlock(&lock)  
}
```

```
Void BeginRead() {  
    bool didWait = False;  
    pthread_mutex_lock(&lock)  
    while(NWriters == 1 ||  
        (WaitingWriters > 0 && \  
         NReaders > 0 && !didWait)) {  
        ++WaitingReaders;  
        cond_wait(CanRead, &lock);  
        --WaitingReaders;  
        didWait = True;  
    }  
    ++NReaders;  
    Signal(CanRead);  
    pthread_mutex_unlock(&lock)  
}
```

```
Void EndRead() {  
    pthread_mutex_lock(&lock)  
    if(--NReaders == 0)  
        Signal(CanWrite);  
    pthread_mutex_unlock(&lock)  
}
```

Student Group Activity: Thread Safe Stack Design

Write the code for a thread-safe stack, where the stack implements a PUSH and POP operation of any arbitrary value (void*).

PUSH should wait while the stack is full and POP should wait while the stack is empty.

Use mutexes and condition variables only. You can assume the stack has a known maximum size

```
void * POP() {  
    ...  
}
```

```
void * PUSH(void *element) {  
    ...  
}
```

Example Stack

```
typedef struct {  
    void* data[STACK_SIZE];  
    int top;  
    pthread_mutex_t mutex;  
    pthread_cond_t cond_not_full;  
    pthread_cond_t cond_not_empty;  
} ThreadSafeStack;
```

```
void stack_push(ThreadSafeStack* stack, void* value) {  
    pthread_mutex_lock(&stack->mutex);  
    while (stack->top == STACK_SIZE - 1) {  
        pthread_cond_wait(&stack->cond_not_full, &stack->mutex);  
    }  
    stack->data[++stack->top] = value;  
    pthread_cond_signal(&stack->cond_not_empty);  
    pthread_mutex_unlock(&stack->mutex);  
}
```

```
void* stack_pop(ThreadSafeStack* stack) {  
    pthread_mutex_lock(&stack->mutex);  
    while (stack->top == -1) {  
        pthread_cond_wait(&stack->cond_not_empty, &stack->mutex);  
    }  
    void* value = stack->data[stack->top--];  
    pthread_cond_signal(&stack->cond_not_full);  
    pthread_mutex_unlock(&stack->mutex);  
    return value;  
}
```

Untested ChatGPT generated code! Exercise: Look over and see if its valid.

Example Identifying Deadlocks

What are the four requirements for a deadlock?

Example Identifying Deadlocks

What are the four requirements for a deadlock?

- **Mutual Exclusion:** Thread gains exclusive control of resource.
- **Hold-and-Wait:** Threads hold resources already allocated while waiting for other resources.
- **No Preemption:** Resources cannot be forcibly removed from threads holding them.
- **Circular Wait:** Circular dependency chain, each thread holds resources requested by next in chain.

Example Identifying Deadlocks

```
void transfer(Account *acct1, Account *acct2, int amount) {  
    lock(acct1->lock);  
    if (acct1->balance > amount) {  
        lock(acct2->lock);  
        acct1->balance = acct1->balance - amount;  
        acct2->balance = acct2->balance + amount;  
        release(acct2->lock);  
    }  
    release(acct1->lock);  
}
```

```
void withdraw(Account* acct) {  
    lock(acct->lock);  
    if (acct->balance > amount) {  
        acct->balance =  
            acct->balance - amount;  
    }  
    release(acct->lock);  
}
```

This Slide Intentionally Left Blank