Discussion 3

Mutual Exclusion, Condition Variables

09/25/24

Staff

CS162

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Homework 1 Due	Homework 2 Release			Project 1 Design Doc Due		
				Midterm 1 (7-9 PM)		

Mutual Exclusion

Mutual Exclusion

Oftentimes, a multithreaded program will involve threads having shared states, meaning threads will share data with one another. Such threads are called **cooperating threads**. If threads don't share any states, they are called **independent threads.**

Cooperation between threads (i.e. **synchronization**) is made possible via **atomic operations,** which always run to completion or not at all.

- Atomic operations are indivisible.
- Typically loads and stores are atomic.

Mutual exclusion is a basic method of **synchronization** where only one thread does a particular action at a time.

- Code that runs as part of mutual exclusion is the **critical section**.
- Critical sections provide the illusion that each thread operates atomically on a shared state.

Locks

Locks are synchronization variables that provide mutual exclusion.

- One thread holding lock → no other thread can hold it.

Two atomic operations.

- **Acquire/Lock**: waits until the lock is free, then marks it as busy. When returning from this call, the thread is said to be holding the lock.
- **Release/Unlock**: marks lock as free. When returning from this call, the thread no longer holds the lock. This can only be called if the thread is holding the lock.

Code in between acquire and release is the critical section.

```
lock_acquire(&lock);

Critical section;

lock_release(&lock);
```

Semaphores

Semaphores are synchronization variables with a nonnegative integer.

Two atomic operations.

- **Down/Wait/P**: waits for semaphore's value to become strictly positive, then decrements it by 1.
- **Up/Post/V**: increments the value of the semaphore by 1.

Technically not allowed to examine the value of a semaphore.

Mutual exclusion workflow: use semaphore as a lock.

1. Initialize semaphore to 1.
2. Down semaphore when entering a critical section.
3. Up semaphore when exiting critical section.

Scheduling workflow: one thread waits for another thread to do something.

1. Initialize semaphore to 0.
2. Down semaphore in the waiting thread.
3. After finishing desired work, up semaphore in the working thread.

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Break down a money transfer into further lines (not necessarily atomic instructions).

`donor->balance -= amount`	`temp = donor->balance;` `temp = temp - amount;` `donor->balance = temp;`
`recipient->balance += amount`	`temp = recipient->balance;` `temp = temp + amount;` `recipient->balance = temp;`

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

alice->balance	5
bob->balance	5
temp1	
temp2	

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)
temp1 = alice->balance;	

alice->balance	5
bob->balance	5
temp1	5
temp2	

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended
    behavior. What changes could you make to the CGFC to defend
    against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

temp1 = alice->balance;
temp1 = temp1 - 5;
alice->balance = temp1;	

alice->balance	0
bob->balance	5
temp1	0
temp2	

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)
`temp1 = alice->balance;` **`temp1 = temp1 - 5;`**	

alice->balance	5
bob->balance	5
temp1	0
temp2	

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

temp1 = alice->balance;
temp1 = temp1 - 5;
alice->balance = temp1;
temp1 = bob->balance;	

alice->balance	0
bob->balance	5
temp1	5
temp2	

Synchronization Stuffs

```c
typedef struct account {
/* ... */
 int balance;
/* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

temp1 = alice->balance;
temp1 = temp1 - 5;
alice->balance = temp1;
temp1 = bob->balance;
temp1 = temp1 + 5;	

alice->balance	0
bob->balance	5
temp1	10
temp2	

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

`temp1 = alice->balance;`
`temp1 = temp1 - 5;`
`alice->balance = temp1;`
`temp1 = bob->balance;`
`temp1 = temp1 + 5;`	

	`temp2 = bob->balance;`

`alice->balance`	0
`bob->balance`	5
`temp1`	10
`temp2`	5

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

temp1 = alice->balance;
temp1 = temp1 - 5;
alice->balance = temp1;
temp1 = bob->balance;
temp1 = temp1 + 5;	

	temp2 = bob->balance; **temp2 = temp2 - 5;**

alice->balance	0
bob->balance	5
temp1	10
temp2	0

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

`temp1 = alice->balance;`
`temp1 = temp1 - 5;`
`alice->balance = temp1;`
`temp1 = bob->balance;`
`temp1 = temp1 + 5;`	

	`temp2 = bob->balance;`
	`temp2 = temp2 - 5;`
	`bob->balance = temp2;`

alice->balance	0
bob->balance	0
temp1	10
temp2	0

Synchronization Stuffs

```c
typedef struct account {
  /* ... */
  int balance;
  /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

temp1 = alice->balance;
temp1 = temp1 - 5;
alice->balance = temp1;
temp1 = bob->balance;
temp1 = temp1 + 5;	

	temp2 = bob->balance;
	temp2 = temp2 - 5;
	bob->balance = temp2;
	temp2 = alice->balance;

alice->balance	0
bob->balance	0
temp1	10
temp2	0

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

`temp1 = alice->balance;`
`temp1 = temp1 - 5;`
`alice->balance = temp1;`
`temp1 = bob->balance;`
`temp1 = temp1 + 5;`	

	`temp2 = bob->balance;`
	`temp2 = temp2 - 5;`
	`bob->balance = temp2;`
	`temp2 = alice->balance;`
	`temp2 = temp2 + 5;`

alice->balance	0
bob->balance	0
temp1	10
temp2	5

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

`temp1 = alice->balance;`
`temp1 = temp1 - 5;`
`alice->balance = temp1;`
`temp1 = bob->balance;`
`temp1 = temp1 + 5;`	

	`temp2 = bob->balance;`
	`temp2 = temp2 - 5;`
	`bob->balance = temp2;`
	`temp2 = alice->balance;`
	`temp2 = temp2 + 5;`
	`alice->balance = temp2;`

`alice->balance`	5
`bob->balance`	0
`temp1`	10
`temp2`	5

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Consider two operations: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)` and a particular interleaving which leads to Bob having 10 and Alice having 5.

transfer(alice, bob, 5)	transfer(bob, alice, 5)

`temp1 = alice->balance;`
`temp1 = temp1 - 5;`
`alice->balance = temp1;`
`temp1 = bob->balance;`
`temp1 = temp1 + 5;`	

	`temp2 = bob->balance;`
	`temp2 = temp2 - 5;`
	`bob->balance = temp2;`
	`temp2 = alice->balance;`
	`temp2 = temp2 + 5;`
	`alice->balance = temp2;`

`bob->balance = temp1;`	

alice->balance	5
bob->balance	10
temp1	10
temp2	5

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against the exploits?

Another possible exploit if a thread switch occurs after the if statement succeeds, which results in donor->balance changing to less than amount.

Synchronization Stuffs

```c
typedef struct account {
 /* ... */
 int balance;
 /* ... */
} account_t;

void transfer(account_t* donor, account_t* recipient, float amount) {
  assert(donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

1. Describe how a malicious user might exploit some unintended
    behavior. What changes could you make to the CGFC to defend
    against the exploits?

Fix exploits by making the entire method a critical section (i.e. surround the code with locks).

Synchronization Stuffs

```
void play_session(struct server* s) {
  connect(s);
  play();
  disconnect(s);
}
```

2. A recent popular game is having issues with its servers lagging heavily due to too many players being connected at a time.

   After testing, it turns out that the servers can run without lagging for a max of up to 1000 players concurrently connected. How can you use synchronization to enforce a strict limit of 1000 players connected at a time? Assume that a game server can create synchronization variables and share them amongst the player threads.

Synchronization Stuffs

Player threads run the following code:

```
void play_session(struct server* s) {
  connect(s);
  play();
  disconnect(s);
}
```

2. A recent popular game is having issues with its servers lagging heavily due to too many players being connected at a time.

    After testing, it turns out that the servers can run without lagging for a max of up to 1000 players concurrently connected. How can you use synchronization to enforce a strict limit of 1000 players connected at a time? Assume that a game server can create synchronization variables and share them amongst the player threads.

Use a semaphore for each server that's initialized to 1000. Down semaphore before calling connect, up semaphore after calling disconnect.

The order here is important - downing the semaphore after connecting but before playing means that there is nothing blocking threads from calling connect().

Upping the semaphore before disconnecting could lead to new connections while there are still players connected to the game. For example, consider the case where 1000 players are connected, one player ups the semaphore (but hasn't disconnected), and the 1001th player connects (due to the upped semaphore) while the previous player hasn't yet disconnected. Now we have 1001 players connected.

Atomic operations

Revisiting the idea of atomic operations:

- In a world where only loads and stores are atomic, it's painful to enforce synchronization
- Hardware support for instructions that read and write a value atomically

Pseudocode for one such instruction, *test_and_set*, that can help us implement locks:

```
int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}
```

Assume that this function only meant to describe the behavior of a single atomic operation.

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.
2. Thread2 is scheduled and run until after it increments hello and is switched.
3. Thread1 runs until it is switched.
4. Main thread resumes and runs until it is switched.
5. Thread2 runs to completion and exits.
6. Main thread runs to completion but doesn't exit.
7. Thread1 runs to completion.

1. Why is this sequence of events possible?

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.

2. Thread2 is scheduled and run until after it increments hello and is switched.

3. Thread1 runs until it is switched.

4. Main thread resumes and runs until it is switched.

5. Thread2 runs to completion and exits.

6. Main thread runs to completion but doesn't exit.

7. Thread1 runs to completion.

1. Why is this sequence of events possible?

In steps 3 and 4, main thread and thread1 make no progress since they can only advance when value is 0.

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.
2. Thread2 is scheduled and run until after it increments hello and is switched.
3. Thread1 runs until it is switched.
4. Main thread resumes and runs until it is switched.
5. Thread2 runs to completion and exits.
6. Main thread runs to completion but doesn't exit.
7. Thread1 runs to completion.

2. For steps where `test_and_set` is called, what does it return?

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.
2. Thread2 is scheduled and run until after it increments hello and is switched.
3. Thread1 runs until it is switched.
4. Main thread resumes and runs until it is switched.
5. Thread2 runs to completion and exits.
6. Main thread runs to completion but doesn't exit.
7. Thread1 runs to completion.

2. For steps where `test_and_set` is called, what does it return?

Step 2: `value` is initially 0, so 0 is returned by `test_and_set` and the while loop condition will not be satisfied. Now `value` is 1.

Steps 3, 4: thread1 and main thread do not advance past the while loop since `test_and_set` keeps returning 1.

Step 6: main thread is able to advance (since `value` got set to 0, meaning `test_and_set` returns 0) and runs to completion.

Step 7: thread1 is able to advance and runs to completion (same reasoning).

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.
2. Thread2 is scheduled and run until after it increments hello and is switched.
3. Thread1 runs until it is switched.
4. Main thread resumes and runs until it is switched.
5. Thread2 runs to completion and exits.
6. Main thread runs to completion but doesn't exit.
7. Thread1 runs to completion.

3. What is the output of this program?

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.
2. Thread2 is scheduled and run until after it increments hello and is switched.
3. Thread1 runs until it is switched.
4. Main thread resumes and runs until it is switched.
5. Thread2 runs to completion and exits.
6. Main thread runs to completion but doesn't exit.
7. Thread1 runs to completion.

3. What is the output of this program?

```
Child thread: 1
Parent thread: 1
Child thread: 2
```

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.
2. Thread2 is scheduled and run until after it increments hello and is switched.
3. Thread1 runs until it is switched.
4. Main thread resumes and runs until it is switched.
5. Thread2 runs to completion and exits.
6. Main thread runs to completion but doesn't exit.
7. Thread1 runs to completion.

4. What is a major issue with this implementation of synchronization?

Test and Set

```c
int value = 0;
int hello = 0;

int test_and_set(int* value) {
  int old_value = *value;
  *value = 1;
  return old_value;
}

void print_hello() {
  while (test_and_set(&value));
  hello += 1;
  printf("Child thread: %d\n", hello);
  value = 0;
  pthread_exit(0);
}
void main() {
  pthread_t thread1;
  pthread_t thread2;
  pthread_create(&thread1, NULL, &print_hello, NULL);
  pthread_create(&thread2, NULL, &print_hello, NULL);
  while (test_and_set(&value));
  printf("Parent thread: %d\n", hello);
  value = 0;
}
```

Assume the following sequence of events.

1. Main starts running and creates both threads and is then context switched right after.
2. Thread2 is scheduled and run until after it increments hello and is switched.
3. Thread1 runs until it is switched.
4. Main thread resumes and runs until it is switched.
5. Thread2 runs to completion and exits.
6. Main thread runs to completion but doesn't exit.
7. Thread1 runs to completion.

4. What is a major issue with this implementation of synchronization?

Using a while loop with `test_and_set` involves a ton of busy waiting.

We should let the thread sleep while it can't make any progress, so the CPU is available for use by another thread.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  _____;
  _____;
  _____;
  shared_data->data = -1;
}

int wait_for_data(shared_data_t* shared_data) {
  _____;
  int data = shared_data->data;
  _____;
  int ref_cnt = --shared_data->ref_cnt;
  _____;
  if (ref_cnt == 0)
    _____;
  return data;
}
```

```c
void* save_data(void* shared_pg) {
  _____;
  shared_data->data = 162;
  _____;
  _____;
  int ref_cnt = --shared_data->ref_cnt;
  _____;
  if (ref_cnt == 0)
    _____;
  return NULL;
}

int main() {
  void* shared_data = malloc(sizeof(shared_data_t));
  initialize_shared_data(shared_data);
  pthread_t tid;
  int error = pthread_create(&tid, NULL, &save_data, shared_data);
  int data = wait_for_data(shared_data);
  printf("Parent: Data is %d\n", data);
  return 0;
}
```

Want to design a program where main thread will wait for other thread to update `shared_data->data` to 162 before printing, so the output is always "`Parent: Data is 162`".

Shared Data

```
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;
```

1. Explain the purpose of each member in the `shared_data_t` struct.

Shared Data

```
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;
```

1. Explain the purpose of each member in the `shared_data_t` struct.

semaphore is used to implement the scheduling workflow.

- Main thread can down semaphore while the other thread can up semaphore after saving the data.

`lock` allows for mutual exclusion on members of the struct that can be modified by both threads.

`ref_cnt` allows for reference counting, which is an indicator for how many threads still hold a reference to this struct.

- Free when `ref_cnt` reaches 0.

`data` is the actual data.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  _____;
  _____;
  _____;
  shared_data->data = -1;
}
```

2. Fill in the missing lines in `initialize_shared_data` to correctly

initialize the members of `shared_data`.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  sem_init(&shared_data->semaphore, 0, 0);
  _____;
  _____;
  shared_data->data = -1;
}
```

2. Fill in the missing lines in `initialize_shared_data` to correctly initialize the members of `shared_data`.

Initialize `semaphore` with a value of 0. When using `sem_init`, the third argument is the actual value of the semaphore. The second argument is `pshared`, where a 0 value indicates the semaphore is shared between threads within a process.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  sem_init(&shared_data->semaphore, 0, 0);
  pthread_mutex_init(&shared_data->lock, NULL);
  _____;
  shared_data->data = -1;
}
```

2. Fill in the missing lines in `initialize_shared_data` to correctly initialize the members of `shared_data`.

Initialize `lock` with default attributes.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  sem_init(&shared_data->semaphore, 0, 0);
  pthread_mutex_init(&shared_data->lock, NULL);
  shared_data->ref_cnt = 2;
  shared_data->data = -1;
}
```

2. Fill in the missing lines in `initialize_shared_data` to correctly initialize the members of `shared_data`.

Initialize `ref_cnt` to 2 since two threads will have access to the data.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

int wait_for_data(shared_data_t* shared_data) {
  sem_wait(&shared_data->semaphore);
  int data = shared_data->data;
  _____;
  int ref_cnt = --shared_data->ref_cnt;
  _____;
  if (ref_cnt == 0)
    _____;
  return data;
}
```

Down semaphore to wait for the other thread to update data.

3. Fill in the missing lines in `wait_for_data` to correctly wait for the other thread until the data is updated.

Shared Data

```
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

int wait_for_data(shared_data_t* shared_data) {
  _____;
  int data = shared_data->data;
  _____;
  int ref_cnt = --shared_data->ref_cnt;
  _____;
  if (ref_cnt == 0)
    _____;
  return data;
}
```

3.   Fill in the missing lines in `wait_for_data` to correctly wait for the
      other thread until the data is updated.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

int wait_for_data(shared_data_t* shared_data) {
  sem_wait(&shared_data->semaphore);
  int data = shared_data->data;
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    _____;
  return data;
}
```

Need to decrement `ref_cnt` within a critical section since the other thread could also modify it.

3.   Fill in the missing lines in `wait_for_data` to correctly wait for the other thread until the data is updated.

Shared Data

```
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

int wait_for_data(shared_data_t* shared_data) {
  sem_wait(&shared_data->sem);
  int data = shared_data->data;
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return data;
}
```

Free when `ref_cnt` reaches 0

3. Fill in the missing lines in `wait_for_data` to correctly wait for the other thread until the data is updated.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void* save_data(void* shared_pg) {
  _____;
  shared_data->data = 162;
  _____;
  _____;
  int ref_cnt = --shared_data->ref_cnt;
  _____;
  if (ref_cnt == 0)
    _____;
  return NULL;
}
```

4.   Fill in the missing lines in `save_data` to correctly update the data.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void* save_data(void* shared_pg) {
  shared_data_t* shared_data = (shared_data_t*) shared_pg;
  shared_data->data = 162;
  _____;
  _____;
  int ref_cnt = --shared_data->ref_cnt;
  _____;
  if (ref_cnt == 0)
    _____;
  return NULL;
}
```

Cast `shared_pg` to `shared_data_t*` type.

4. Fill in the missing lines in `save_data` to correctly update the data.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void* save_data(void* shared_pg) {
  shared_data_t* shared_data = (shared_data_t*)shared_pg;
  shared_data->data = 162;
  sem_post(&shared_data->sem);
  _____;
  int ref_cnt = --shared_data->ref_cnt;
  _____;
  if (ref_cnt == 0)
    _____;
  return NULL;
}
```

Up semaphore after setting data.

4.   Fill in the missing lines in `save_data` to correctly update the data.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void* save_data(void* shared_pg) {
  shared_data_t* shared_data = (shared_data_t*)shared_pg;
  shared_data->data = 162;
  sem_post(&shared_data->sem);
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return NULL;
}
```

Decrement `ref_cnt` and free `shared_data` in the same way as the main thread.

4.   Fill in the missing lines in `save_data` to correctly update the data.

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  sem_init(&shared_data->sem, 0, 0);
  pthread_mutex_init(&lock, NULL);
  shared_data->ref_cnt = 2;
  shared_data->data = -1;
}

int wait_for_data(shared_data_t* shared_data) {
  sem_wait(&shared_data->sem);
  int data = shared_data->data;
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return data;
}
```

```c
void* save_data(void* shared_pg) {
  shared_data_t* shared_data = (shared_data_t*)shared_pg;
  shared_data->data = 162;
  sem_post(&shared_data->sem);
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return NULL;
}

int main() {
  void* shared_data = malloc(sizeof(shared_data_t));
  initialize_shared_data(shared_data);
  pthread_t tid;
  int error = pthread_create(&tid, NULL, &save_data, shared_data);
  int data = wait_for_data(shared_data);
  printf("Parent: Data is %d\n", data);
  return 0;
}
```

5.   Why does `shared_data->data` not need to be surrounded by locks when reading/writing to it in `wait_for_data`/`save_data`?

Shared Data

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  sem_init(&shared_data->sem, 0, 0);
  pthread_mutex_init(&lock, NULL);
  shared_data->ref_cnt = 2;
  shared_data->data = -1;
}

int wait_for_data(shared_data_t* shared_data) {
  sem_wait(&shared_data->sem);
  int data = shared_data->data;
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return data;
}
```

```c
void* save_data(void* shared_pg) {
  shared_data_t* shared_data = (shared_data_t*)shared_pg;
  shared_data->data = 162;
  sem_post(&shared_data->sem);
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return NULL;
}

int main() {
  void* shared_data = malloc(sizeof(shared_data_t));
  initialize_shared_data(shared_data);
  pthread_t tid;
  int error = pthread_create(&tid, NULL, &save_data, shared_data);
  int data = wait_for_data(shared_data);
  printf("Parent: Data is %d\n", data);
  return 0;
}
```

5. Why does `shared_data->data` not need to be surrounded by locks when reading/writing to it in `wait_for_data`/`save_data`?

Semaphore guarantees the main thread will never access `shared_data->data` before the other thread sets it. Similarly, `shared_data->data` is never modified by the other thread afterwards.

Condition Variables

Condition Variables and Monitors

Condition variables are synchronization variables that let a thread efficiently wait for a change to a shared state.

- A condition variable is a queue of threads where waiting threads are allowed to sleep **inside** the critical section (in contrast to other synchronization variables like semaphores).

A **monitor** is made up of a lock and zero or more condition variables.

Three condition variable operations:

- **Wait**: atomically releases lock and suspends execution of calling thread.
- **Signal**: wake up the next waiting thread in the queue.
- **Broadcast**: wake up all waiting threads in the queue.

Important: a thread must hold the lock when performing any of the condition variable operations.

Infinite synchronized buffer

```
Lock bufferLock;
ConditionVar bufferCV;

Producer() {
  bufferLock.acquire();
  put 1 coke in machine
  bufferCV.signal(bufferLock);
  bufferLock.release();
}

Consumer() {
  bufferLock.acquire();
  while (machine is empty)
    bufferCV.wait(bufferLock);
  take 1 coke out
  bufferLock.release();
}
```

Semantics

Hoare

Ownership of lock is *immediately transferred* to waiting thread when a thread is signaled.

After thread releases lock, ownership of lock transferred back to signaling thread.

Signaling can be thought of as atomic.

```
if (machine is empty)
  bufferCV.wait(bufferLock);
take 1 coke out
```

Mesa

No guarantees about execution order when a thread is signaled.

```
while (machine is empty)
  bufferCV.wait(bufferLock);
take 1 coke out
```

Need while loop because condition can change due to a thread interleaving between signaling and waiting thread.

Condition Check

```
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;
void print_hello() {
  hello += 1;
  printf("First line (hello=%d)\n", hello);
  pthread_cond_signal(&cv);
  pthread_exit(0);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, (void*)&print_hello, NULL);
  while (hello < 1)
    pthread_cond_wait(&cv, &lock);
  printf("Second line (hello=%d)\n", hello);
}
```

1. Will this program compile/run?

Condition Check

```c
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;
void print_hello() {
  hello += 1;
  printf("First line (hello=%d)\n", hello);
  pthread_cond_signal(&cv);
  pthread_exit(0);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, (void*)&print_hello, NULL);
  while (hello < 1)
    pthread_cond_wait(&cv, &lock);
  printf("Second line (hello=%d)\n", hello);
}
```

1. Will this program compile/run?

Will not run properly since the thread needs to be holding the lock before performing a condition variable operation like wait or signal.

Lock and condition variable were also never initialized.

Condition Check

```c
int ben = 0;
_____;
_____;

void* helper(void* arg) {
  _____;
  ben += 1;
  _____;
  _____;
  pthread_exit(NULL);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, NULL);
  pthread_yield();
  _____;
  _____;
  _____;
  if (ben == 1)
    printf("Yeet Haw\n");
  else
    printf("Yee Howdy\n");
  _____;
}
```

2. Fill in the blanks such that the program always prints "Yeet Haw".

Assume the system behaves with Mesa semantics.

Condition Check

```c
int ben = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void* helper(void* arg) {
  _____;
  ben += 1;
  _____;
  _____;
  pthread_exit(NULL);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, NULL);
  pthread_yield();
  _____;
  _____;
  _____;
  if (ben == 1)
    printf("Yeet Haw\n");
  else
    printf("Yee Howdy\n");
  _____;
}
```

2. Fill in the blanks such that the program always prints "Yeet Haw".
   Assume the system behaves with Mesa semantics.

Initialize a lock and condition variable.

Condition Check

```c
int ben = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void* helper(void* arg) {
  pthread_mutex_lock(&lock);
  ben += 1;
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
  pthread_exit(NULL);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, NULL);
  pthread_yield();
  _____;
  _____;
  _____;
  if (ben == 1)
    printf("Yeet Haw\n");
  else
    printf("Yee Howdy\n");
  _____;
}
```

2. Fill in the blanks such that the program always prints "Yeet Haw". Assume the system behaves with Mesa semantics.

The helper thread should signal once it has incremented ben. Make sure to acquire the lock before signaling.

Condition Check

```c
int ben = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void* helper(void* arg) {
  pthread_mutex_lock(&lock);
  ben += 1;
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
  pthread_exit(NULL);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, NULL);
  pthread_yield();
  pthread_mutex_lock(&lock);
  while (ben != 1)
    pthread_cond_wait(&cv, &lock);
  if (ben == 1)
    printf("Yeet Haw\n");
  else
    printf("Yee Howdy\n");
  pthread_mutex_unlock(&lock);
}
```

2. Fill in the blanks such that the program always prints "Yeet Haw". Assume the system behaves with Mesa semantics

The main thread should use pthread_cond_wait to sleep until signaled by the helper thread. Again, make sure to acquire the lock before performing any condition variable operations (in this case, pthread_cond_wait).