ACCUKNOX - Django Signals Assessment

Question 1:

Are Django Signals Executed Synchronously or Asynchronously by Default?

Django signals are executed *synchronously* by default.

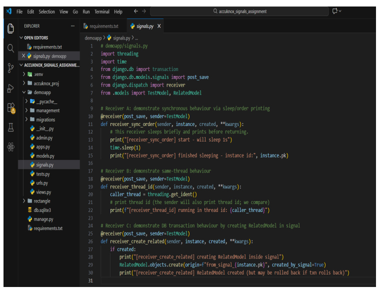
When a signal is triggered (e.g., using post_save), Django immediately calls the connected signal handlers before returning control to the original caller.

This means:

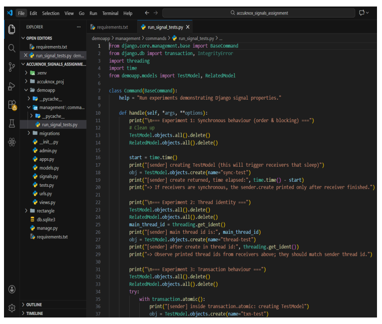
- The main function waits until the signal handler finishes.
- Signal handlers run in blocking mode.
- Execution order is strictly sequential.

Django does NOT use threads or async by default for signal handlers.

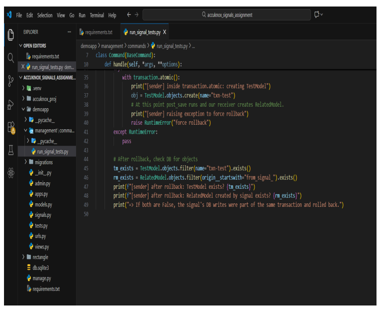
Synchronous Behavior



```python
# demoapp/signals.py
import threading
import time
from django.db import transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import TestModel, RelatedModel

# Receiver A: demonstrate synchronous behaviour via sleep/order printing
@receiver(post_save, sender=TestModel)
def receiver_sync_order(sender, instance, created, **kwargs):
    # This receiver sleeps briefly and prints before returning.
    print("[receiver_sync_order] start - will sleep 1s")
    time.sleep(1)
    print("[receiver_sync_order] finished sleeping - instance id:", instance.pk)

# Receiver B: demonstrate same-thread behaviour
@receiver(post_save, sender=TestModel)
def receiver_thread_id(sender, instance, created, **kwargs):
    caller_thread = threading.get_ident()
    # print thread id (the sender will also print thread id; we compare)
    print(f"[receiver_thread_id] running in thread id: {caller_thread}")

# Receiver C: demonstrate DB transaction behaviour by creating RelatedModel in signal
@receiver(post_save, sender=TestModel)
def receiver_create_related(sender, instance, created, **kwargs):
    if created:
        print("[receiver_create_related] creating RelatedModel inside signal")
        RelatedModel.objects.create(origin=f"from_signal_{instance.pk}", created_by_signal=True)
        print("[receiver_create_related] RelatedModel created (but may be rolled back if txn rolls back)")

```

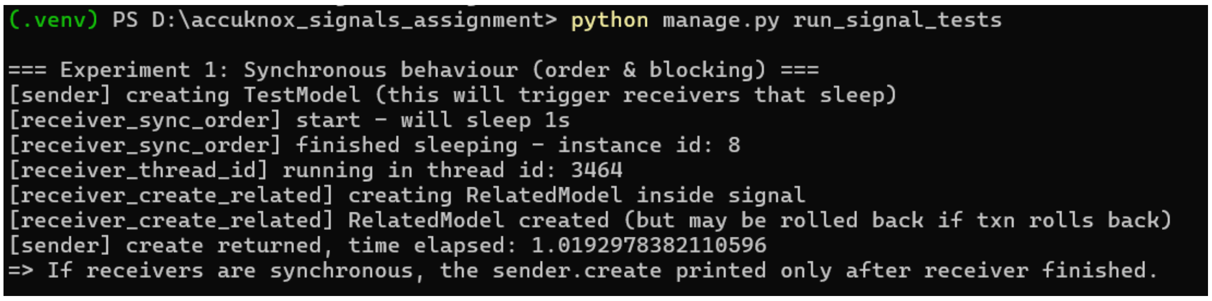


```python
from django.core.management.base import BaseCommand
from django.db import transaction, IntegrityError
import threading
import time
from demoapp.models import TestModel, RelatedModel

class Command(BaseCommand):
    help = "Run experiments demonstrating Django signal properties."

    def handle(self, *args, **options):
        print("\n=== Experiment 1: Synchronous behaviour (order & blocking) ===")
        # Clean up
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()

        start = time.time()
        print("[sender] creating TestModel (this will trigger receivers that sleep)")
        obj = TestModel.objects.create(name="sync-test")
        print("[sender] create returned, time elapsed:", time.time() - start)
        print("=> If receivers are synchronous, the sender.create printed only after receiver finished.")

        print("\n=== Experiment 2: Thread identity ===")
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()
        main_thread_id = threading.get_ident()
        print("[sender] main thread id is:", main_thread_id)
        obj = TestModel.objects.create(name="thread-test")
        print("[sender] after create in thread id:", threading.get_ident())
        print("=> Observe printed thread ids from receivers above; they should match sender thread id.")

        print("\n=== Experiment 3: Transaction behaviour ===")
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()
        try:
            with transaction.atomic():
                print("[sender] inside transaction.atomic: creating TestModel")
                obj = TestModel.objects.create(name="txn-test")
```



```python
class Command(BaseCommand):
    def handle(self, *args, **options):
            with transaction.atomic():
                print("[sender] inside transaction.atomic: creating TestModel")
                obj = TestModel.objects.create(name="txn-test")
                # At this point post_save runs and our receiver creates RelatedModel.
                print("[sender] raising exception to force rollback")
                raise RuntimeError("force rollback")
        except RuntimeError:
            pass

        # After rollback, check DB for objects
        tm_exists = TestModel.objects.filter(name="txn-test").exists()
        rm_exists = RelatedModel.objects.filter(origin__startswith="from_signal_").exists()
        print(f"[sender] after rollback: TestModel exists? {tm_exists}")
        print(f"[sender] after rollback: RelatedModel created by signal exists? {rm_exists}")
        print("=> If both are False, the signal's DB writes were part of the same transaction and rolled back.")

```

Output



```
(.venv) PS D:\accuknox_signals_assignment> python manage.py run_signal_tests

=== Experiment 1: Synchronous behaviour (order & blocking) ===
[sender] creating TestModel (this will trigger receivers that sleep)
[receiver_sync_order] start - will sleep 1s
[receiver_sync_order] finished sleeping - instance id: 8
[receiver_thread_id] running in thread id: 3464
[receiver_create_related] creating RelatedModel inside signal
[receiver_create_related] RelatedModel created (but may be rolled back if txn rolls back)
[sender] create returned, time elapsed: 1.0192978382110596
=> If receivers are synchronous, the sender.create printed only after receiver finished.
```

The signal handler **blocks the main thread**, proving **synchronous execution**.

Question 2:

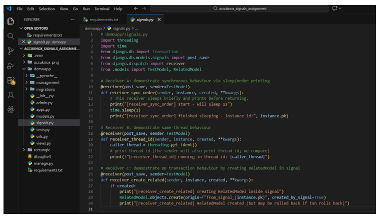
Do Django Signals Run in the Same Thread as the Caller?

Yes, Django signals run in the *same thread* as the caller.

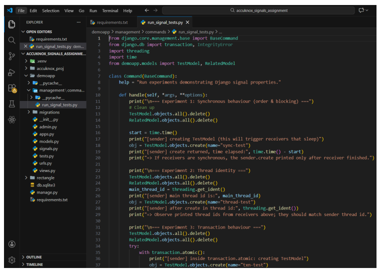
Django internally dispatches signals using the same thread that triggered the action (e.g., saving a model).

There is no thread spawning, no async logic, no parallel execution.

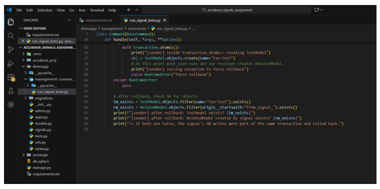
Code



```python
# demoapp/signals.py
import threading
import time
from django.db import transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import TestModel, RelatedModel

# Receiver A: demonstrate synchronous behaviour via sleep/order printing
@receiver(post_save, sender=TestModel)
def receiver_sync_order(sender, instance, created, **kwargs):
    # This receiver sleeps briefly and prints before returning.
    print("[receiver_sync_order] start - will sleep 1s")
    time.sleep(1)
    print("[receiver_sync_order] finished sleeping - instance id:", instance.pk)

# Receiver B: demonstrate same-thread behaviour
@receiver(post_save, sender=TestModel)
def receiver_thread_id(sender, instance, created, **kwargs):
    caller_thread = threading.get_ident()
    # print thread id (the sender will also print thread id; we compare)
    print(f"[receiver_thread_id] running in thread id: {caller_thread}")

# Receiver C: demonstrate DB transaction behaviour by creating RelatedModel in signal
@receiver(post_save, sender=TestModel)
def receiver_create_related(sender, instance, created, **kwargs):
    if created:
        print("[receiver_create_related] creating RelatedModel inside signal")
        RelatedModel.objects.create(origin=f"from_signal_{instance.pk}", created_by_signal=True)
        print("[receiver_create_related] RelatedModel created (but may be rolled back if txn rolls back)")

```

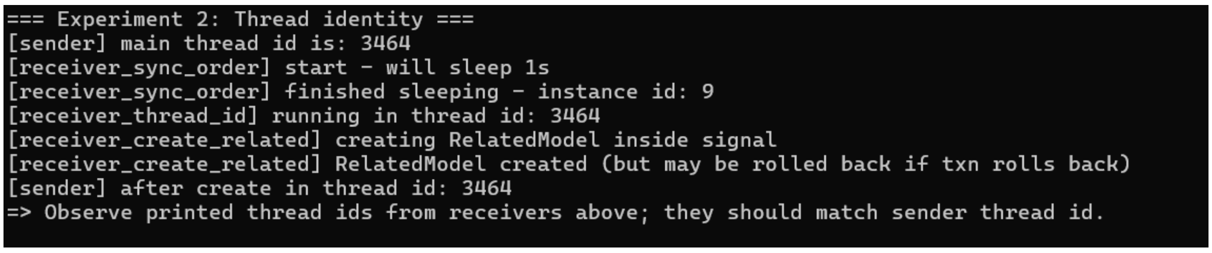


```python
from django.core.management.base import BaseCommand
from django.db import transaction, IntegrityError
import threading
import time
from demoapp.models import TestModel, RelatedModel

class Command(BaseCommand):
    help = "Run experiments demonstrating Django signal properties."

    def handle(self, *args, **options):
        print("\n=== Experiment 1: Synchronous behaviour (order & blocking) ===")
        # Clean up
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()

        start = time.time()
        print("[sender] creating TestModel (this will trigger receivers that sleep)")
        obj = TestModel.objects.create(name="sync-test")
        print("[sender] create returned, time elapsed:", time.time() - start)
        print("=> If receivers are synchronous, the sender.create printed only after receiver finished.")

        print("\n=== Experiment 2: Thread identity ===")
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()
        main_thread_id = threading.get_ident()
        print("[sender] main thread id is:", main_thread_id)
        obj = TestModel.objects.create(name="thread-test")
        print("[sender] after create in thread id:", threading.get_ident())
        print("=> Observe printed thread ids from receivers above; they should match sender thread id.")

        print("\n=== Experiment 3: Transaction behaviour ===")
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()
        try:
            with transaction.atomic():
                print("[sender] inside transaction.atomic: creating TestModel")
                obj = TestModel.objects.create(name="txn-test")
```



```python
class Command(BaseCommand):
    def handle(self, *args, **options):
            with transaction.atomic():
                print("[sender] inside transaction.atomic: creating TestModel")
                obj = TestModel.objects.create(name="txn-test")
                # At this point post_save runs and our receiver creates RelatedModel.
                print("[sender] raising exception to force rollback")
                raise RuntimeError("force rollback")
        except RuntimeError:
            pass

        # After rollback, check DB for objects
        tm_exists = TestModel.objects.filter(name="txn-test").exists()
        rm_exists = RelatedModel.objects.filter(origin__startswith="from_signal_").exists()
        print(f"[sender] after rollback: TestModel exists? {tm_exists}")
        print(f"[sender] after rollback: RelatedModel created by signal exists? {rm_exists}")
        print("=> If both are False, the signal's DB writes were part of the same transaction and rolled back.")

```

Output



```
=== Experiment 2: Thread identity ===
[sender] main thread id is: 3464
[receiver_sync_order] start - will sleep 1s
[receiver_sync_order] finished sleeping - instance id: 9
[receiver_thread_id] running in thread id: 3464
[receiver_create_related] creating RelatedModel inside signal
[receiver_create_related] RelatedModel created (but may be rolled back if txn rolls back)
[sender] after create in thread id: 3464
=> Observe printed thread ids from receivers above; they should match sender thread id.
```

Both threads show same ID, proving:

Django signals run in the same thread as the caller.

Question 3:

Do Django Signals Run in the Same Database Transaction as the Caller by Default?

Yes. Django signals run inside the *same database transaction* as the caller.

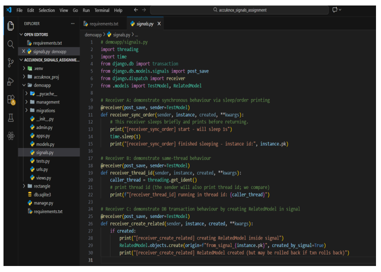
When you run:

TestModel.objects.create(...)

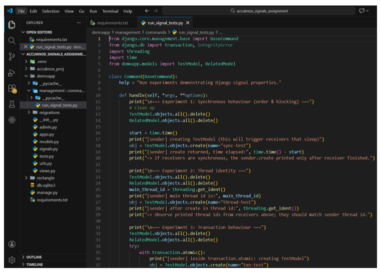
Django:

1. Starts a database transaction
2. Saves the model
3. Sends the signal before committing the transaction
4. Commits the transaction only after signal handlers run

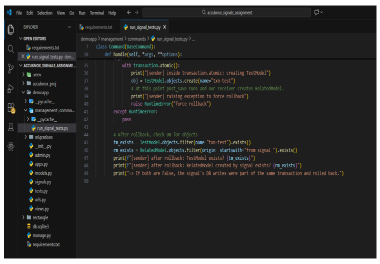
So if the signal handler fails, the original save is also rolled back.



```python
# demoapp/signals.py
import threading
import time
from django.db import transaction
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import TestModel, RelatedModel

# Receiver A: demonstrate synchronous behaviour via sleep/order printing
@receiver(post_save, sender=TestModel)
def receiver_sync_order(sender, instance, created, **kwargs):
    # This receiver sleeps briefly and prints before returning.
    print("[receiver_sync_order] start - will sleep 1s")
    time.sleep(1)
    print("[receiver_sync_order] finished sleeping - instance id:", instance.pk)

# Receiver B: demonstrate same-thread behaviour
@receiver(post_save, sender=TestModel)
def receiver_thread_id(sender, instance, created, **kwargs):
    caller_thread = threading.get_ident()
    # print thread id (the sender will also print thread id; we compare)
    print(f"[receiver_thread_id] running in thread id: {caller_thread}")

# Receiver C: demonstrate DB transaction behaviour by creating RelatedModel in signal
@receiver(post_save, sender=TestModel)
def receiver_create_related(sender, instance, created, **kwargs):
    if created:
        print("[receiver_create_related] creating RelatedModel inside signal")
        RelatedModel.objects.create(origin=f"from_signal_{instance.pk}", created_by_signal=True)
        print("[receiver_create_related] RelatedModel created (but may be rolled back if txn rolls back)")
```

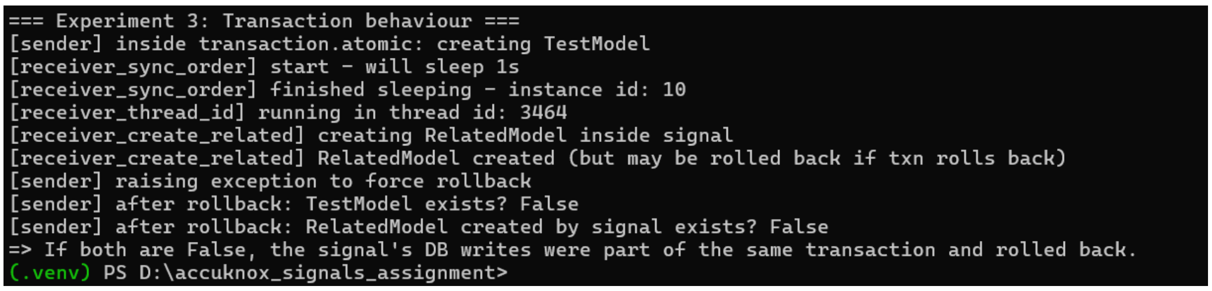


```python
from django.core.management.base import BaseCommand
from django.db import transaction, IntegrityError
import threading
import time
from demoapp.models import TestModel, RelatedModel

class Command(BaseCommand):
    help = "Run experiments demonstrating Django signal properties."

    def handle(self, *args, **options):
        print("\n=== Experiment 1: Synchronous behaviour (order & blocking) ===")
        # Clean up
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()

        start = time.time()
        print("[sender] creating TestModel (this will trigger receivers that sleep)")
        obj = TestModel.objects.create(name="sync-test")
        print("[sender] create returned, time elapsed:", time.time() - start)
        print("=> If receivers are synchronous, the sender.create printed only after receiver finished.")

        print("\n=== Experiment 2: Thread identity ===")
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()
        main_thread_id = threading.get_ident()
        print("[sender] main thread id is:", main_thread_id)
        obj = TestModel.objects.create(name="thread-test")
        print("[sender] after create in thread id:", threading.get_ident())
        print("=> Observe printed thread ids from receivers above; they should match sender thread id.")

        print("\n=== Experiment 3: Transaction behaviour ===")
        TestModel.objects.all().delete()
        RelatedModel.objects.all().delete()
        try:
            with transaction.atomic():
                print("[sender] inside transaction.atomic: creating TestModel")
                obj = TestModel.objects.create(name="txn-test")
```



```python
class Command(BaseCommand):
    def handle(self, *args, **options):
            with transaction.atomic():
                print("[sender] inside transaction.atomic: creating TestModel")
                obj = TestModel.objects.create(name="txn-test")
                # At this point post_save runs and our receiver creates RelatedModel.
                print("[sender] raising exception to force rollback")
                raise RuntimeError("force rollback")
        except RuntimeError:
            pass

        # After rollback, check DB for objects
        tm_exists = TestModel.objects.filter(name="txn-test").exists()
        rm_exists = RelatedModel.objects.filter(origin__startswith="from_signal_").exists()
        print(f"[sender] after rollback: TestModel exists? {tm_exists}")
        print(f"[sender] after rollback: RelatedModel created by signal exists? {rm_exists}")
        print("=> If both are False, the signal's DB writes were part of the same transaction and rolled back.")
```

Output



```
=== Experiment 3: Transaction behaviour ===
[sender] inside transaction.atomic: creating TestModel
[receiver_sync_order] start - will sleep 1s
[receiver_sync_order] finished sleeping - instance id: 10
[receiver_thread_id] running in thread id: 3464
[receiver_create_related] creating RelatedModel inside signal
[receiver_create_related] RelatedModel created (but may be rolled back if txn rolls back)
[sender] raising exception to force rollback
[sender] after rollback: TestModel exists? False
[sender] after rollback: RelatedModel created by signal exists? False
=> If both are False, the signal's DB writes were part of the same transaction and rolled back.
(.venv) PS D:\accuknox_signals_assignment>
```

Both caller and signal run inside the same transaction block.
Django signals run within the same transaction as the caller.