CS 340
Computer Systems

#10/11: Deadlock and Producer-Consumer

Sept. 27, 2022 · Wade Fagen-Ulmschneider

Solving Deadlock

On Thursday, we explored the four necessary conditions for deadlock. In the context of the dining philosophers problem, how do we remove each of the four?

1. Mutual Exclusion

2. Circular Wait

3. Hold and Wait

4. No Preemption

Deadlock Solution Considerations

1. Fairness:

2. Livelock:

10/producer-consumer-2.c

```c
#define THINGS_MAX 10
#define THREAD_CT 5

int things[THINGS_MAX];
int things_ct = 0;



```

```c
void *producer(void *vptr) {
 while (1) {










  assert(things_ct <
                THINGS_MAX);

  // Produce a thing:
  things[things_ct] =
              rand() % 100;
  printf("Produced [%d]: %d
     -> ", things_ct,
          things[things_ct]);
  things_ct++;
  print_things_as_list();






 }
}
```

```c
void *consumer(void *vptr) {
 while (1) {










  assert(things_ct > 0);

  // Consume a thing:
  things_ct--;
  int value =
         things[things_ct];
  printf("Consumed [%d]: %d
           <- ", things_ct,
                value);
  print_things_as_list();







 }
}
```

```c
int main() {
  int i;

  // Create `thread_ct` threads of each producer and consumer:
  pthread_t tid_consumer[THREAD_CT];
  pthread_t tid_producer[THREAD_CT];
  for (i = 0; i < THREAD_CT; i++) {
    pthread_create(&tid_consumer[i], NULL, producer, NULL);
    pthread_create(&tid_producer[i], NULL, consumer, NULL);
  }

  // Join threads:
  for (i = 0; i < THREAD_CT; i++) {
    pthread_join(tid_consumer[i], NULL);
    pthread_join(tid_producer[i], NULL);
  }
}
```

Synchronization Primitives

In programming, a key synchronization primitive has evolved to become common features of many programming languages.

Primitive: ______________________________

- Allow asynchronous execution until a _________________.
- In JavaScript and Python, this is ____________________.

10/barrier.c

```c
typedef pthread_t * promise_t;

void *idle_task(void *vptr) {
  printf("idle_task is running.\n");
  sleep(5);
  printf("idle_task has finished.\n");
  return NULL;
}

promise_t async_task(void*(* task)(void *), void *arg) {
  pthread_t *tidptr = malloc(sizeof(pthread_t));
  pthread_create(tidptr, NULL, task, arg);
  return tidptr;
}

void *async_wait(promise_t promise) {
  void *result;
  pthread_join(*promise, &result);
  return result;
}

int main() {
  promise_t p = async_task(idle_task, NULL);
  printf("main thread is running at the same time...\n");
  async_wait(p);
  printf("main thread has running again...\n");
  return 0;
}
```

General Pattern:

- An “async” call returns a ______________, not a return value.
- The “await” call ________ and then ____________________.

Multi-Threaded Uses

There are several different reasons you will come across the use of multi-threaded applications:

(1): Concurrent Compute

(2): User Interactivity

(3): Responsiveness in Requests

(4): **<u>NOT</u>** for Security

(5): **<u>NOT</u>** for Isolation