PURDUE UNIVERSITY

CS 25200: Systems Programming

Lecture 23: Condition Synchronization, Bounded Buffer

Prof. Turkstra

Lecture 23

- POSIX Semaphores
- Condition synchronization
- Producer/consumer
- Bounded buffer

POSIX Semaphores

- Declaration

  ```
  #include <semaphore.h>
  sem_t sem;
  ```

- Initialization

  ```
  sem_init(sem_t *sem, int pshared, int value);
  ```

- Decrement

  ```
  sem_wait(sem_t *sem);
  ```

- Vincrement

  ```
  sem_post(sem_t *sem);
  ```

Semaphore

- count = 1 → mutex or lock
- count > 1 → permit n processes access
- count = 0 → wait for an event

count = 1

```
sem_wait(milk_semaphore)
if (!milk)
  buy milk
sem_post(milk_semaphore)
```

count = 3

```
sem_init(&sem, 3);

sem_wait(&sem)
print
sem_post(&sem)
```

- Suppose we have five threads

Drawbacks of MT LinkedList

- return -1 if list is empty
  - Would rather wait
- Need more than a lock or mutex to implement this

linked_list Semaphores

- Consider the linked_list functions
- empty_sem = 0
- remove() will call sem_wait(&empty_sem)
  - Block until insert() calls sem_post(&empty_sem)
- count then represents number of items in list

```
typedef struct linked_list {
  pthread_mutex_t mutex;
  sem_t empty_sem;
  struct node *head;
} linked_list;

typedef struct node {
  int value;
  struct node *next;
} node;

linked_list *list = malloc(sizeof(list));
pthread_mutex_init(&list->mutex, NULL);
sem_init(&list->empty_sem, 0, 0);
```

```
int insert(linked_list *list, int val) {
  pthread_mutex_lock(&list->mutex);
  if (!list->head) {
    list->head = malloc(sizeof(node));
    list->head->value = val;
  }
  else {
    node *new_node = malloc(sizeof(node));
    new_node->next = list->head;
    list->head = new_node;
  }
  pthread_mutex_unlock(&list->mutex);
  sem_post(&list->empty_sem);
  return;
}
```

```
int remove_head(linked_list *list) {
  Node *tmp = NULL;
  int val;

  sem_wait(&list->empty_sem);
  pthread_mutex_lock(&list->mutex);
  tmp = list->head;
  if (tmp == NULL) {
    pthread_mutex_unlock(&list->mutex);
    return -1;
  }
  list->head = list->head->next;
  pthread_mutex_unlock(&list->mutex);
  val = tmp->value;
  free(tmp);
  tmp = NULL;

  return val;
}
```

Notes

- Still need the mutex, right?
- What happens if we sem_wait() inside the critical section?

> N

N

1

N = items in list

```
int remove_head(linked_list *list) {
  node *tmp = NULL;
  int val;

  sem_wait(&list->empty_sem);



  pthread_mutex_lock(&list->mutex);
  tmp = list->head;
  if (tmp == NULL) {
    pthread_mutex_unlock(&list->mutex);
    return -1;
  }
  list->head = list->head->next;
  pthread_mutex_unlock(&list->mutex);
  val = tmp->value;
  free(tmp);
  tmp = NULL;

  return val;
}
```

Purdue Trivia

- Slayter Center of the Performing Arts
  - Completed in 1964, dedicated May 1, 1965
  - Gift from Dr. Games Slayter and wife Marie
  - Designed to reflect Stonehenge

Producer/Consumer Problem

- Producer: creates instances of a resource
- Consumer: uses (destroys) instances of a resource
- Buffers: used to convey resources between the two
- Synchronization: producer and consumer are synchronized

Constraints

- Consumer must wait for producer to fill buffers
- Producer must wait for consumer to empty buffers if all buffer space used
- Only one process may use buffer pool at once

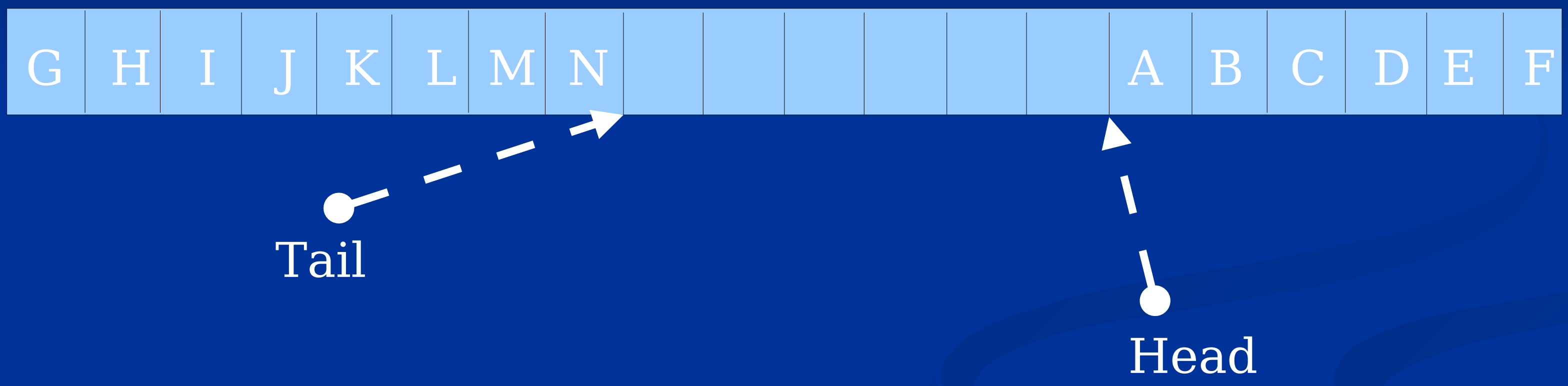
Producer	Consumer
while (1) {	while (1) {
get empty buffer from pool of empties	get full buffer from pool of fulls
produce data in buffer	consume data from buf
put full buffer in pool of fulls	put empty buffer in pool of empties
}	}

Bounded Buffer

- Suppose we have a circular buffer that is of a fixed size
- Want multiple threads to communicate using this buffer
- Common paradigm in device drivers and pipes
  - Although usually have a pool of buffers instead of a single buffer

Circular buffer

- 20 byte buffer with 14 elements



Interface

- enqueue() - add an item to the queue
  - Block if queue is full
- dequeue() - remove an item from the queue
  - Block if empty
- Need condition synchronization – empty and full
- Do we need mutual exclusion?

Implementation

```
#include <pthread.h>
#define MAXSIZE 32

typedef struct {
  char queue[MAXSIZE];
  int head;
  int tail;
  pthread_mutex_t mutex;
  sem_t empty_sem;
  sem_t full_sem;
} bounded_buffer;
```

Initialization

```
bounded_buffer bb;

pthread_mutex_init(&bb->mutex, NULL);
sem_init(&bb->empty_sem, 0, 0);
sem_init(&bb->full_sem, 0, MAXSIZE);
bb->head = 0;
bb->tail = 0;
```

enqueue() and dequeue()

```
void enqueue(int val) {
  sem_wait(&bb->full_sem);
  pthread_mutex_lock(&bb->mutex);
  bb->queue[tail] = val;
  bb->tail = (tail + 1) % MAXSIZE;
  pthread_mutex_unlock(&bb->mutex);
  sem_post(&bb->empty_sem);
}

int dequeue() {
  sem_wait(&bb->empty_sem);
  pthread_mutex_lock(&bb->mutex);
  int val = queue[head];
  head = (head + 1) % MAXSIZE;
  pthread_mutex_unlock(&bb->mutex);
  sem_post(&bb->full_sem);
  return val;
}
```

Bounded buffer notes

- empty_sem represents the number of items in the queue
- full_sem represents the number of (empty) spaces in the queue
- If this seems backwards, you can switch them
- So, do we still need mutexes?

Questions?