CS3.301 Operating Systems and Networks

Concurrency - Condition Variables

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

SERC
Software Engineering Research Centre

Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems in three easy pieces by Remzi et al.

What if we leverage Turns and Tickets

Think about going to some crowded office space

Source: juvale.com

Fetch and Add

Yet another hardware primitive but very powerful

Fetch And Add

```
int FetchAndAdd(int *ptr)
{
  int old = *ptr;
  *ptr = old + 1;
  return old;    //returns old value
}
```

- Atomically increment a value while returning the old value at a particular address
- Used to build interesting type of lock - **The ticket lock**
- Instead of single variable a combination of ticket and turn variable is used
- Not just flag: ticket and turn

Ticket Lock

Ticket Lock

```c
typedef struct __lock_t
{
    int ticket;
    int turn;
} lock_t;

void lock_init (lock_t *lock)
{
  lock_t -> ticket = 0;
  lock_t -> turn = 0;
}
```

Ticket Lock

```c
void lock (lock_t *lock)
{
  int myturn = FetchAndAdd(&lock->ticket);
  //when ticket value = my turn, thread goes into CS
  while (lock->turn != myturn)
  {
    // keep spinning
  }
}

void unlock (lock_t *lock)
{
  // next waiting thread can enter CS
  FetchAndAdd(&lock->turn);
}
```

An Illustration of Ticket Lock

Four Processor Ticket Lock Example

Row	Action	next_ticket	now_serving	P1 my_ticket	P2 my_ticket	P3 my_ticket	P4 my_ticket
1	Initialized to 0	0	0	-	-	-	-
2	P1 tries to acquire lock (succeed)	1	0	0	-	-	-
3	P3 tries to acquire lock (fail + wait)	2	0	0	-	1	-
4	P2 tries to acquire lock (fail + wait)	3	0	0	2	1	-
5	P1 releases lock, P3 acquires lock	3	1	0	2	1	-
6	P3 releases lock, P2 acquires lock	3	2	0	2	1	-
7	P4 tries to acquire lock (fail + wait)	4	2	0	2	1	3
8	P2 releases lock, P4 acquires lock	4	3	0	2	1	3
9	P4 releases lock	4	4	0	2	1	3
10	...	4	4	0	2	1	3

Source: wikipedia article on ticket lock

How good is the spin based locks?

- Simple hardware based locks are simple to implement and powerful
- They are also quite inefficient especially when it comes to performance
  - Consider that there are two threads and one thread has the lock
  - When thread has lock, it may get interrupted, the other thread spins for a time slice, waste CPU cycle
  - Think about N threads, N-1 threads might waste CPU cycles in spinning (especially if round robin)
- **Can we come up with something better instead of wasting cycles with spinning?**

OS support can help

The yield call

- If the thread is aware that it is going to spin - Why not give up the CPU to some other thread?
  - Simple OS primitive system call: **yield()**
  - Moves the thread from running state to ready state => another thread can run
  - Does this solution work efficiently?
    - What if there are 100 threads?
    - Still costly! - 99 threads runs to yield
    - Possibility of infinite yields as well - **Starvation! - Why?**

```
while (TestAndSet(&lock->flag,1) == 1)
{
    yield ();
}
```

Can we make thread sleep rather than spinning?

- Why don’t we make use of some queue based structures?
- Keep a queue to track which thread needs access to CS
- Syscalls by Solaris: **park()** and **unpark()**
  - park(): puts a thread to sleep
  - unpark(tid): wakeup that particular thread
- If a thread wants to acquire a lock
  - Check if others have the lock, if yes put thread to sleep
  - If lock is free, wake up the thread and give the lock

Locks do help in access to CS! But more challenges

- Locks ensures that thread can get access to CS
  - With help of HW and SW mechanisms efficient locks can be built
- But, thread while executing may want to check for some conditions
  - A parent thread may want to check if the child thread has completed before proceeding
    - Remember join() operation? - How to make it work?
    - **Why don't we use shared variable?**

Checks using shared variable

```
int done = 0;

void *child (void *arg)
{
  printf (" child\n");
  done = 1;
  return NULL;
}

int main (int argc, char *argv[])
{
  printf ("parent\n");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  while (done == 0)
  {
    ; //Keep spinning
  }
  printf (" done \n");
  return 0;
}
```

Condition Variables

- **Condition variable:** Explicit queues that the threads can put themselves on when a state of condition is not as desired
    - **Eg:** lock is not available (flag might be 0)
- When condition is met, thread can be woken up to continue

pthread_cond_t c;

- c is a condition variable with two operations - wait() and signal()
- **wait():** when thread wants to put itself to sleep
- **signal():** there is some change and thread wants to wake up thread waiting on condition

Condition Variables in Action

```c
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
void thread_exit()
{
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void *worker_thread(void *arg)
{
    printf("child \n");
    thread_exit();
    return NULL;
}
```

```c
void thread_join()
{
    pthread_mutex_lock(&m);
    while (done == 0)
    {
        pthread_cond_wait(&c,&m);
    }
    pthread_mutex_unlock(&m);
}

int main (int argc, char *argv[])
{
    pthread_t thread_p1;
    printf("Starting parent thread \n");
    pthread_create(&thread_p1, NULL, worker_thread, NULL);
    thread_join();
    printf("Parent: end\n");
    return 0;
}
```

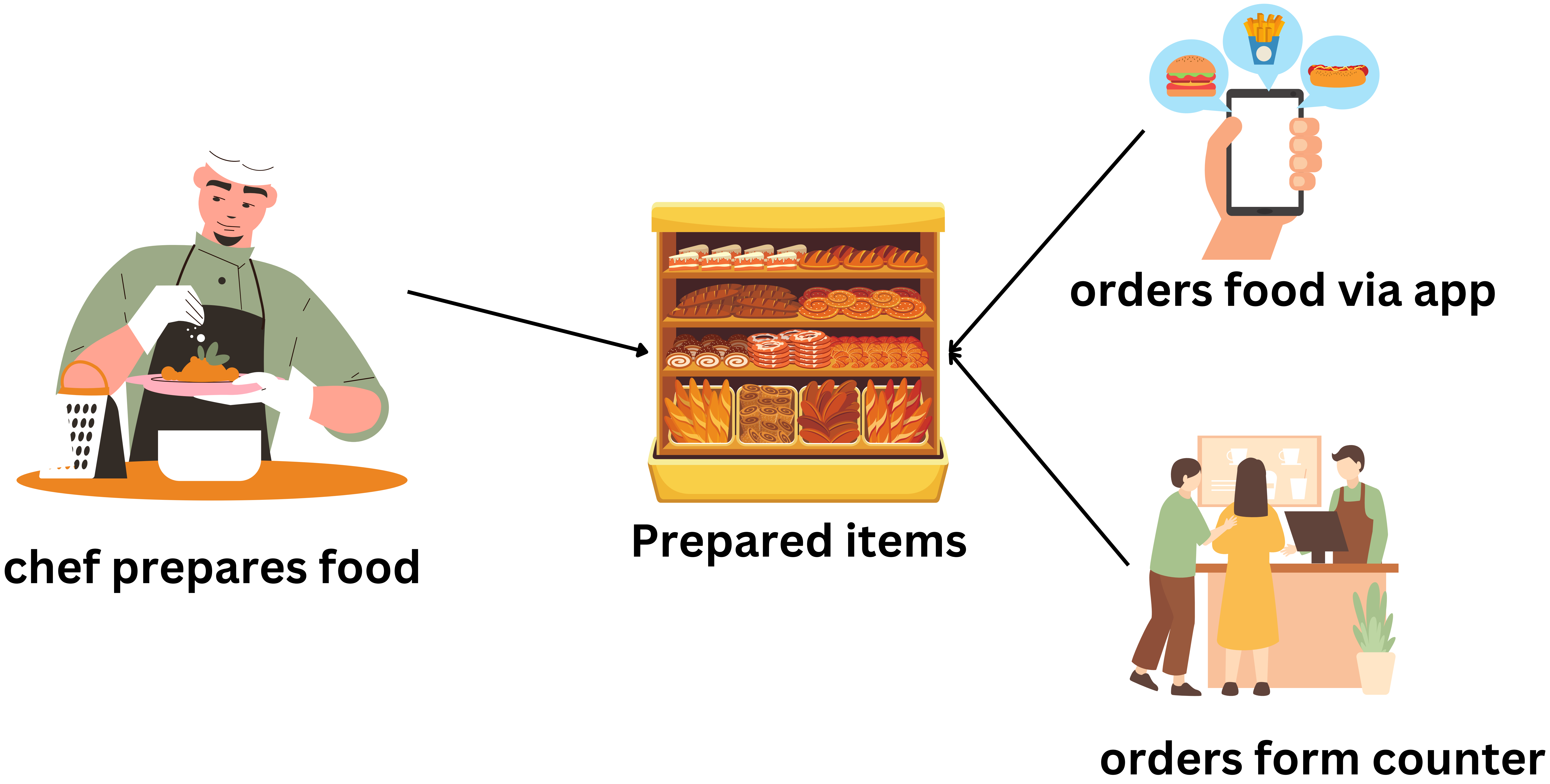
Two cases to consider as it works

- **Parent creates the Child and continues running**
  - Goes into the join call
  - Checks the state variable since child is not done, puts itself to sleep
  - Child runs and invokes exit -> updates state variable and wakes up parent thread
  - Parent will run returning from wait and prints done
- **Child runs immidiately upon creation**
  - Sets done to 1, wakes up sleeping thread (none available) so returns
  - Parent runs join, the done variable is 1 so returns
- **Do we need while loop for checking state and do we need locks?**

State variable and Locks

- **What if we don't have the state variable done**
  - Exit and join functions simply calls wait and join
  - What if child runs first and calls exit, child will signal but no parent thread
  - When parent runs, it will simply wait and never come out of it
- **What if there are no locks around statements in exit and join?**
  - Parent calls join, checks that done is 0, sleeps
  - Just before sleep call, interrupt, child runs and sets done to 1 and signals
  - No thread is waiting, parent runs goes into sleep and forever sleeps - **Race condition**

An Analogy



One cannot get food items that are not yet ready!

The Producer/Consumer Problem

AKA Bounded Buffer Problem

- Think about web servers
  - **Producer:** Produces HTTP requests into a queue
  - **Consumer:** The threads that process the HTTP requests from the queue
  - **Bounded buffer:** The work queue
- Piped calls in unix: grep linux os.txt | wc -l
  - **Producer:** grep gets text that contains “linux” from os.txt and puts them to standard output
  - **Consumer:** Shell redirects them to pipe call, where wc as another process counts and prints the number of lines
  - **Buffer:** Shared resource

Wait There is a challenge

- Bounded buffer is a shared resource
- Producer puts data to empty buffer
- Consumer can only consume from full buffer
- We need synchronisation mechanisms to access it
  - Else it may result in **race conditions**
- How to solve the problem?
- What kind of synchronisation mechanisms can be developed?

SCAN ME

Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere