TCSS 422: OPERATING SYSTEMS

Condition Variables

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Condition variables
- Consumer/Producer
- Covering condition

CONDITION VARIABLES

- There are many cases where a thread wants to wait for another thread before proceeding with execution
- Consider when a precondition must be fulfilled before it is meaningful to proceed …

CONDITION VARIABLES - 2

- Support a signaling mechanism to alert thread s when preconditions have been satisfied
- Eliminate busy waiting
- Alert one or more threads to "consume" a result, or respond to state changes in the application
- Threads are placed on an **explicit queue** (FIFO) to wait for signals
- **Signal**: wakes one thread
  broadcast wakes all (ordering by the OS)

CONDITION VARIABLES - 3

- Condition variable

```
pthread_cond_t c;
```

  - Requires initialization
- Condition API calls

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);   // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

- wait() accepts a mutex parameter
  - Releases lock, puts thread to sleep
- signal()
  - Wakes up thread, awakening thread acquires lock

EXAMPLE: MATRIX GENERATOR

- Thread A continuously generates 2-D matrices
- Thread B computes the average value of 2-D matrices
- Thread B has nothing to do before Thread A generates a matrix
- To simplify memory management, Thread A and Thread B share a pointer to the most recently created matrix
- Thread A can't generate a new array with the shared pointer before Thread B completes calculating an average

MATRIX GENERATOR - 2

- Solution ?
  - Thread B could employ a spin lock to continuously check a state variable
- Issues
  - Continuously making a comparison to check a state variable will occupy a CPU core
  - And without hardware support ensuring atomicity in critical sections can not be guaranteed

MATRIX GENERATOR - 3

Matrix generation example

MATRIX GENERATOR - 4

- Consider the importance of the **state** variable
- What will the code do if we remove it?
- Consider
  rows = 1000
  cols = 1000
- Consider
  rows = 10
  cols = 10

MATRIX GENERATOR - 5

- If the child thread is not created fast enough, or if there is a OS context before the child obtains the lock to generate the array, the program could deadlock…
- The program may still execute (no deadlock), but shared data could be changed out of sequence leading to errors
- Try adding printf statement to observe whether the child thread or parent (int main) is deadlocking…

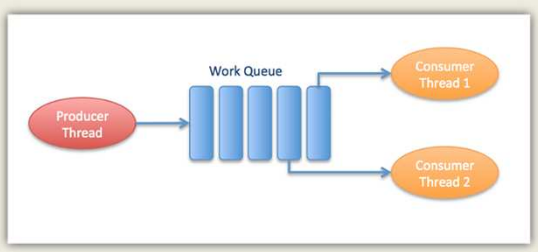
SUBTLE RACE CONDITION

```
void thr_exit() {
        done = 1;
        Pthread_cond_signal(&c);
}

void thr_join() {
        if (done == 0)
                Pthread_cond_wait(&c);
}
```

- Parent thread calls thr_join() and executes the comparison
- The context switches to the child
- The child runs thr_exit() and signals the parent, but the parent is not waiting yet.
- **The signal is lost**
- The parent deadlocks

PRODUCER / CONSUMER



PRODUCER / CONSUMER

- **Producer**
  - Produces items – consider the child matrix maker
  - Places them in a buffer
    - Example: the buffer is only 1 element (single array pointer)
- **Consumer**
  - Grabs data out of the buffer
  - In our example the parent thread receives dynamically generated matrices and performs an operation on them
    - Example: calculates average value of every element (integer)
- Multithreaded web server example
  - Http requests placed into work queue; threads process

PRODUCER / CONSUMER - 2

- Producer / Consumer is also known as **Bounded Buffer**
- Bounded buffer
  - Similar to piping output from one Linux process to another
  - grep pthread signal.c | wc –l
  - Synchronized access send output from grep to wc as it is produced
  - File stream

PUT/GET ROUTINES

- Buffer is a one element shared data structu re
- Producer "puts" data
- Consumer "gets" data
- Shared data structure requires synchronization

```
int buffer;
int count = 0;   // initially, empty

void put(int value) {
        assert(count == 0);
        count = 1;
        buffer = value;
}

int get() {
        assert(count == 1);
        count = 0;
        return buffer;
}
```


PRODUCER / CONSUMER - 3

- Producer adds data
- Consumer removes data (busy waiting)
- **Will this code work (spin locks) with 2-threads?**

1. Producer 2. Consumer

```
void *producer(void *arg) {
        int i;
        int loops = (int) arg;
        for (i = 0; i < loops; i++) {
                put(i);
        }
}

void *consumer(void *arg) {
        int i;
        while (1) {
                int tmp = get();
                printf("%d\n", tmp);
        }
}
```


PRODUCER / CONSUMER - 3

- The shared data structure needs synchronization!

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                // p1
        if (count == 1)                            // p2
            Pthread_cond_wait(&cond, &mutex);      // p3
        put(i);                                    // p4
        Pthread_cond_signal(&cond);                // p5
        Pthread_mutex_unlock(&mutex);              // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                // c1
```

Producer

PRODUCER/CONSUMER - 4

```
        if (count == 0)                            // c2
            Pthread_cond_wait(&cond, &mutex);      // c3
        int tmp = get();                           // c4
        Pthread_cond_signal(&cond);                // c5
        Pthread_mutex_unlock(&mutex);              // c6
        printf("%d\n", tmp);
    }
}
```

Consumer

- This code as-is works with just:
  - (1) Producer
  - (1) Consumer
- If we scale to (2+) consumer's it fails
  - How can it be fixed ?

EXECUTION TRACE

- Two threads

Legend
c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in …
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	… and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	**Oh oh! No data**

PRODUCER/CONSUMER SYNCHRONIZATION

- When producer threads awake, they do not check if there is any data in the buffer…
  - Need while, not if
- What if T_p puts a value, wakes T_{c1} whom consumes the value
- Then T_p has a value to put, but T_{c1}'s signal on &cond wakes T_{c2}
- There is nothing for T_{c2} consume, so T_{c2} sleeps
- T_{c1}, T_{c2}, and T_p all sleep forever
- T_{c1} needs to wake T_p to T_{c2}

EXECUTION TRACE - 2

Legend
c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke T_{c2}**

EXECUTION TRACE - 3

- T_{c2} runs, no data to consume

Legend
c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
…	…	…	…	…	…	…	*(cont.)*
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	**Everyone asleep …**

TWO CONDITIONS

- Add a second condition
  - One condition handles the producer
  - the other the consumer

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}
```


FINAL PRODUCER/CONSUMER

- Change buffer from int, to int buffer[MAX]
- Add indexing variables
- Typo: two variables named fill, need separate namespaces

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```

FINAL P/C - 2

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                  // p1
        while (count == MAX)                         // p2
            Pthread_cond_wait(&empty, &mutex);       // p3
        put(i);                                      // p4
        Pthread_cond_signal(&fill);                  // p5
        Pthread_mutex_unlock(&mutex);                // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                  // c1
        while (count == 0)                           // c2
            Pthread_cond_wait(&fill, &mutex);        // c3
        int tmp = get();                             // c4
```


COVERING CONDITIONS - 2

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...;                  // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);          //
    Pthread_mutex_unlock(&m);
}
```

Check available memory

Broadcast

FINAL P/C - 3

```
(Cont.)
            Pthread_cond_signal(&empty);             // c5
            Pthread_mutex_unlock(&mutex);            // c6
            printf("%d\n", tmp);
        }
    }
```

- Producer: only sleeps when buffer is full
- Consumer: only sleeps if buffers are empty

COVER CONDITIONS - 3

- Broad awakens all blocked threads requesting memory
- Will evaluate each request: (bytesLeft < size)
  - Reject: requests which cannot be fulfilled
    - *Insufficient memory*
  - Process all requests which **can** be fulfilled
    - with newly available memory
- Overhead
  - Many threads may be awoken which take no action

COVERING CONDITIONS

- Excellent use case for pthread_cond_broadcast
- Consider memory allocation:
  - What if a program deals with huge memory allocation/deallocation on the heap
  - Access to the heap must be managed with low memory resources
    PREVENT: Out of memory- - - queue requests until memory is free

QUESTIONS