


## TCSS 422: OPERATING SYSTEMS

### Condition Variables



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

## OBJECTIVES

- Condition variables
- Consumer/Producer
- Covering condition

February 8, 2017 TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma L9.4

## FEEDBACK

- Questions regarding program #1:
  - **execvp**
  - `execvp("ls", "ls", "-l", "/bin/??", (char *)NULL);`
  - Allows an indeterminate number of arguments.
  - 1st arg: command to run
  - Addtl args: `arg[0]-arg[n]`, NULL terminated
  - For `exec` w/o pointers:
  - `execvp(arg[0], arg[0], arg[1], arg[2], arg[3], ..., (char *)NULL);`
  - **execvp**
  - First arg is address of string (char array) of first command
  - Second argument is pointer to list of arguments
    - The first argument is the command

February 8, 2017 TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma L9.2

## 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 ...


February 8, 2017 TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma L9.5

## FEEDBACK - 2

- Program 2
  - Posted - Due
- Midterm: Monday February 13
  - CPU Scheduling (Virtualizing the CPU)
  - Chapters 4, 6, 7, 8, 9
  - Concurrency
  - Chapters 26, 27, 28, 29, 30, 32\*
    - \* - *deadlocks: common causes, how to avoid*

February 8, 2017 TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma L9.3

## CONDITION VARIABLES - 2



- Support a signaling mechanism to alert threads 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)

February 8, 2017 TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma L9.6

## 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()

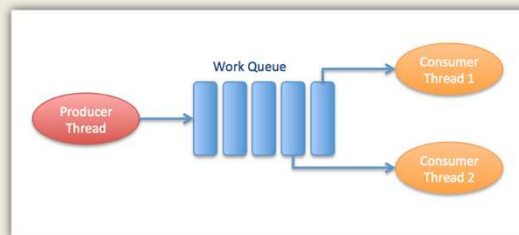
- Wakes up thread, awakening thread acquires lock

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.7

## PRODUCER / CONSUMER



February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.10

## MATRIX GENERATOR

Matrix generation example

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.8

## 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
- Our example: 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

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.11

## SUBTLE RACE CONDITION: WITHOUT A WHILE

```
1 void thr_exit() {
2   done = 1;
3   pthread_cond_signal(&c);
4 }
5
6 void thr_join() {
7   if (done == 0)
8     pthread_cond_wait(&c);
9 }
```

- 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

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.9

## 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:
  - sends output from grep → wc as it is produced
- File stream

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.12

## PUT/GET ROUTINES

- Buffer is a one element shared data structure (int)
- Producer "puts" data
- Consumer "gets" data
- Shared data structure requires synchronization

```

1  int buffer;
2  int count = 0; // initially, empty
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }

```

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.13

## PRODUCER/CONSUMER - 4

```

20     if (count == 0) // c2
21         Pthread_cond_wait(&cond, &mutex); // c3
22     int tmp = get(); // c4
23     Pthread_cond_signal(&cond); // c5
24     Pthread_mutex_unlock(&mutex); // c6
25     printf("kd\n", tmp);
26 }
27 }

```

- 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?

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.16

## PRODUCER / CONSUMER - 3

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

1. Producer 2. Consumer

```

1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     int i;
11     while (1) {
12         int tmp = get(i);
13         printf("kd\n", tmp);
14     }
15 }

```

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.14

## 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 <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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	
		Ready		Ready	p3	Running	1	Buffer now full
		Ready		Ready	p4	Running	1	T <sub>c1</sub> awoken
		Ready		Ready	p5	Running	1	
		Ready		Ready	p6	Running	1	
		Ready		Ready	p1	Running	1	
		Ready		Ready	p2	Running	1	
		Ready		Ready	p3	Sleep	1	Buffer full; sleep
		Ready		Ready	c1	Running	1	T <sub>c2</sub> sneaks in ...
		Ready		Ready	c2	Running	1	
		Ready		Ready	c3	Sleep	1	
		Ready		Ready	c4	Running	0	... and grabs data
		Ready		Ready	c5	Running	0	T <sub>p</sub> awoken
		Ready		Ready	c6	Running	0	
		Ready		Ready	c1	Ready	0	Oh oh! No data

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.17

## PRODUCER / CONSUMER - 3

- The shared data structure needs synchronization!

```

1  cond_t cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex); // p1
8          if (count == 1) // p2
9              Pthread_cond_wait(&cond, &mutex); // p3
10         put(i); // p4
11         Pthread_cond_signal(&cond); // p5
12         Pthread_mutex_unlock(&mutex); // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex); // c1

```

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.15

## 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<sub>p</sub> puts a value, wakes T<sub>c1</sub> whom consumes the value
- Then T<sub>p</sub> has a value to put, but T<sub>c1</sub>'s signal on &cond wakes T<sub>c2</sub>
- There is nothing for T<sub>c2</sub> consume, so T<sub>c2</sub> sleeps
- T<sub>c1</sub>, T<sub>c2</sub>, and T<sub>p</sub> all sleep forever
- T<sub>c1</sub> needs to wake T<sub>p</sub> to T<sub>c2</sub>

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.18

## 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 <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	State	Count	Comment
c1	Running				Ready	0	
c2	Running				Ready	0	
c3	Sleep				Ready	0	Nothing to get
		c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep	p1	Running	0	Nothing to get
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p3	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T <sub>c1</sub> 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 <sub>c1</sub> grabs data
c5	Running		Sleep		Sleep	0	Oops! Woke T <sub>c2</sub>

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.19

## FINAL PRODUCER/CONSUMER

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

```

1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }

```

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.22

## EXECUTION TRACE - 3

- T<sub>c2</sub> 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 <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 ...

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.20

## FINAL P/C - 2

```

1  while (t.empty(), fill);
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          pthread_mutex_lock(&mutex);          // p1
8          while (count == MAX)                 // p2
9              pthread_cond_wait(&empty, &mutex); // p3
10         put(i);
11         pthread_cond_signal(&fill);           // p4
12         pthread_mutex_unlock(&mutex);         // p5
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                    // c2
21             pthread_cond_wait(&fill, &mutex); // c3
22         int tmp = get(i);                     // c4

```

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.23

## TWO CONDITIONS

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

```

1  while (t.empty(), fill);
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          pthread_mutex_lock(&mutex);
8          while (count == 1)
9              pthread_cond_wait(&empty, &mutex);
10         put(i);
11         pthread_cond_signal(&fill);
12         pthread_mutex_unlock(&mutex);
13     }
14 }
15

```

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.21

## FINAL P/C - 3

```

(Cont.)
23  pthread_cond_signal(&empty);                // c5
24  pthread_mutex_unlock(&mutex);               // c6
25  printf("%d\n", tmp);
26  }
27 }

```

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

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.24

## COVERING CONDITIONS

- A condition that covers **all** cases (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 when memory is scarce

**PREVENT:** Out of memory:  
- queue requests until memory is free

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.25

## QUESTIONS



February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.28

## COVERING CONDITIONS - 2

```

1 // how many bytes of the heap are free?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // need lock and condition too
5 cond_t c;
6 mutex_t m;
7
8 void *
9 allocate(int size) {
10     pthread_mutex_lock(&m);
11     while (bytesLeft < size) {
12         pthread_cond_wait(&c, &m);
13         void *ptr = ...; // get mem from heap
14         bytesLeft += size;
15         pthread_mutex_unlock(&m);
16         return ptr;
17     }
18 }
19 void free(void *ptr, int size) {
20     pthread_mutex_lock(&m);
21     bytesLeft += size;
22     pthread_cond_signal(&c); // Broadcast
23     pthread_mutex_unlock(&m);
24 }
```

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.26

## COVER CONDITIONS - 3

- Broadcast awakens all blocked threads requesting memory
- Each thread evaluates if there's enough memory: (`bytesLeft < size`)
  - Reject: requests that cannot be fulfilled- go back to sleep
    - *Insufficient memory*
  - Run: requests which **can** be fulfilled
    - with newly available memory!
- **Overhead**
  - Many threads may be awoken which can't execute

February 8, 2017

TCSS422: Operating Systems [Winter 2017]  
Institute of Technology, University of Washington - Tacoma

L9.27