TCSS 422: OPERATING SYSTEMS

Condition Variables, Concurrency Problems

Wes J. Lloyd
School of Engineering and Technology,
University of Washington - Tacoma

November 7, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma

FEEDBACK FROM 11/5

- How does pthread_join() join thread values?

NAME
pthread_join - join with a terminated thread

SYNOPSIS
#include <pthread.h>
int pthread_join(pthread_t thread, **void **retval**);

2nd parameter provides a void ** pointer
Can return pointer to any user defined struct

November 7, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L12.2

OBJECTIVES

- Quiz 3 – Synchronized Array
- Multi-threaded Programming
- Chapter 30 – Condition Variables
- Chapter 32 – Concurrency Problems
- Memory Virtualization
- Chapters 13, 14, 15, 16….

November 7, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L12.3

FORK() COPY ON WRITE

- Processes and Threads share the code segment.
- From: https://en.wikipedia.org/wiki/Copy-on-write
- When fork() is called, a copy of all parent process pages is created, and loaded into a separate memory location by the OS for the child process.
- But this is not needed in certain cases.
- If a child executes an "exec" call or exits very soon after the fork(), there is no need to copy the parent process' pages.
- As an optimization, Linux uses a technique called **copy-on-write (COW).**

COPY ON WRITE - 2

- When the fork() occurs, parent process pages are *NOT* copied for the child process.
- Pages are shared between the parent and child.
- When a process (parent or child) modifies a memory page, a separate copy of the page is made for that process (parent or child) which performed the modification.
- This process uses the newly copied page rather than the shared one in future references.
- The other process (the one which did not modify the shared page) continues to use the original copy of the page (which is now no longer shared).
- This technique is called copy-on-write since the page is copied only when some process modifies to it.
- Binary C files are unmodified, with COW they are shared

CHAPTER 30 – CONDITION VARIABLES

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 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)

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

CONDITION VARIABLES - QUESTIONS

- **Why would we want to put waiting threads on a queue… why not use a stack?**
  - Queue (FIFO), Stack (LIFO)
  - Using condition variables eliminates busy waiting by putting threads to "sleep" and yielding the CPU.
- **Why do we want to not busily wait for the lock to become available?**
- A program has 10-threads, where 9 threads are waiting. The working thread finishes and broadcasts that the lock is available. **What happens next?**

MATRIX GENERATOR

Matrix generation example

Chapter 30
signal.c

MATRIX GENERATOR

- The main thread, and worker thread (generates matrices) share a single matrix pointer.
- What would happen if we don't use a condition variable to coordinate exchange of the lock?
- Let's try "nosignal.c"

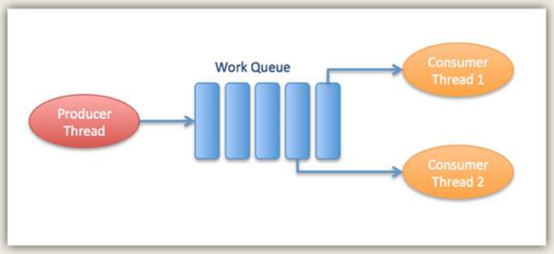
SUBTLE RACE CONDITION: WITHOUT A WHILE

```
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
  - 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

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

PUT/GET ROUTINES

- Buffer is a one element shared data structure (int)
- 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!

Producer

```c
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
```

November 7, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L12.19

QUESTIONS