Concurrency: Condition Variables

Ram Alagappan

CS 537, Spring 2020

03/03/2020

Announcements

Shivaram is out of town

Project 3 is due on Thursday 03/05 at 10 pm

Current module: concurrency

- so far: how to use, build locks
- today: how to order and coordinate – condition variables

Locks Recap

Why locks?

mutual exclusion – only one thread must execute critical section

How to build locks?

hardware support – test-and-set, compare-and-swap etc.

spin if not able to acquire lock

Queue locks

spinning can be inefficient (esp. on uniprocessors)

need OS support for queue locks

park()/unpark() in Solaris, futex in Linux

Metrics to evaluate locks: correctness, performance, fairness

Concurrency Objectives

Mutual exclusion (e.g., A and B don't run at same time)

- solved with locks

Ordering (e.g., B runs after A does something)

- solved with condition variables (this class) and semaphores (next class)

Ordering Example: Join

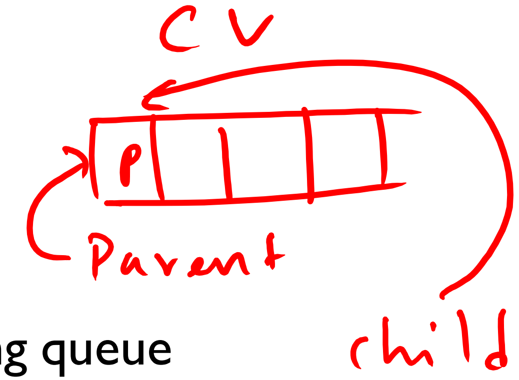
```
pthread_t p1;
Pthread_create(&p1, NULL, mythread, "A");
// join waits for the thread to finish
Pthread_join(p1, NULL);
printf("parent: end\n");
return 0;
```

parent

requirement: parent must wait for child thread to finish

how to implement join()? – cv

Condition Variables



Condition variables help in such situations

Condition variable: queue of waiting threads

Parent waits on the condition, putting itself on the waiting queue

– wait(CV, …)

Child sends signal to CV when it is done

– signal(CV, …)

Condition Variable Operations

wait(cond_t *cv, mutex_t *lock) — Parent call

hold

- assumes the lock is held when wait() is called — hold lock
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing — no-op

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // x
        Cond_wait(&c, &m);       // y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);          // a
        Cond_signal(&c);         // b
        Mutex_unlock(&m);        // c
}
```

Example schedule:

Parent releases m and also sleeps

Parent:	x	y				z
Child:			a	b	c	

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);        // x
        Cond_wait(&c, &m);     // y
        Mutex_unlock(&m);      // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);        // a
        Cond_signal(&c);       // b
        Mutex_unlock(&m);      // c
}
```

Can you spot a problem?

Example problematic schedule:

Parent:				x	y
Child:	a	b	c		

Stuck!

Rule of Thumb 1

Keep state in addition to the condition variable

CVs are used to signal threads when state changes

If state is already as needed, thread doesn't wait for a signal

Join example

- a state variable called done, denoting whether child is done or not
- initially false
- child signals when it sets done to true
- parent checks if done is true; if yes, doesn't wait

Join Implementation: Attempt 2

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0)           // x
            Cond_wait(&c, &m);  // y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        done = 1;           // a
        Cond_signal(&c); // b
}
```

Example schedule:

Parent:			w	x	y	z
Child:	a	b				

Join Implementation: Attempt 2

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
            Cond_wait(&c, &m);      // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        done = 1;           // a
        Cond_signal(&c);    // b
}
```

Can you spot the problem?

Note: no mutex in child

Example problematic schedule:

Parent:	w	x			y Stuck!
Child:			a	b	parent wakes up.

Join Implementation: Correct

Parent:

```
void thread_join() {
        Mutex_lock(&m);           // w
        if (done == 0)            // x
            Cond_wait(&c, &m); // y
        Mutex_unlock(&m);          // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);          // a
        done = 1;                // b
        Cond_signal(&c);         // c
        Mutex_unlock(&m);        // d
}
```

Parent:	w	x	y					z
Child:				a	b	c	d	

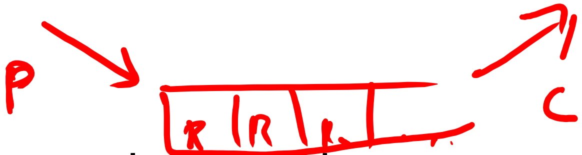
Rule of thumb 2: use mutex to ensure no race between interacting with state and wait/signal

Producer/Consumer Problem

A common paradigm, also called the bounded-buffer problem

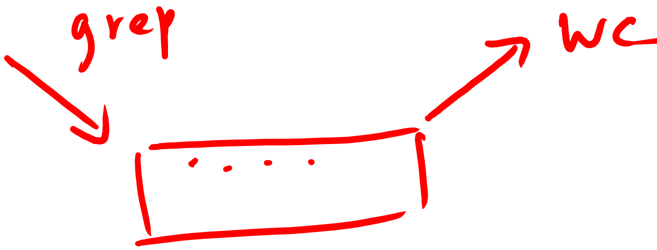
Producer/consumer problems are frequent in systems (e.g. web servers)

- producer puts requests in a queue
- consumers picks them up and processes



Producer and consumers need to access a shared buffer in a coordinated fashion

Example: UNIX Pipes



Pipe output of one process into another process

e.g., grep foo bar.txt | wc -l

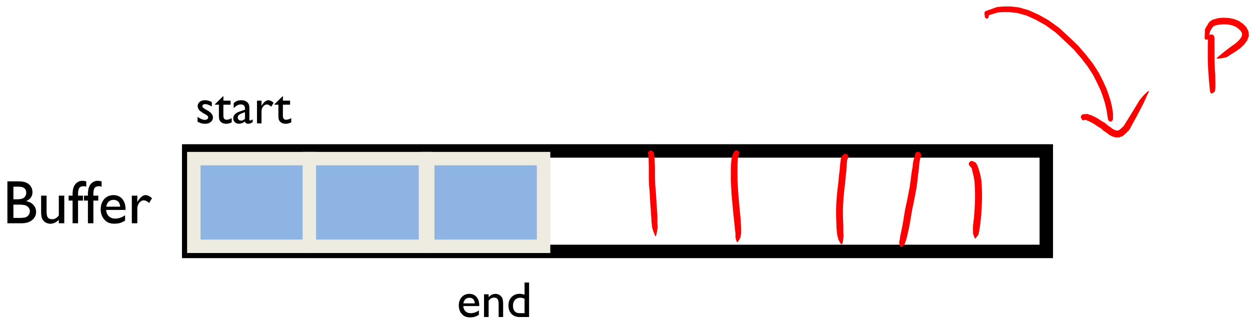
grep is the producer, wc is the consumer

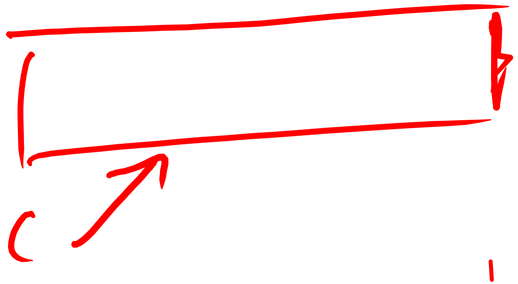
The finite-sized pipe (internally maintained by OS) is the shared buffer

Producer (grep) adds data to the buffer

Reader (wc) removes data from the buffer for processing

Accessing the Shared Buffer Correctly





Reads/writes to buffer require locking

When buffers are full, writers must wait

When buffers are empty, readers must wait

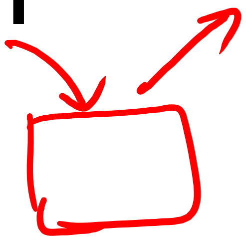
Solving Producer/Consumer Problem

Use lock for mutual exclusion

Use condition variables to:

- make producers wait when buffers are full
- make consumers wait when there is nothing to consume

Produce/Consumer Example



Start with easy case:

- one producer thread
- one consumer thread
- one shared buffer slot to fill/consume (max = 1)

State: numfull = number of buffer slots currently filled

0, 1

Solution assuming one P and one C

max=1

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if(numfull == max) // p2
            Cond_wait(&cond, &m); // p3
        do_fill(i); // p4
        Cond_signal(&cond); // p5
        Mutex_unlock(&m); // p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m); // c1
        if(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf(“%d\n”, tmp); // c7
    }
}
```

C_1 C_2 C_3 P_1 P_4 P_5 P_6 C_4 C_5 C_6 C_7

What about 2 consumers?

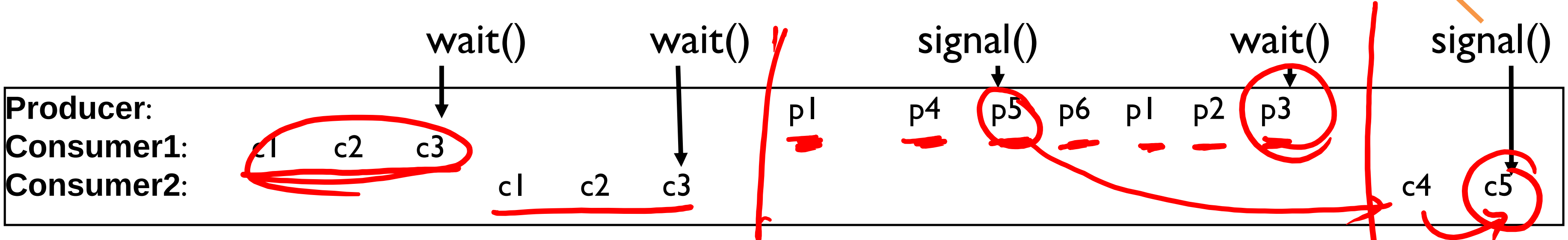
Can you find a problematic schedule with 2 consumers (still 1 producer)?

Hint: think about to which thread a signal from a consumer is delivered

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if(numfull == max) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);   // c1
        if(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```

if C1 is woken, it will read empty buffer



How to wake the right thread?

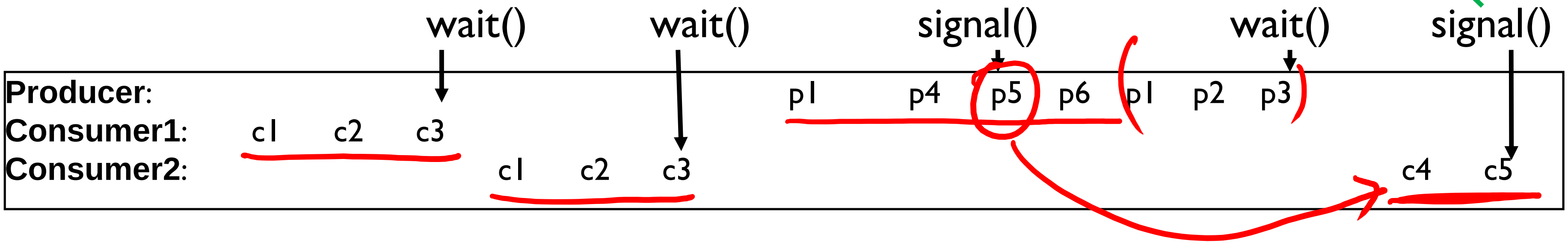
Use two condition variables

Producer/Consumer: Two CVs

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i);  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

only producer is signaled



Producer/Consumer: Two CVs

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i);  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m); // c1
        if (numfull == 0) // c2
            Cond_wait(&fill, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&empty); // c5
        Mutex_unlock(&m); // c6
    }
}
```

signal arrives late!

Is there a problem, still?

Producer:				p1	p4	p5	p6					
Consumer1:	c1	c2	c3									c4 BUG!
Consumer2:								c1	c4	c5	c6	

new consumer sneaks in

Producer/Consumer: Two CVs and WHILE

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i);   // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m); // c1
        while (numfull == 0) // c2
            Cond_wait(&fill, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&empty); // c5
        Mutex_unlock(&m); // c6
    }
}
```

signal arrives late…
but that's ok!

Correct behavior

Producer:				p1	p4	p5	p6								
Consumer1:	c1	c2	c3											c2	c3
Consumer2:										c1	c4	c5	c6		

Semantics are Important

Mesa semantics – not necessary that only the woken thread will run next; a newly arriving thread may run

A signal is just a hint that the state has changed

Need to recheck state using a while

Hoare semantics – the woken thread will run immediately after woken up, a newly arriving thread cannot sneak in

Most (if not all) systems use the Mesa semantics; thus, always good to use while

Rule of Thumb 3

Whenever a lock is acquired, recheck assumptions about state!

Another thread could grab lock in between signal and wakeup from wait

Note that some libraries also have "spurious wakeups" (may wake multiple waiting threads at signal or at any time)

Summary: rules of thumb for CVs

1. Keep state in addition to CVs
2. Always do wait/signal with lock held
3. Whenever thread wakes from waiting, recheck state

① Join – 'done', P/c – mindful

② wait – must hold the lock
signal – always good by / must hold the lock.

③ Signal – hints – Mesa
recheck state using WHILE

Apache Webserver Hang Bug

```
// global state
pthread_mutex_t mutex; pthread_cond_t cond; ServerSocket sock; Bool stopped=FALSE;
```

```
void run ( ){
  ...
  while (!stopped) {
    sock.accept (...);
/*accept returns when
it receives a request from client
or a thread executes sock.cancel
during the sock.accept ( ) */
    ...
  }
  pthread_mutex_lock(&mutex);
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

```
void DoStop ( ){
  sock.cancel ( );
  stopped=TRUE;
  pthread_mutex_lock(&mutex);
  pthread_cond_wait(&cond,&mutex);
  pthread_mutex_unlock(&mutex);
  ...
}
```

Can you find the bug?

Which rule was broken?

Covering Conditions

Example: multi-threaded memory allocator

```
int bytesLeft = MAX_HEAP_SIZE;
cond_t c; mutex_t m;

void* allocate(int size){
 Pthread_mutex_lock(&m);
 while (bytesLeft < size)
   Pthread_cond_wait(&c, &m);
 void* ptr = ...;
 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);
}
```

avail = 0; allocate(100); allocate(10); free(50); which to wake?