Condition Variables and Semaphore

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

Copyright © 2024

Topics

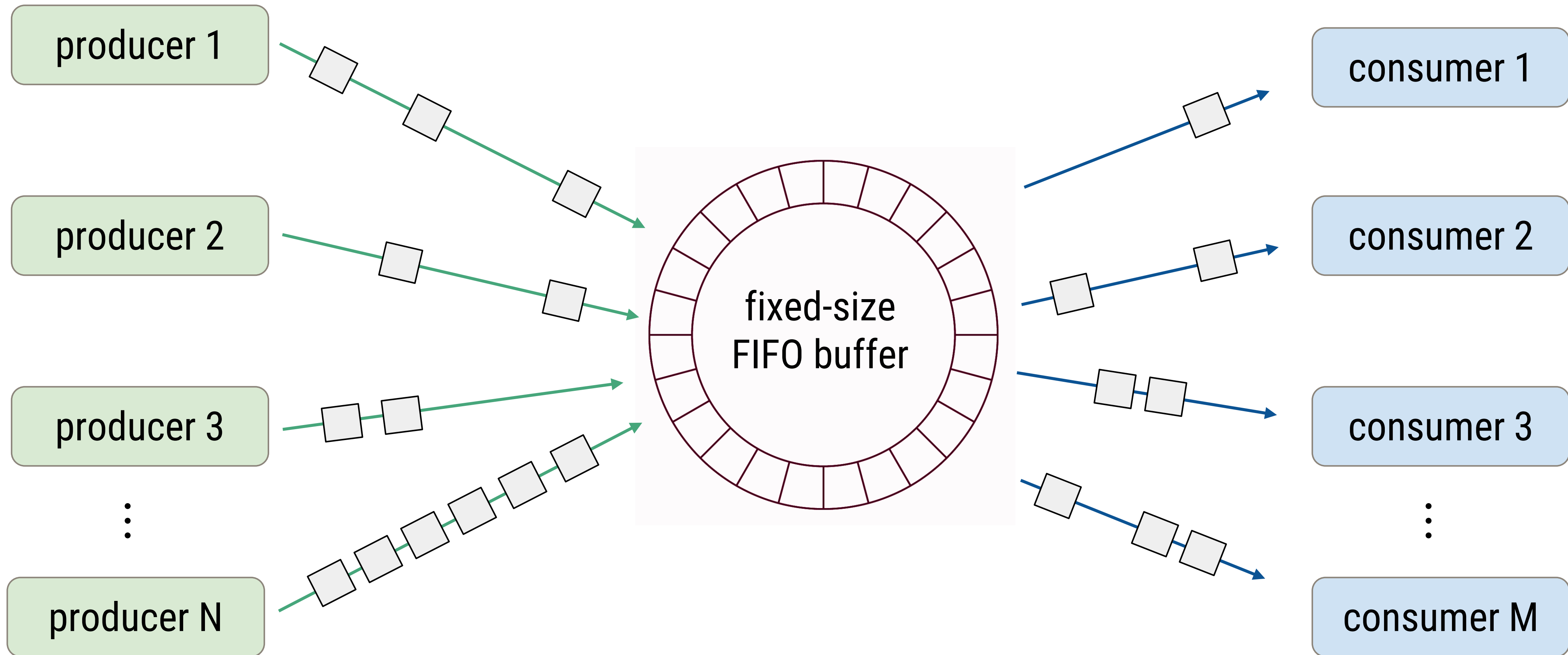
- producer-consumer problem
- mutexes and condition variables
- semaphores

UNIVERSITY OF CALGARY

Producer/Consumer Problem

UNIVERSITY OF CALGARY

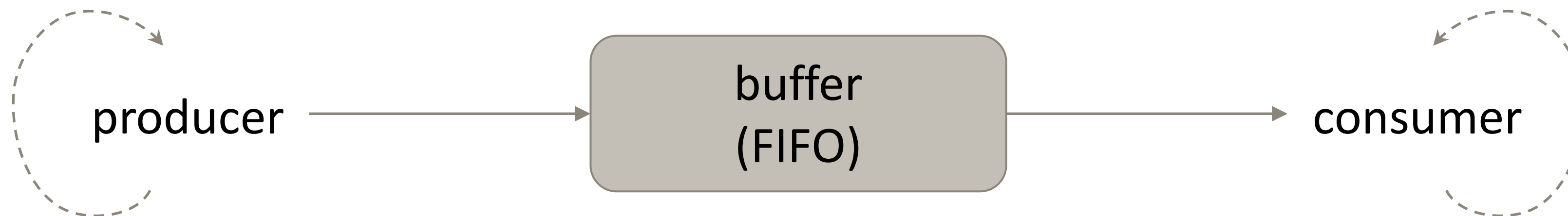
Producer-consumer problem



UNIVERSITY OF CALGARY

The producer-consumer problem

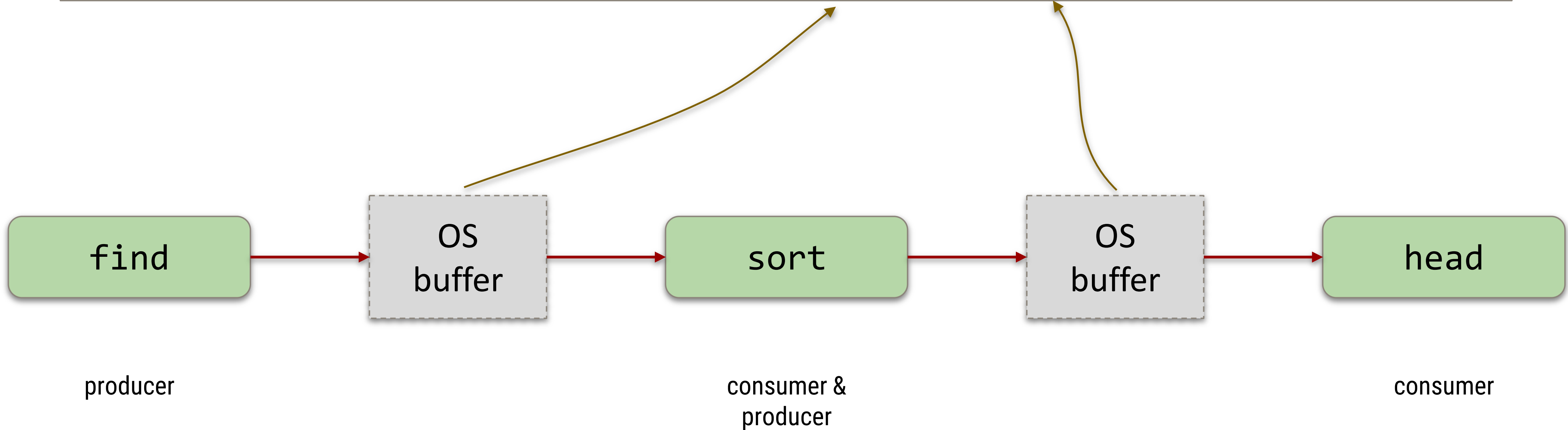
- simplest case: one consumer and one producer processes/threads
- the two processes or threads share a **fixed-size buffer**, used as a queue
- producer puts data into buffer, must wait if buffer full
- consumer takes data out of the buffer, must wait if buffer empty
- both could be producing and consuming at different rates



Unix Pipes

UNIVERSITY OF CALGARY

UNIX pipes

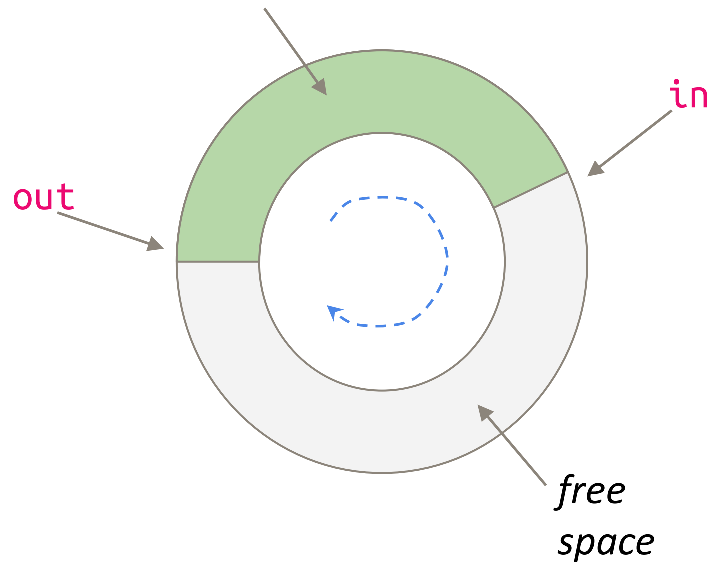


UNIVERSITY OF CALGARY

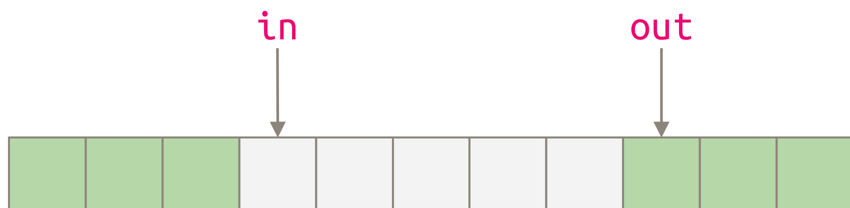
Circular Buffer

Circular buffer

Concept:



Implementation:



- common way to implement fixed-size queue

```
typedef struct { ... } item;
item buffer[BUFF_SIZE];
int in = 0;     // next free position
int out = 0;    // next filled position
int count = 0; // number of items in buffer
```

- buffer is empty when:
  `in == out` or when `count == 0`
- buffer is full when:
  `(in + 1) % BUFF_SIZE == out`
  or when
  `count == BUFF_SIZE`

count is not really needed since
`count = (in-out+BS) % BS`
but we'll use it anyways

UNIVERSITY OF CALGARY

Implementations

UNIVERSITY OF CALGARY

Possible implementation - with race condition

Thread 1 – producer

```
while(1) {
  item = produceItem();
  // wait while buffer is full
  while((in+1) % BUFF_SIZE == out) {;}
  // insert item into buffer
  buffer[in] = item;
  in = (in + 1) % BUFF_SIZE;
  count ++;
}
```

Thread 2 – consumer

```
while(1) {
  // wait while buffer is empty
  while( in == out) {;}
  // remove item from buffer
  item = buffer[out];
  out = (out+1) % BUFF_SIZE;
  count --;
  consumeItem(item);
}
```

Can you spot the race condition?

UNIVERSITY OF CALGARY

Possible implementation - with race condition

Thread 1 – producer

```
while(1) {
  item = produceItem();
  // wait while buffer is full
  while((in+1) % BUFF_SIZE == out) {;}
  // insert item into buffer
  buffer[in] = 
  in = (in + 1)
  count ++;
}
```

```
reg1   = count
reg1   = reg1 + 1
count = reg1
```

Thread 2 – consumer

```
while(1) {
  // wait while buffer is empty
  while( in == out) {;}
  // remove item from buffer
  item = buffer[out];
  out = (out+1)
  count --;
  consumeItem(i
}
```

```
reg2   = count
reg2   = reg2 - 1
count = reg2
```

Possible execution sequence (starting eg. with count=5):

```
T1: reg1   = count        // count=5
T2: reg2   = count        // count=5
T2: reg2   = reg2 - 1     // count=5
T2: count = reg2          // count=4
T1: reg1   = reg1 + 1     // count=4
T1: count = reg1          // count=6
```


UNIVERSITY OF CALGARY

Possible implementation - with a mutex

Thread 1 – producer

```
while(1) {
  item = produceItem();
  // wait while buffer is full
  while((in+1) % BUFF_SIZE == out) {;}
  // insert item into buffer
  buffer[in] = item;
  in = (in + 1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count ++; // critical section
  pthread_mutex_unlock(& mut);
}
```

Thread 2 – consumer

```
while(1) {
  // wait while buffer is empty
  while( in == out) {;}
  // remove item from buffer
  item = buffer[out];
  out = (out+1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count --; // critical section
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

Did we fix all problems?

UNIVERSITY OF CALGARY

Possible implementation - with a mutex

Thread 1 – producer

```
while(1) {
  item = produceItem();
  // wait while buffer is full
  while((in+1) % BUFF_SIZE == out) {;}
  // insert item into buffer
  buffer[in] = item;
  in = (in + 1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count ++; // critical section
  pthread_mutex_unlock(& mut);
}
```

Thread 2 – consumer

```
while(1) {
  // wait while buffer is empty
  while( in == out) {;}
  // remove item from buffer
  item = buffer[out];
  out = (out+1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count --; // critical section
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

- this solution would work, but only for **one** producer thread and **one** consumer thread
- with multiple producers and/or multiple consumers we would have race condition[s]
- another important problem: busy wait!

UNIVERSITY OF CALGARY

Possible implementation – with a mutex and deadlock

Thread 1 – producer

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    while((in+1) % BUFF_SIZE == out) {;}
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_unlock(& mut);
}
```

Thread 2 – consumer

```
while(1) {
  pthread_mutex_lock(& mut);
    while( in == out) {;}
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

Not a good solution – there is now a deadlock possibility… can you find it?

UNIVERSITY OF CALGARY

Possible implementation – with a mutex and deadlock

Thread 1 – producer

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    while((in+1) % BUFF_SIZE == out) {;}
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_unlock(& mut);
}
```

Thread 2 – consumer

```
while(1) {
  pthread_mutex_lock(& mut);
    while( in == out) {;}
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

- one possible deadlock: buffer is empty, and consumer enters its critical section …
- consumer spins in a while loop inside CS, while producer is blocked on trying to acquire mutex
- another deadlock: can you find it?

UNIVERSITY OF CALGARY

Possible implementation – with a mutex and deadlock

Thread 1 – producer

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    while((in+1) % BUFF_SIZE == out) {;}
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_unlock(& mut);
}
```

Thread 2 – consumer

```
while(1) {
  pthread_mutex_lock(& mut);
    while( in == out) {;}
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```


UNIVERSITY OF CALGARY

Condition Variables

UNIVERSITY OF CALGARY

Condition variables (CVs)

- condition variables are another type of synchronization primitive, used with mutexes
- perfect for implementing critical sections with loops that wait for some 'condition' to happen

```
// critical section protected with mutex m
lock(m)
  …
  while( ! some_condition() ) { /* do nothing ???  */ }
  …
unlock(m)
```

some_condition()
represents arbitrary user code, e.g. that checks status of some global variable(s)

- in many cases, while inside the "do nothing" loop, it would be safe to let another thread run its critical section

UNIVERSITY OF CALGARY

Condition variables (CVs)

- would this work?

```
// critical section protected with mutex m:
m.lock();
  // some code requiring mutex to be locked
  while( ! some_condition() ) {
    // assume that while we are inside the loop, it would be safe
    // to let some other thread run…
    m.unlock();
    std::this_thread::sleep_for(std::chrono::milliseconds(x));
    m.lock();
  }
  // more code requiring mutex to be locked
unlock(m)
```

- it would "kind of" work, but what is the right amount of sleep **x** ?
- it would work much better if we could sleep until the other thread wakes us up…

UNIVERSITY OF CALGARY

Common pattern for using CVs

- a thread locks a mutex and enters its critical section
- while still inside CS, the thread needs to wait for some condition to become true
- but the condition can only become true by allowing some other thread to lock the mutex
- to facilitate this, the thread calls wait(cv) , which releases the mutex and puts thread to sleep

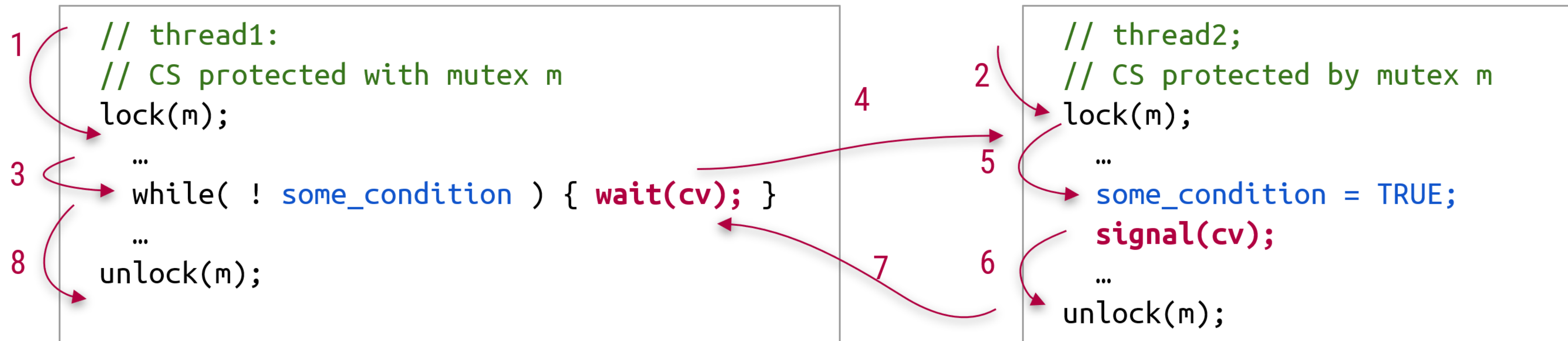
```
// thread1:
// CS protected with mutex m
lock(m);
  …
  while( ! some_condition ) { wait(cv); }
  …
unlock(m);
```

in this example
some_condition
is a global variable

- now some other thread can lock the mutex and execute code that will satisfy the condition
- yes, two threads are now in CS, but one of them is sleeping and cannot cause damage…
- when the other thread releases mutex, the mutex in the first thread is automatically re-locked

Condition variables

- eventually some other thread
  - locks the mutex (optional)
  - changes some state that will satisfy the condition
  - notifies the waiting thread via the condition variable, via signal(cv)
  - releases mutex (optional)
- the waiting thread then wakes up, and acquires the mutex back automatically



Condition Variables in pthreads

UNIVERSITY OF CALGARY

Condition variables with pthreads

```
pthread_mutex_t mutex; // mutex
pthread_cond_t cond;   // condition variable
pthread_cond_wait(&cond, &mutex);
```

- unlocks mutex and puts the calling thread to sleep,
  until some other thread wakes it up via `pthread_cond_signal(&cond)`
- after waking up, the mutex is automatically re-acquired
- after returning, the condition must be rechecked !!! (spurious wakeups)

```
pthread_cond_signal(&cond);
```

- wakes up one thread waiting on `cond`
- if no threads waiting on `cond`, the signal is lost
- must be paired with `pthread_mutex_unlock()` if the blocked thread uses the same mutex

UNIVERSITY OF CALGARY

Condition variables

`pthread_cond_init(& cond, & attr)`

- initializes condition variable

`pthread_cond_destroy(& cond)`

- destroys a condition variable

`pthread_cond_broadcast(& cond)`

- wakes up all threads waiting on the condition

UNIVERSITY OF CALGARY

Condition Variables Examples

UNIVERSITY OF CALGARY

Condition variable example (C)

- thread 1 - decrementing counter, but never below 0
- thread 2 - incrementing counter

Thread 1

```
while(1) {
  pthread_mutex_lock(&mutex);
    while(counter == 0) {
      // busy wait & deadlock here
    }
    counter --; // CS
  pthread_mutex_unlock(&mutex);
}
```

Thread 2

```
while(1) {
  pthread_mutex_lock(&mutex);
    counter ++; // CS

  pthread_mutex_unlock(&mutex);
}
```

- this code has busy wait, and nearly guaranteed deadlock
- it is trivial to fix the above with a condition variable

Condition variable example (pthreads)

- thread 1 - decrementing counter, but never below 0
- thread 2 - incrementing counter

Thread 1

```
while(1) {
  pthread_mutex_lock(&mutex);
    while(counter == 0) {
      pthread_cond_wait(&cond, &mutex);
    }
    counter --; // CS
  pthread_mutex_unlock(&mutex);
}
```

Thread 2

```
while(1) {
  pthread_mutex_lock(&mutex);
    counter ++; // CS
    pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

- no deadlocks
- no busy waiting

UNIVERSITY OF CALGARY

Condition variable example (C++)

Thread 1

```
// globals
std::mutex m;
std::condition_variable cv;

while(1) {
  m.lock();
    while( counter == 0) {
      cv.wait(m);
    }
    counter --; // CS
  m.unlock();
}
```

Thread 2

```
while(1) {
  m.lock();
    counter ++; // CS
    cv.notify_one();
  m.unlock();
}
```

Condition variable example (with C++ unique lock)

Thread 1

```
// globals
std::mutex m;
std::condition_variable cv;

while(1) {
  std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{
      return counter != 0;
    });
    counter --; // CS
}
```

Thread 2

```
while(1) {
  m.lock();
    counter ++; // CS
    cv.notify_one();
  m.unlock();
}
```


UNIVERSITY OF CALGARY

Let's fix the producer/consumer deadlock

- recall the deadlock due to one thread stuck in a loop inside CS (after locking mutex), while other thread has no chance to run its CS to allow the other thread to exit the loop

Producer thread

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    // wait while buffer is full
    while((in+1) % BUFF_SIZE == out) {;}
    // insert item into buffer
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_unlock(& mut);
}
```

Consumer thread

```
while(1) {
  pthread_mutex_lock(& mut);
    // wait while buffer is empty
    while( in == out) {;}
    // remove item from buffer
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

Fixed consumer/producer with condition variables

```
pthread_mutex_t mut;
pthread_cond_t full, empty;
```

Producer thread

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    while((in+1) % BUFF_SIZE == out) {
      pthread_cond_wait(& full,& mut);
    }
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_cond_signal(& empty);
  pthread_mutex_unlock(& mut);
}
```

Consumer thread

```
while(1) {
  pthread_mutex_lock(& mut);
    while( in == out) {
      pthread_cond_wait(& empty,& mut);
    }
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_cond_signal(& full);
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```


UNIVERSITY OF CALGARY

Semaphores

UNIVERSITY OF CALGARY

Semaphore

- another synchronization primitive
- some similarity to mutex
- you can think of mutex as a special boolean variable shared by all threads
  - with 3 special operations: initialize, lock and unlock
- you can think of semaphore as a special integer variable shared by all threads
  - with 3 special operations: initialize, increment and decrement

UNIVERSITY OF CALGARY

Semaphore

- semaphore has thread safe operations:
  1. initialization
     - can be initialized with any value (0 … max)
  2. decrement
     - reduce semaphore by 1
     - blocks the calling thread if value goes below 0

     down(s), **wait**(s) or **sem_wait**(s)
  3. increment
     - increase value by 1
     - and possibly unblock another blocked process

     up(s), **signal**(s) or **sem_post**(s)

UNIVERSITY OF CALGARY

Semaphore

- can be used to protect critical sections, similar to how a mutex would be used

```
initialize semaphore
  s(1);
…
sem_wait(s);
  // critical section
sem_post(s);
```

- similar to mutex, each semaphore maintains a set of processes blocked on the semaphore
- but a semaphore can be unlocked by **any** thread
- as opposed to mutex, where a locking/unlocking must be done by the **same** thread

UNIVERSITY OF CALGARY

Semaphore

- pseudocode implementation, using busy-waiting:

```
sem_wait(s) {
    while (s == 0) {;}
    s --;
}
```

```
sem_post(s) {
    s ++;
}
```

but the bodies must execute atomically

UNIVERSITY OF CALGARY

POSIX Semaphores

UNIVERSITY OF CALGARY

POSIX semaphores

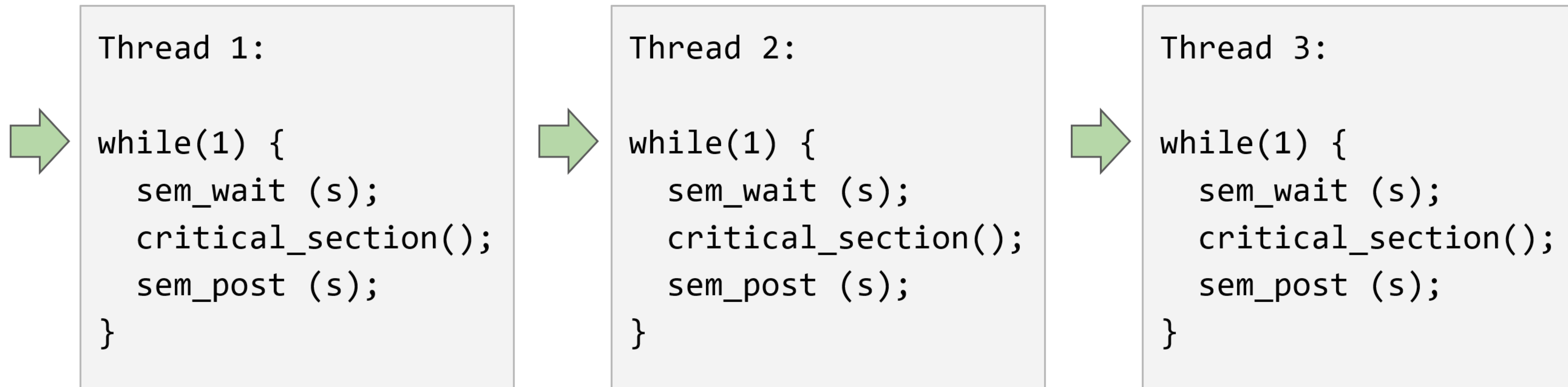
- `int sem_init (sem_t *sem, int pshared, unsigned int value)`
  initializes semaphore to `value`
- `int sem_destroy (sem_t * sem)`
  destroys the semaphore, fails if some threads are waiting on it
- `int sem_wait (sem_t * sem)`
  suspends the calling thread until the semaphore is non-zero, then atomically decreases the semaphore count
- `int sem_post (sem_t * sem)`
  atomically increases the semaphore, never blocks, may unblock blocked threads
- `int sem_getvalue (sem_t * sem, int * sval)`
  returns the value of semaphore via `sval`
- `int sem_trywait (sem_t * sem)`
  non blocking version of `sem_wait()`

UNIVERSITY OF CALGARY

Example Semaphores

Example with semaphore = 2

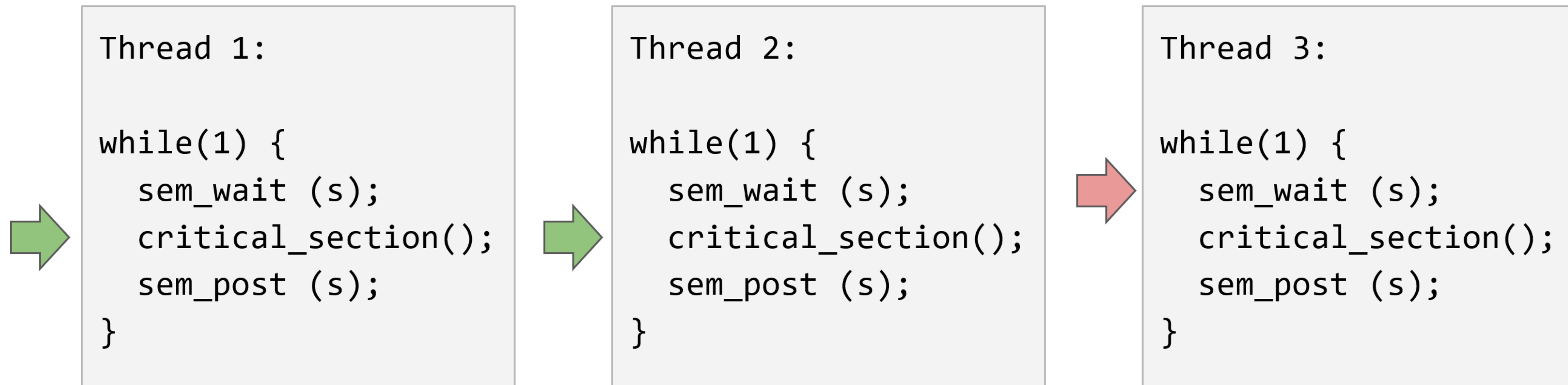
```
sem_t s;
sem_init( & s, 0, 2); // initialize semaphore to 2
```



- suppose 3 threads try to enter their CSs protected by a semaphore

Example with semaphore = 2

```
sem_t s;
sem_init( & s, 0, 2); // initialize semaphore to 2
```

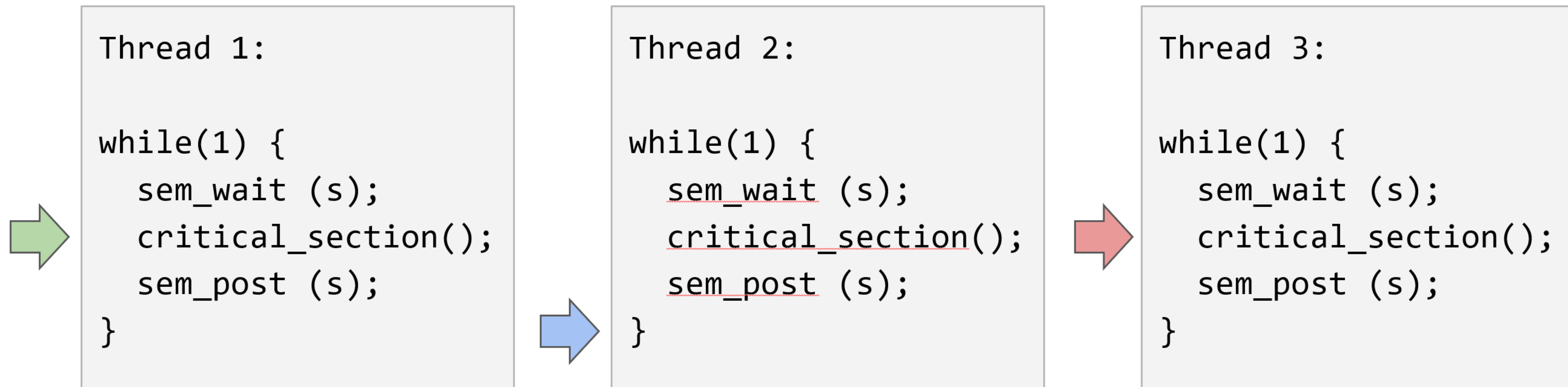


- two threads will enter their CS simultaneously
- the other thread will be blocked

UNIVERSITY OF CALGARY

Example with semaphore = 2

```
sem_t s;
sem_init( & s, 0, 2); // initialize semaphore to 2
```



- as soon as one thread leaves CS, the last thread will be allowed to enter its CS

UNIVERSITY OF CALGARY

Book!

UNIVERSITY OF CALGARY

The Little Book of Semaphores

- free book !!!
- http://greenteapress.com/wp/semaphores/

UNIVERSITY OF CALGARY

Binary Semaphore

UNIVERSITY OF CALGARY

Binary semaphore

- special type of semaphore with value either 0 or 1
- possible implementations in pseudocode:
-
```
sem_wait(s) {
    while (s == 0) {;}
    s = 0;
}
```

```
sem_post(s) {
    s = 1;
}
```

  where the bodies are executed **atomically**
- **atomic operation** is an operation that appears to execute instantaneously with respect to the rest of the system, e.g. cannot be interrupted by signals, threads, interrupts, ...

UNIVERSITY OF CALGARY

Dining Philosophers with Semaphores

Dining philosophers with semaphores

```
#define N 5         /* number of philosophers */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1   /* philosopher is trying to get forks */
#define EATING 2   /* philosopher is eating */

int state[N]; /* each philosophers is either THINKING, HUNGRY or EATING,
initialized to THINKING */
sem_t cs_m;    /* semaphore for the critical section, shared by all
philosophers, initialized to 1 */
sem_t p_m[N]; /* one semaphore per philosopher, initialized to 0 */

int left(int i) { return (i+N-1) % N; }
int right(int i) { return (i+1) % N; }

void philosopher(int i) /* to be executed by different threads */
{
  while (1) {
    think( );       /* philosopher is thinking */
    take_forks(i); /* acquire two forks or block */
    eat( );         /* yum-yum, spaghetti */
    put_forks(i);  /* put both forks back on table */
  }
}
```


UNIVERSITY OF CALGARY

Dining philosophers with semaphores

```
void take_forks(int i) {
  down(& cs_m);            /* enter critical region */
    state[i] = HUNGRY;     /* record fact that philosopher i is hungry */
    test _fork(i);               /* try to acquire 2 forks */
  up(& cs_m);              /* exit critical region */
  down(& p_m[i]);          /* block if forks were not acquired */
}

void put_forks(int i) {        /* i: philosopher number, from 0 to N-1 */
  down(& cs_m);            /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test_fork(left(i));          /* see if left neighbour can now eat */
    test _fork(right(i));         /* see if right neighbour can now eat */
  up(& cs_m);              /* exit critical region */
}

void test_fork(int i) {                /* i: philosopher number, from 0 to N-1 */
  if (state[i] == HUNGRY && state[left(i)] != EATING && state[right(i)] !=
EATING) {
    state[i] = EATING;     /* only start eating if hungry, and neighbours not
eating */
    up(& p_m[i]);
  }
}
```


UNIVERSITY OF CALGARY

Common Mistake

Common mistake with semaphores

Thread 1:

```
for(i=0; i<7; i++)
  sem_wait(S);

// critical section

for(i=0; i<7; i++)
  sem_post(S);
...
```

Thread 2:

```
for(i=0; i<6; i++)
  sem_wait(S);

// critical section

for(i=0; i<6; i++)
  sem_post(S);
...
```

- order of operations:
  - thread 1 requests 6 resources, then scheduler switches to thread 2
  - thread 2 requests 4 resources, exhausting all available resources
  - both threads are stuck → **deadlock**

UNIVERSITY OF CALGARY

Common mistake with semaphores

- Scenario: managing a pool of N resources
  - a counting semaphore S is used to keep track of the # of available resources
  - initialization: `sem_init(S,0,N)`
  - each process may need $K_i \leq N$ resources at a time
  - you might be tempted to accomplish this by K_i consecutive invocations of `sem_wait(S)`
  - Can you see the problem?

- Example:
  - 10 resources, thread 1 needs 7 resources and thread 2 needs 6 resources
  - depending on scheduling, we may get a **deadlock**

Semaphores vs. Condition Variables

UNIVERSITY OF CALGARY

Semaphores vs. condition variables

- `sem_post()` compared to `cv_signal()`:
  - `cv_signal()` is lost (has no effect) if no thread is waiting
  - `sem_post()` increments the semaphore always, and possibly wakes up a thread

- `sem_wait()` compared to `pthread_cond_wait()`:
  - `pthread_cond_wait()` always blocks
  - `sem_wait()` checks the value of the semaphore, and may or may not block

Semaphore Psuedocode

UNIVERSITY OF CALGARY

Semaphore pseudocode, with a queue & blocking

- `S->list` is a list of processes/threads
- a process can `block()` itself
- a process can be unblocked by `wakeup()`
- `getpid()` returns process or thread ID

```
typedef struct {
    int value;
    struct process *list;
} semaphore;

void sem_wait(semaphore *S) {
    S->value --;
    if (S->value < 0) {
        S->list->push(getpid());
        block();
    }
}

void sem_post(semaphore *S) {
    S->value ++;
    if (S->value <= 0) {
        pid = S->list->pop();
        wakeup(pid);
    }
}
```


RSITY OF
GARY

Semaphore implemented with mutexes and cond. vars.

```
struct sem {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int count = 0;
};

void sem_wait(sem * s) {
  lock(s->mutex);
    while(s->count == 0)
      wait(s->mutex,s->cond);
    s->count --;
  unlock(s->mutex);
}
```

```
void sem_post(sem * s) {
  lock(s->mutex);
    s->count ++;
    signal(s->cond);
  unlock(s->mutex);
}
```


UNIVERSITY OF CALGARY

It’s Hard!

It is tricky

- concurrent programming can be more difficult than coding in assembly
- any error with semaphores or mutexes will potentially result in race conditions, deadlocks, and other forms of unpredictable and irreproducible behaviour
- one subtle error and everything comes to a grinding halt
- for example:

Violate mutual exclusivity:

```
sem_post(sem);
// critical section
sem_wait(sem);
```

Deadlock:

```
lock(mutex);
// critical section
lock(mutex);
```


UNIVERSITY OF CALGARY

Review

Review

- Race conditions are not a problem among processes, only among threads.
  True or False?
- What is the main difference between pthread_cond_signal() for mutex and sem_post() for semaphore?
- A mutex is identical to binary semaphore.
  True or False?
- What does the value of the semaphore tell you?
- Define atomic operation.

UNIVERSITY OF CALGARY

Onward to … synchronizations mechanisms

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/