Condition Variables
Semaphores

Concurrency Tools and Problems

- Concurrency Tools (Primitives)
  - Locks, Condition Variables, Semaphores
- Problems
  - Mutual exclusion
    - Threads A and B run separately in a critical region
    - Solved with locks
  - Ordering
    - Thread B runs after thread A (or A runs after B)
    - Solved with condition variables and semaphores

Barrier Synchronization (ordering) Problem (part of Lab Multithreading)

- A group of threads where all threads must stop computing at a the barrier and wait until all of the group has arrived at the barrier

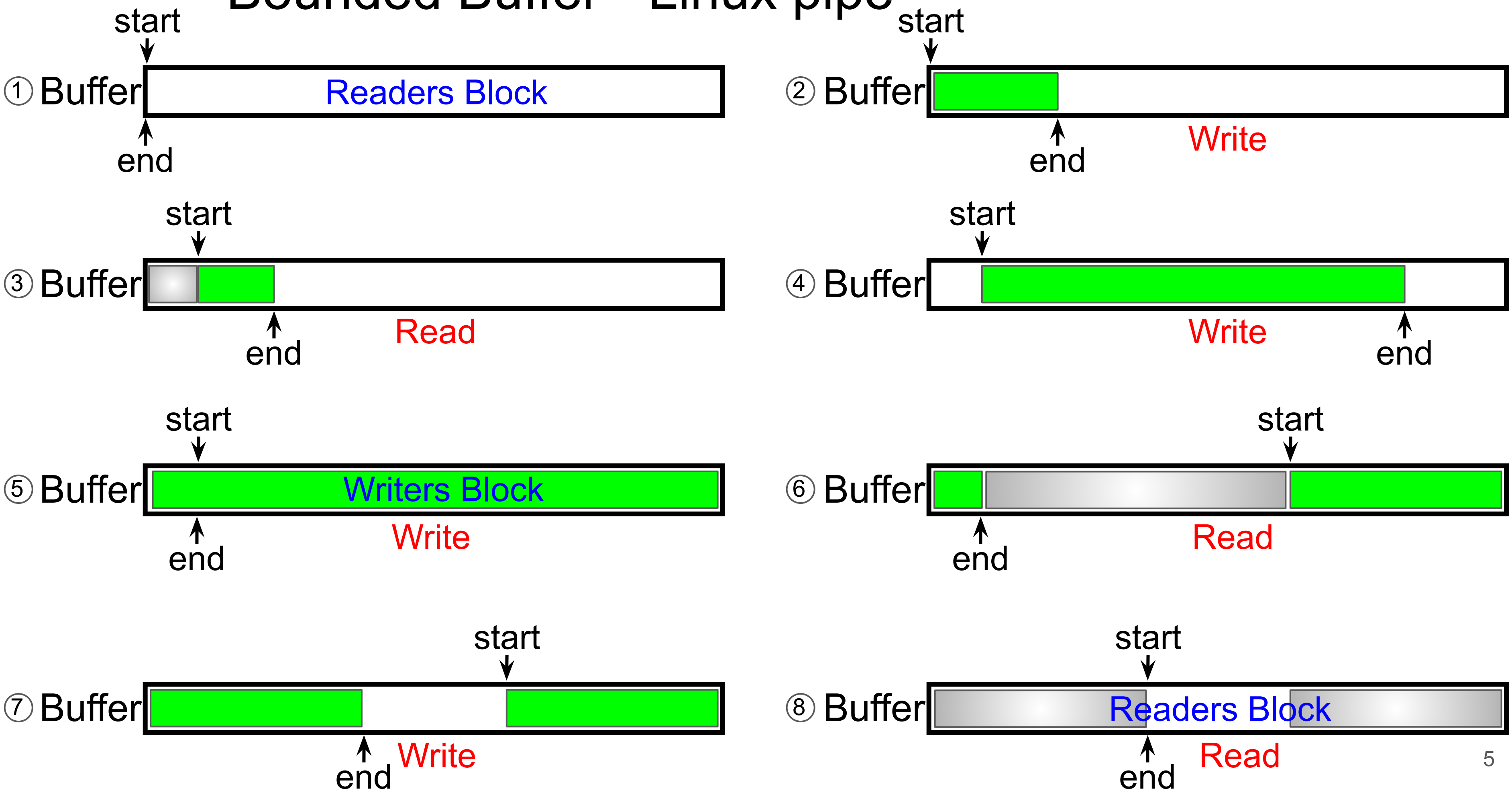
pthread Barrier API (from https://en.wikipedia.org/wiki/Barrier_(computer_science))

- `pthread_barrier_init()`
  Initialize the thread barrier with the number of threads needed to wait at the barrier in order to lift it
- `pthread_barrier_destroy()`
  Destroy the thread barrier to release back the resource
- `pthread_barrier_wait()`
  Calling this function will block the current thread until the number of threads specified by `pthread_barrier_init()` call `pthread_barrier_wait()` to lift the barrier.

Producer-Consumer - Linux pipes - Ordering Problem

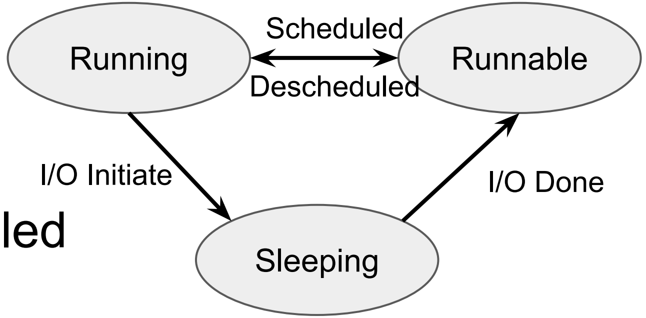
- A pipe may have many writers and readers
- Internally, there is a finite-sized buffer
- Writers (producers) add data to the buffer
  - Writers must wait if buffer is full
- Readers (consumers) remove data from the buffer
  - Readers must wait if buffer is empty
- Producer-Consumer or Bounded Buffer

Bounded Buffer - Linux pipe



Condition Variable - has queue of waiting threads

- Thread B waits for a signal on condition variable before running
  - `wait(CV, …)` - Thread B sleeps (or blocks)
- Thead A signals condition variable when time for Thread B to run
  - `signal(CV, …)` - Thread B moves from sleeping to runnable.



Condition Variable API

- `wait(cond_t *cv, mutex_t *lock)`
  - Assumes the lock is held when `wait()` is called
  - Puts caller to sleep and releases the lock
  - 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
    - The signal is not remembered, the signal is gone

Producer-Consumer Helper Functions

```
int max;  // variables and main
int loops;
int *buffer;
int use_ptr  = 0;
int fill_ptr = 0;
int num_full = 0;
int main(int argc, char **argv) {
  max = atoi(argv[1]);
  loops = atoi(argv[2]);
  consumers = atoi(argv[3]);

  buffer = malloc(max*sizeof(int));
…
}
```

```
// buffer functions
void do_fill(int value) {
   buffer[fill_ptr] = value;
   fill_ptr = (fill_ptr + 1) % max;
   num_full++;
}

int do_get() {
   int tmp = buffer[use_ptr];
   use_ptr = (use_ptr + 1) % max;
   num_full--;
   return tmp;
}
```

Producer-Consumer - Incorrect - One CV

```
void *producer(void *arg) {
  for (int i=0; i<loops; i++) {
    mutex_lock(&m);              //p1
    while(num_full == 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
    while(num_full == 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
 }
}
```

wait() wait() signal() wait() signal()

Producer: p1 p2 p4 p5 p6 p1 p2 p3

Consumer1: c1 c2 c3

Consumer2: c1 c2 c3 c2 c4 c5

does last signal wake producer or consumer2?

Producer-Consumer - Correct - Two CVs

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    mutex_lock(&m);
    while (numfull == max)
      cond_wait(&empty, &m);
    do_fill(i)
    cond_signal(&fill);
    mutex_unlock(&m);
  }
}
```

```
void *consumer(void *arg) {
  while (1) {
    mutex_lock(&m);
    while (numfull == 0)
      cond_wait(&fill, &m);
    int tmp = do_get();
    cond_signal(&empty);
    mutex_unlock(&m);
  }
}
```

Correct!
- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every do_fill()
- a producer will get to run after every do_get()

Programming with Condition Variables

- Programming uses a condition variable, mutex, and state variable
  - Keep state (`numfull` in prior charts) in addition to CV’s
- Must always do wait/signal with mutex lock held
- Whenever thread wakes from waiting, recheck state (This is the while loop)
  - Possible for another thread to grab lock in between signal and wakeup from wait

Condition Variables and Semaphores

- Condition variables have a wait queue, but they do not have other state information
  - Programmer tracks state with variables
  - For example, we added the variable `numfull` for the producer/consumer solution
- Semaphores have have a wait queue and an integer state
  - State is maintained by the semaphore semantics. Calling semaphore API alters the underlying state

Semaphore API

```
sem_t sem;
sem_init(sem_t *s, int zero, int initval)
sem_post(sem_t *s)
sem_wait(sem_t *s)
```

- `sem_init` - initializes the integer state of the semaphore
- `sem_wait` - decrement the semaphore by 1, wait on the queue if the value is negative.
- `sem_post` - increment the semaphore by 1, if there are threads waiting on the queue, wake one.

Create Mutex Lock with Semaphore

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(&lock->sem, 1); // create sem with val == 1
}
void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}
void release(lock_t *lock) {
    sem_post(&lock->sem);
}
```

`sem_wait()`: Decrement sem, If < 0, wait
`sem_post()`: Increment sem, then wake a waiter

Create Semaphore with CV

```
typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} sem_t;
```

```
void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```

`sem_wait()`: Decrement sem, If < 0, wait
`sem_post()`: Increment sem, then wake a waiter

```
sem_wait{sem_t *s) {
    mutex_lock(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond);
    s->value--;
    mutex_unlock(&s->lock);
}
```

```
sem_post{sem_t *s) {
    mutex_lock(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    mutex_unlock(&s->lock);
}
```

See zemaphore.c in Lab Multithreading

Producer/Consumer - Semaphores

- Circular Buffer - multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer
- Solution uses 2 semaphores
  - `emptyBuffer`: Initialize to N, producer can put N items in buffer
  - `fullBuffer`: Initialize to 0, consumer cannot consume until producer puts

Producer

```
int fill = 0;
put(int v) {
   buffer[fill] = v;
   fill = (fill+1) % N

}
i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    put(value);
    sem_post(&fullBuffer);
}
```

Consumer

```
int use = 0;
int get() {
   int t = buffer[use];
   use = (use+1) % N;
   return t;
}
j = 0;
while (1) {
    sem_wait(&fullBuffer);
    int value = get();
    sem_post(&emptyBuffer);
}
```

must guard

critical region

Producer/Consumer - Semaphores

- Circular Buffer - multiple producer threads, multiple consumer threads

Producer

```
int fill = 0;
put(int v) {
   buffer[fill] = v;
   fill = (fill+1)%N
}
sem_t mutex = 1;
i = 0;
while (1) {
   sem_take(&mutex);
   sem_take(&emptyBuffer);
   put(value);
   sem_post(&fullBuffer);
   sem_post(&mutex);
}
```

Producer waiting on mutex

Consumer waiting on fullBuffer

critical region

Consumer

```
int use = 0;
int get() {
   int t = buffer[use];
   use = (use+1)%N;
   return t;
}
j = 0;
while (1) {
   sem_take(&mutex);
   sem_take(&fullBuffer);
   int value = get();
   sem_post(&emptyBuffer);
   sem_post(&mutex)
}
```

Deadlock

Producer/Consumer - Semaphores

- Circular Buffer - multiple producer threads, multiple consumer threads

Producer

```
int fill = 0;
put(int v) {
    buffer[fill] = v;
    fill = (fill+1)%N
}
sem_t mutex = 1;
i = 0;
while (1) {
    sem_take(&emptyBuffer);
    sem_take(&mutex);
    put(value);
    sem_post(&mutex);
    sem_post(&fullBuffer);
}
```

Consumer

```
int use = 0;
int get() {
    int t = buffer[use];
    use = (use+1)%N;
    return t;
}
j = 0;
while (1) {
    sem_take(&fullBuffer);
    sem_take(&mutex);
    int value = get();
    sem_post(&mutex)
    sem_post(&emptyBuffer);
}
```

critical region

Deadlock Fixed

Semaphores

- Semaphores are equivalent to locks and condition variables
  - Can be used for both mutual exclusion and ordering
- Semaphores contain state - an integer
  - How they are initialized depends on how they will be used
  - Init to 1: Mutex
  - Init to N: Number of available resources - producer/consumer solution

Semaphore API

- `sem_wait` - decrement the semaphore by 1, wait if the value < 0
- `sem_wait` - waits until value > 0, then decrement (atomic)
- `sem_post` - increment the semaphore by 1, if there are threads waiting, wake one

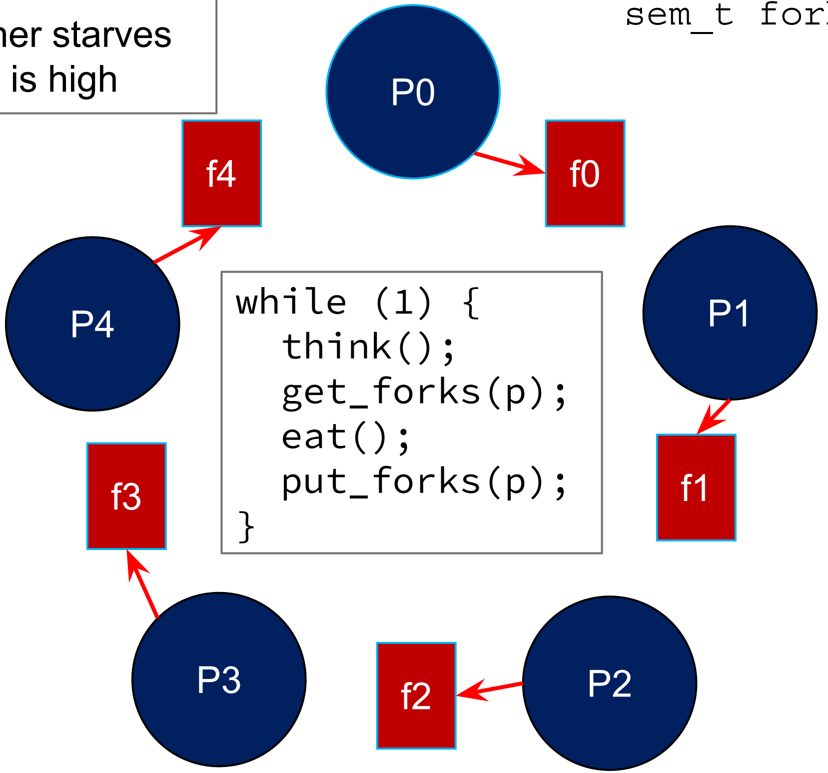
Producer-Consumer Code

- Lab Multithreading (git checkout thread), folder notxv6/pc has code
  - pc_cv.c - condition variable solution to producer-consumer
  - pc_sem.c - semaphore solution to producer-consumer
- Run both programs with various values for
  - <buffersize> <loops> <consumers>
  - Write observations in your notebook
- Update the programs to allow for multiple producers, run your updated program
  - Write observations in your notebook
- Questions
  - How is the buffer defined?
  - What is the argument passed to producer and consumer threads?
  - What is the technique used to terminate the consumer threads?
  - Which of the two solutions do your like better? Why?

Dining Philosophers - (See OSTEP Ch 31, pp 13-18)

Write get_forks() and put_forks() such that
- no deadlock
- no philosopher starves
- concurrency is high

```
sem_t forks[5] = {1};
```




```
while (1) {
  think();
  get_forks(p);
  eat();
  put_forks(p);
}
```

Deadlock Solution

Philosphers all grab forks to the left.
Everyone is waiting on a fork.

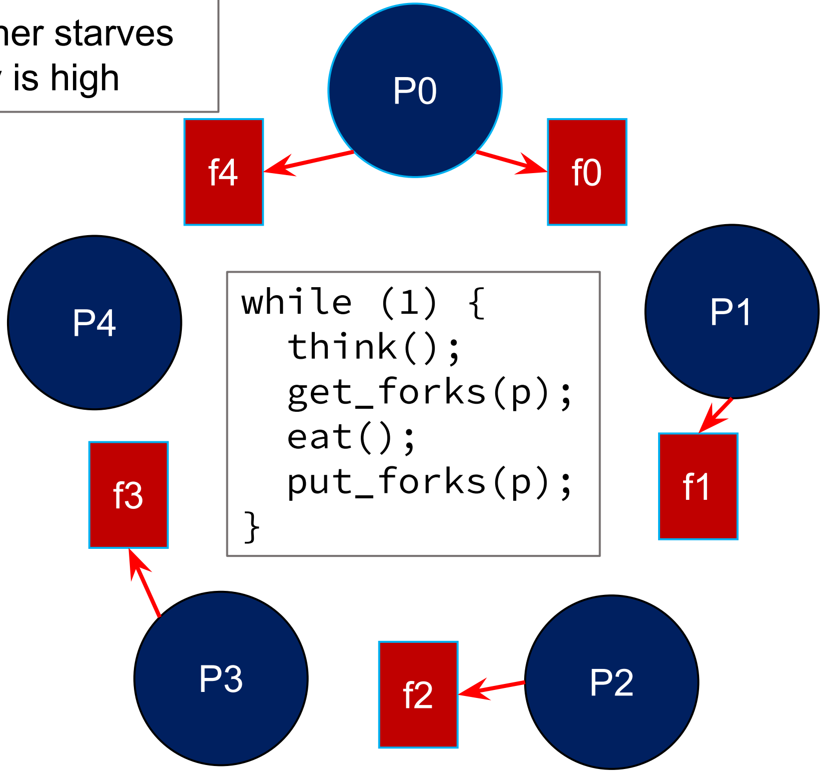
```
void get_forks(int p) {
    Sem_wait(&forks[left(p)]);
    Sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    Sem_post(&forks[left(p)]);
    Sem_post(&forks[right(p)]);
}
```

20

Dining Philosophers - (See OSTEP Ch 31, pp 13-18)

Write get_forks() and put_forks() such that
- no deadlock
- no philosopher starves
- concurrency is high




```
while (1) {
  think();
  get_forks(p);
  eat();
  put_forks(p);
}
```

```
sem_t forks[5] = {1};
```

Correct Solution

Four Philosophers all grab lowest number forks first.
Fifth philosopher grabs highest number fork first.

```
void get_forks(int p) {
  if (p == 0) {
     Sem_wait(&forks[right(p)]);
     Sem_wait(&forks[left(p)]);
  } else {
     Sem_wait(&forks[left(p)]);
     Sem_wait(&forks[right(p)]);
  }
}
void put_forks(int p) {
  Sem_post(&forks[left(p)]);
  Sem_post(&forks[right(p)]);
}
```