Synchronization Primitives (ch. 30+31+32)

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Condition Variables

- Many cases a thread wishes to wait until a certain **condition**
- e.g., waiting for another thread to complete
  - Often called a `join()`
- Shared variable: works, but hugely inefficient

How should a thread wait for a condition?

Condition Variables

- **Waiting** on a condition
  - Thread puts itself in a queue until some state of execution
- **Signaling** on a condition
  - Some other thread can wake waiting thread

Condition Variables

- **Waiting** on a condition
  - Thread puts itself in a queue until some state of execution
- **Signaling** on a condition
  - Some other thread can wake waiting thread

- Name is a bit misleading
  - More of a queue
  - We are responsible for the actual “condition”

Definitions and Routines

- Declare a condition variable:

```
pthread_cond_t cv;
```

- Operations:

```
// wait:
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
// signal:
pthread_cond_signal(pthread_cond_t *c);
```

- Wait call takes **mutex** as a parameter
  - Caller must be its **owner** (have it locked)
  - Releases the lock, puts caller to sleep
  - On wake up, re-acquires lock and returns

Parent Waiting For Child

- Two threads:
  - **Parent**:
    - Creates child thread
    - Waits on CV until child completes
  - **Child**:
    - Prints a message ("child")
    - Wakes parent by signaling on CV

Parent Waiting For Child

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void* child(void* arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}
int main(void) {
    printf("parent: begin\n");
    pthread_t p;
    pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Parent Waiting For Child

```
void thr_exit() {
    pthread_cond_signal(&c);
}
void thr_join() {
    pthread_mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Why might this code fail?

Parent Waiting For Child

```
void thr_exit() {
    pthread_cond_signal(&c);
}
void thr_join() {
    pthread_mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Why might this code fail?
    - Child runs immediately
    - Will signal, but no thread asleep on CV
    - Parent runs, calls wait and gets stuck
    - Solution?

Parent Waiting For Child

```
void thr_exit() {
    pthread_cond_signal(&c);
}
void thr_join() {
    pthread_mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Why might this code fail?
  - Child runs immediately
  - Will signal, but no thread asleep on CV
  - Parent runs, calls wait and gets stuck
  - Solution? use `done` variable

Parent Waiting For Child

```c
int done = 0;
void thr_exit() {
    done = 1;
    pthread_cond_signal(&c);
}
void thr_join() {
    pthread_mutex_lock(&m);
    if (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Why might this code fail?

Parent Waiting For Child

```
int done = 0;
void thr_exit() {
    done = 1;
    pthread_cond_signal(&c);
}
void thr_join() {
    pthread_mutex_lock(&m);
    if (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Why might this code fail?
    - Parent calls join, sees `done=0`
    - Interrupted just before wait, context switch to child
    - Child sets done, signal is lost, parent is stuck again
    - Solution?

Parent Waiting For Child

```c
int done = 0;
void thr_exit() {
    done = 1;
    pthread_cond_signal(&c);
}
void thr_join() {
    pthread_mutex_lock(&m);
    if (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Why might this code fail?
  - Parent calls join, sees `done=0`
  - Interrupted just before wait, context switch to child
  - Child sets done, signal is lost, parent is stuck again
  - Solution? hold lock while signaling

Parent Waiting For Child

```c
int done = 0;
void thr_exit() {
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
void thr_join() {
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Additionally, check variable in a loop
  - Condition variable may signal unexpectedly
  - Also crucial for more than 2 threads

Covering Conditions

- Memory allocator implementation
- Assume zero bytes are free:
  - Thread A calls `allocate(100)`
  - Thread B calls `allocate(10)`
  - Both A and B wait on the condition
  - Thread C calls `free(50)`

Covering Conditions

- Memory allocator implementation
- Assume zero bytes are free:
  - Thread A calls `allocate(100)`
  - Thread B calls `allocate(10)`
  - Both A and B wait on the condition
  - Thread C calls `free(50)`
    - Which waiting thread wakes up?

Covering Conditions

- Memory allocator implementation
- Assume zero bytes are free:
  - Thread A calls `allocate(100)`
  - Thread B calls `allocate(10)`
  - Both A and B wait on the condition
  - Thread C calls `free(50)`
    - Which waiting thread wakes up?
- Solution: wake up all waiting threads
  - Use `pthread_cond_broadcast()`
  - Performance cost: too many threads might be woken

Producer / Consumer

- Also: **bounded buffer** problem
- **Producer**
  - Produces data items
  - Wishes to place items in a buffer
- **Consumer**
  - Grabs items out of the buffer
  - Consumes items in some way
- e.g., web server consumes HTTP requests (work queue)

Producer / Consumer

- Also used in pipes:
  - `grep foo file.txt | wc -l`
  - `grep` - output lines from `file.txt` containing `foo`
  - `wc -l` - output number of lines from input
  - Shell redirects `grep` standard output to a **pipe**
    - Created by the `pipe` system call
  - Other end connected to standard input of `wc`
  - `grep` - producer, `wc` - consumer
- In-kernel bounded buffer

Producer / Consumer

```c
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;
}
```

First Attempt

- What are the two problems?

```c
pthread_cond_t cond;
pthread_mutex_t mutex;

void produce(int i) {
    pthread_mutex_lock(&mutex);
    if (count == 1)
        pthread_cond_wait(&cond, &mutex);
    put(i);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
int consume() {
    pthread_mutex_lock(&mutex);
    if (count == 0)
        pthread_cond_wait(&cond, &mutex);
    int tmp = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return tmp;
}
```

First Attempt

- More than one consumer / producer

First Attempt

- More than one consumer / producer
- Two consumers:
  - No `while` on CV in `consume()`
  - Can consume when empty!
  - (and the same for `produce()`)

```
int consume() {
    pthread_mutex_lock(&mutex);
    while (count == 0)
        pthread_cond_wait(&cond, &mutex);
    int tmp = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return tmp;
}
```

First Attempt

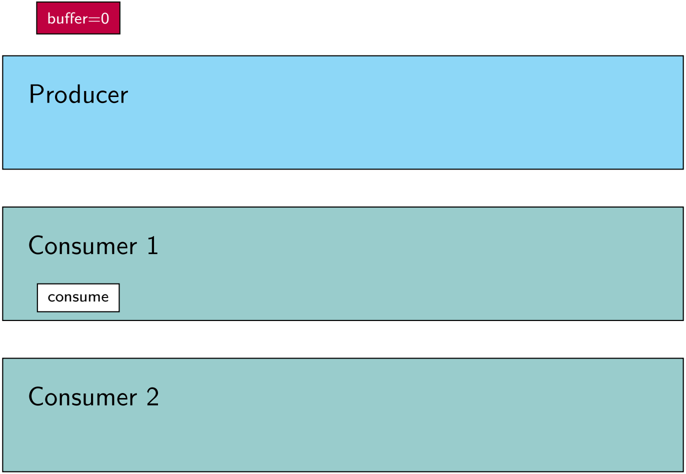
buffer=0

Producer

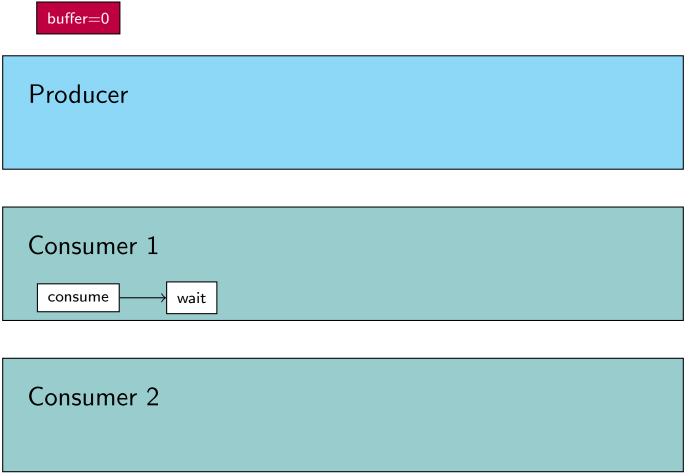
Consumer 1

Consumer 2

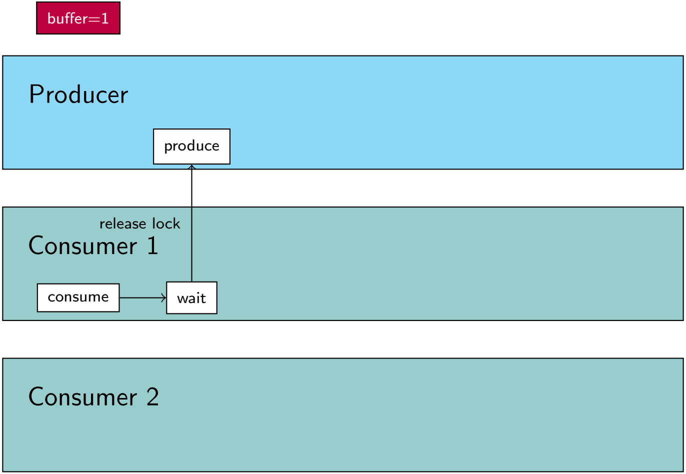
First Attempt



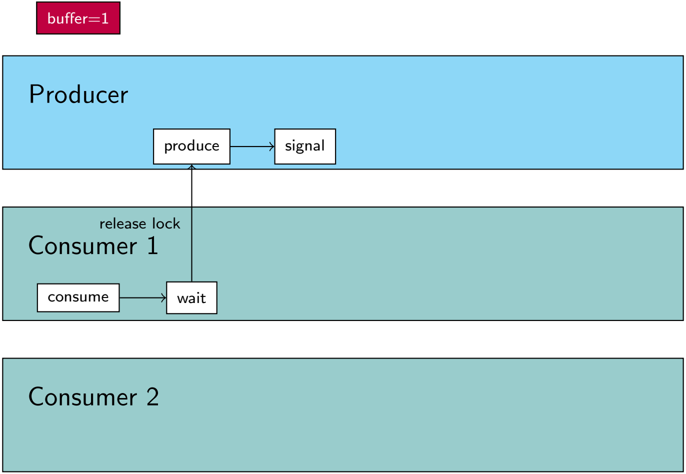
First Attempt



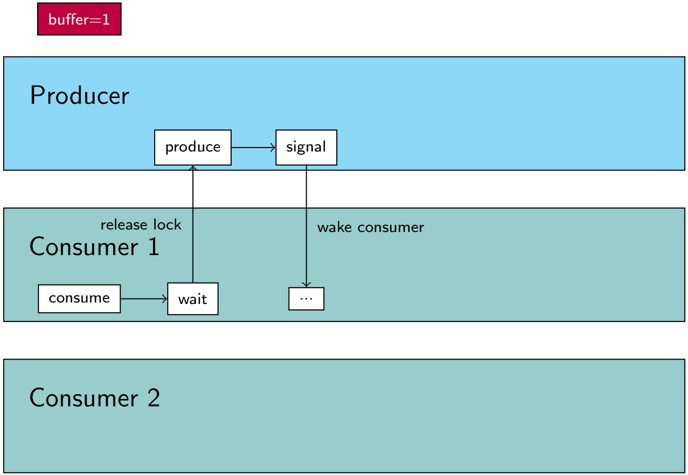
First Attempt



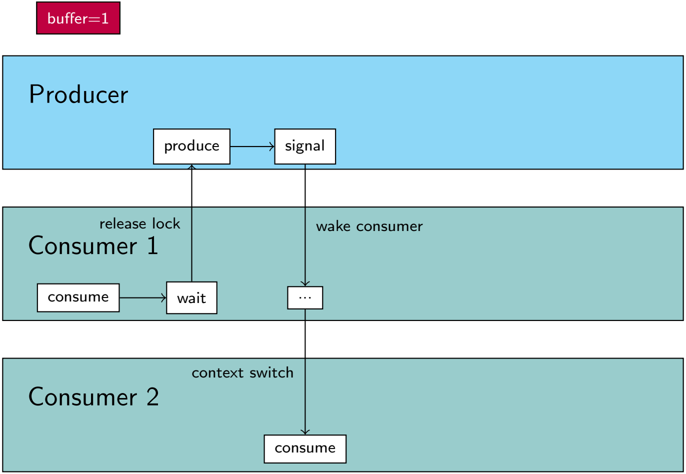
First Attempt



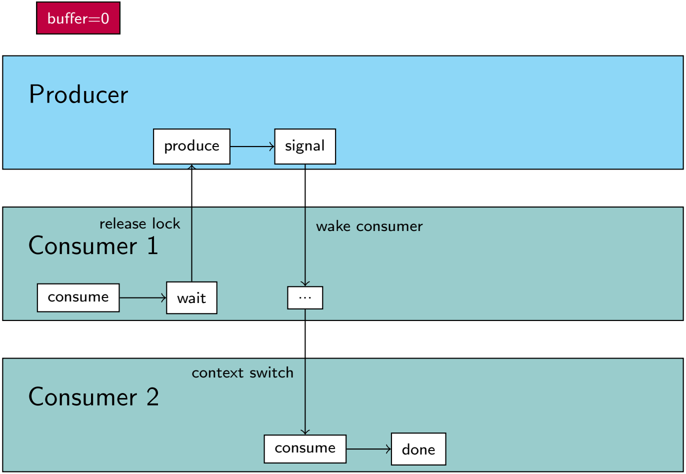
First Attempt



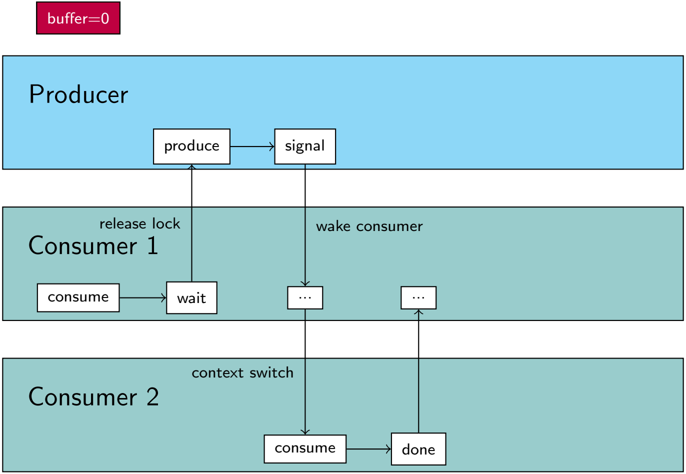
First Attempt



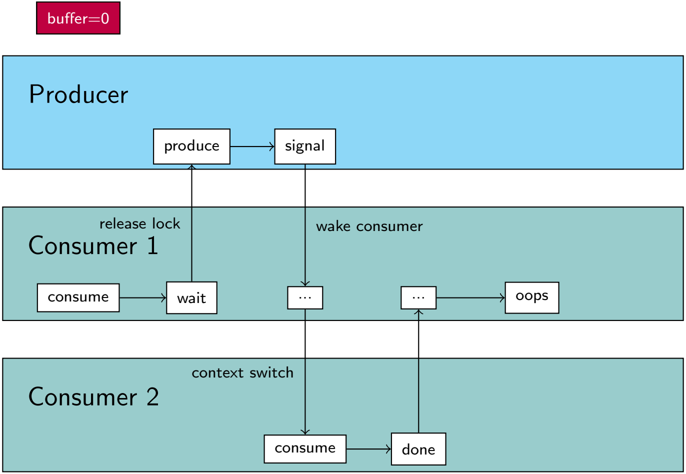
First Attempt



First Attempt



First Attempt



First Attempt

- Only one condition variable!
  - Consumer might wake another consumer
  - Producer might wake another producer
- Solution?

First Attempt

- Only one condition variable!
  - Consumer might wake another consumer
  - Producer might wake another producer
- Solution? use **two** condition variables
  - Producer threads wait on `empty`
  - Consumer threads wait on `full`

Semaphores

- Another **synchronization primitive**
  - Can use **semaphores** as both locks and condition variables
- A **semaphore**
  - Object with an integer value
  - `sem_wait()`, `sem_post()`

Semaphores

- Initialization:

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

- Initialized to the value 1 (third argument)
  - Second argument: 0 to share between threads (vs. processes)
  - We will only use 0

Semaphores

- `sem_wait()`:
  - Decrement semaphore value by one
  - Wait if value is negative
- `sem_post()`:
  - Increment semaphore value by one
  - Wake one waiting thread
- Both operations are performed **atomically**

Binary Semaphores

- How can we use a semaphore as a lock?

```
sem_init(&m, 0, X); // what should X be?
...
sem_wait(&m);
// critical section
sem_post(&m);
```

- What should **X** be?

Binary Semaphores

- How can we use a semaphore as a lock?

```
sem_init(&m, 0, X); // what should X be?
...
sem_wait(&m);
// critical section
sem_post(&m);
```

- What should **X** be?
  - X = 1
  - **Binary semaphore**

Ordering Primitive

- Similar to **condition variables**
- How can we use a semaphore to wait for an event?

```
sem_t s;

void* child(void* arg) {
    printf("child\n");
    sem_post(&s); // child is done
    return NULL;
}
int main() {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait for child
    printf("parent: end\n");
    return 0;
}
```

- What should **X** be?

Ordering Primitive

- Similar to **condition variables**
- How can we use a semaphore to wait for an event?

```
sem_t s;

void* child(void* arg) {
    printf("child\n");
    sem_post(&s); // child is done
    return NULL;
}
int main() {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait for child
    printf("parent: end\n");
    return 0;
}
```

- What should **X** be?
    - X=0

Semaphores

- What value should a semaphore be initialized to?
  - General rules: number of resources
  - Lock: 1 $\rightarrow$ can be locked after initialization
  - Ordering: 0 $\rightarrow$ nothing to give away at the start

Producer / Consumer

- How can we implement a **bounded buffer** with semaphores?

```c
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
}
int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    return tmp;
}
```

First Attempt

- What is the problem?

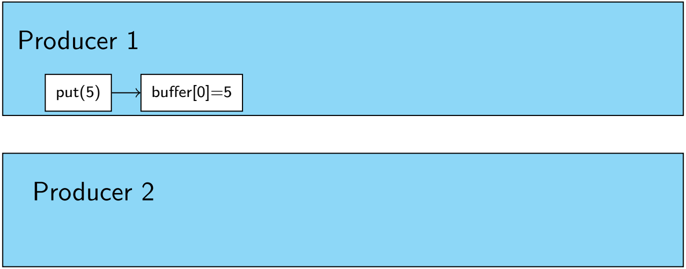
```c
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&empty);
    put(value);
    sem_post(&full);
}
int consume() {
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    return tmp;
}
```

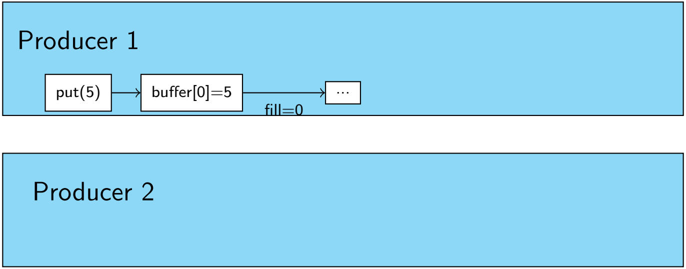
First Attempt

First Attempt

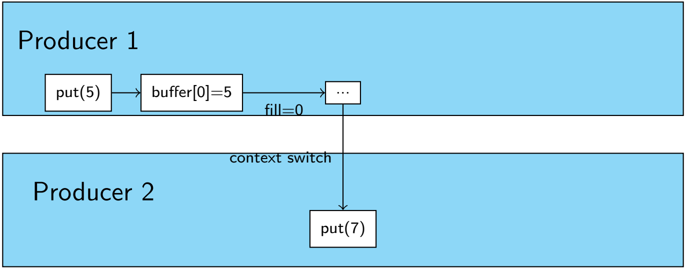
First Attempt



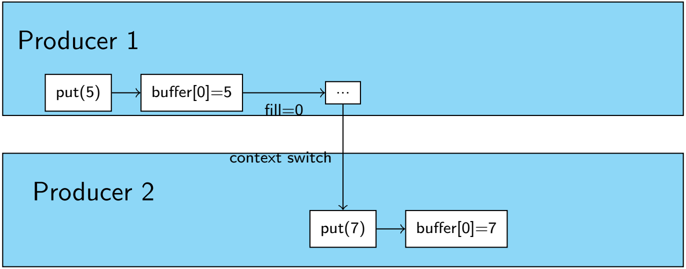
First Attempt



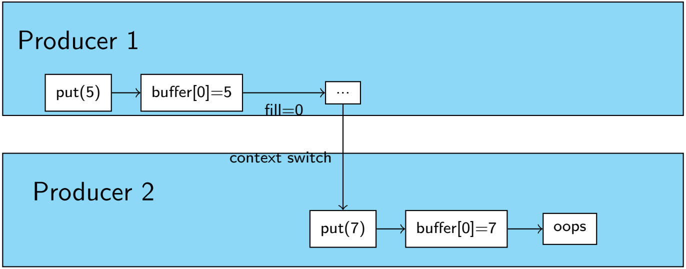
First Attempt



First Attempt



First Attempt



With Mutual Exclusion

- What is the problem?

```c
sem_t mutex; // binary semaphore
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(value);
    sem_post(&full);
    sem_post(&mutex);
}
int consume() {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    return tmp;
}
```

With Mutual Exclusion

- What is the problem? **deadlock!**
  - Producer waits on empty, holds mutex, consumer can't consume

```
sem_t mutex; // binary semaphore
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(value);
    sem_post(&full);
    sem_post(&mutex);
}
int consume() {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    return tmp;
}
```

With Mutual Exclusion

- Solution: use mutex around the critical section

```
sem_t mutex; // binary semaphore
sem_t empty; // initialized to MAX
sem_t full;  // initialized to 0

void produce(int value) {
    sem_wait(&empty);
    sem_wait(&mutex);
    put(value);
    sem_post(&mutex);
    sem_post(&full);
}
int consume() {
    sem_wait(&full);
    sem_wait(&mutex);
    int tmp = get();
    sem_post(&mutex);
    sem_post(&empty);
    return tmp;
}
```

Reader-Writer Locks

- More flexible locking primitive
  - e.g., concurrent operations: inserts and lookups
  - Insert changes state $\rightarrow$ traditional critical section
  - Lookup reads data structure $\rightarrow$ many at once (if no insert)
- **Reader-writer lock**
  - Four operations: acquire/release read/write lock

Reader-Writer Locks

- A **single writer** can acquire the lock
- Once a reader acquires a **read lock**:
  - **More readers** are allowed to acquire the read lock
  - A writer waits until all readers are finished

Reader-Writer Locks

- A **single writer** can acquire the lock
- Once a reader acquires a **read lock**:
  - **More readers** are allowed to acquire the read lock
  - A writer waits until all readers are finished

```
typedef struct _rwlock_t {
    sem_t lock;          // binary semaphore
    sem_t writelock;     // used to allow ONE writer
    int readers;         // count of readers in CS
} rwlock_t;

void rwlock_init(rwlock_t* rw) {
    rw->readers = 0;
    sem_init(&rw-lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}
```

Reader-Writer Locks

```
void rwlock_acquire_writelock(rwlock_t* rw) {
    sem_wait(&rw->writelock);
}
void rwlock_release_writelock(rwlock_t* rw) {
    sem_post(&rw->writelock);
}

void rwlock_acquire_readlock(rwlock_t* rw) {
    sem_wait(&rw->lock);     // CS for readers
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock); // first reader grabs writelock
    sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t* rw) {
    sem_wait(&rw->lock);     // CS for readers
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock); // last reader releases writelock
    sem_post(&rw->lock);
}
```

Reader-Writer Locks

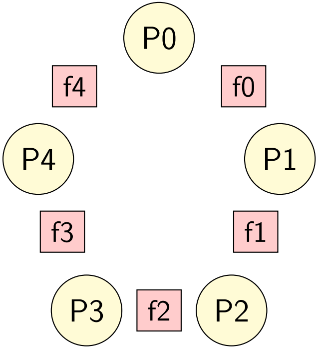
- What is the problem?

Reader-Writer Locks

- What is the problem? **fairness**
  - Easy to **starve** writer
  - How to prevent readers from starving writers?

Dining Philosophers

- Five philosophers around a table
  - Single fork between each pair
  - Philosophers **think** and **eat**
  - Two forks to eat (left and right)



Dining Philosophers

- As code:

```c
while (1) {
    think();
    getforks();
    eat();
    putforks();
}
```

```c
int left(int p) {
    return p;
}
int right(int p) {
    return (p - 1) % 5;
}
```

Dining Philosophers

- Use a semaphore for each 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)]);
}
```

- The problem?

Dining Philosophers

- Use a semaphore for each 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)]);
}
```

- The problem? **deadlock!**
  - Each philosopher grabs fork on their left
  - All waiting for their right

Dining Philosophers

- Solution: break the dependency

```c
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    }
    else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

Thread Throttling

- For example: hundreds of threads work in parallel
- Section of code allocates a lot of memory
    - All threads at the same time $\rightarrow$ exceeds physical memory
    - Machine will start thrashing (swapping to and from the disk)

- Solution?

Thread Throttling

- For example: hundreds of threads work in parallel
- Section of code allocates a lot of memory
  - All threads at the same time → exceeds physical memory
  - Machine will start thrashing (swapping to and from the disk)

- Solution? **semaphore**!
  - Initialized to max threads we wish to enter code section
  - Surrounds code section, limits concurrent threads in it

Implementing Semaphores

- Doesn't maintain invariant: negative value $\rightarrow$ # waiting threads
  - Easier, matches the Linux implementation

```c
typedef struct __zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
};

void zem_init(zem_t* s, int value) {
    s->value = value;
    pthread_cond_init(&s->cond);
    pthread_mutex_init(&s->lock);
}
void zem_wait(zem_t* s) {
    pthread_mutex_lock(&s->lock);
    while (s->value <= 0)
        pthread_cond_wait(&s->cond, &s->lock);
    s->value--;
    pthread_mutex_unlock(&s->lock);
}
void zem_post(zem_t* s) {
    pthread_mutex_lock(&s->lock);
    s->value++;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->lock);
}
```

Summary

- **Condition variables**
  - Thread waits until a certain condition
  - `wait()`, `signal()`
  - **Hold lock** while signaling
  - Check value **in a loop**
- **Semaphore**
  - Integer value
  - Decrement on acquire, wait if negative, increment on release
- **Read-write lock**
  - **Single writer** or **multiple readers**
- Dining philosophers
  - Think and eat
- Producer / consumer (bounded buffer)