CSE 323: Operating System Design

Condition Variables

Salman Shamil

North South University (NSU)
Fall 2025

Original slides by Mathias Payer and Sanidhya Kashyap [EPFL]

Lecture Topics

- Condition Variables
- Producer-Consumer Problem

This slide deck covers chapters 30 in OSTEP.

Condition Variables (CV)

In concurrent programming, a common scenario is one thread waiting for another thread to complete an action.

```
bool done = false;

/* called in the child to signal termination */
void thr_exit() {
 done = true;
}
/* called in the parent to wait for a child thread */
void thr_join() {
  while (!done);
}
```

Condition Variables (CV)

- Locks enable mutual exclusion of a shared region.
    - Unfortunately they are oblivious to ordering
- Waiting and signaling (i.e., T2 waits until T1 completes a given task) could be implemented by spinning until the value changes

Condition Variables (CV)

- Locks enable mutual exclusion of a shared region.
  - Unfortunately they are oblivious to ordering
- Waiting and signaling (i.e., T2 waits until T1 completes a given task) could be implemented by spinning until the value changes
- But spinning is incredibly *inefficient*
- New synchronization primitive: ***condition variables***

Condition Variables (CV)

- A CV allows:
  - A thread to wait for a condition
  - Another thread signals the waiting thread
- Implement CV using queues

Condition Variables (CV)

- A CV allows:
  - A thread to wait for a condition
  - Another thread signals the waiting thread
- Implement CV using queues
- API: `wait`, `signal` or `broadcast`
  - `wait`: wait until a condition is satisfied
  - `signal`: wake up one waiting thread
  - `broadcast`: wake up all waiting threads
- On Linux, `pthreads` provides CV implementation

Signal parent that child has exited

```c
bool done = false;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
/* called in the child to signal termination */
void thr_exit() {
  pthread_mutex_lock(&m);
  done = true;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
 }
/* called in the parent to wait for a child thread */
void thr_join() {
 pthread_mutex_lock(&m);
 while (!done)
   pthread_cond_wait(&c, &m);
 pthread_mutex_unlock(&m);
}
```

Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?

Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?
- Thread may have already exited, i.e., no need to wait
  - Principle: Check the condition before sleeping

Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?
- Thread may have already exited, i.e., no need to wait
  - Principle: Check the condition before sleeping
- Question: Why can't we use `if` when waiting?

Signal parent that child has exited (2)

- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
  - Assume mutex `m` is held; *atomically* unlock mutex when waiting, retake it when waking up
- Question: Why do we need to check a condition before sleeping?
- Thread may have already exited, i.e., no need to wait
  - Principle: Check the condition before sleeping
- Question: Why can't we use `if` when waiting?
- Multiple threads could be woken up, racing for `done` flag
  - Principle: `while` instead of `if` when waiting

Signal parent that child has exited (3)

- Question: Why do we need to proctect `done` with mutex `m`?

Signal parent that child has exited (3)

- Question: Why do we need to proctect `done` with mutex `m`?
- Mutex `m` allows one thread to access `done` for protecting against missed updates
  - Parent reads `done == false` but is interrupted
  - Child sets `done = true` and signals but no one is waiting
  - Parent continues and goes to sleep (forever)
- Lock is therefore required for wait/signal synchronization

Producer/Consumer Problem

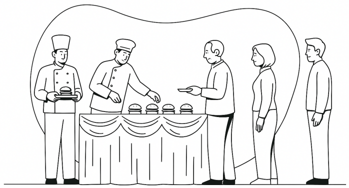

Figure 1: Producer-Consumer/Bounded Buffer Problem

- Producer/consumer is a common programming pattern
- For example: map (producers) / reduce (consumer)
- For example: a concurrent database (consumers) handling parallel requests from clients (producers)
  - Clients produce new requests (encoded in a queue)
  - Handlers consume these requests (popping from the queue)

Producer/Consumer with Bounded Buffer

- One or more producers create items, store them in buffer
- One or more consumers process items from buffer

Producer/Consumer with Bounded Buffer

- One or more producers create items, store them in buffer
- One or more consumers process items from buffer
- Need synchronization for buffer
  - Want concurrent production and consumption
  - Use as many cores as available
  - Minimize access time to shared data structure

Producer/Consumer with Bounded Buffer

- One or more producers create items, store them in buffer
- One or more consumers process items from buffer
- Need synchronization for buffer
  - Want concurrent production and consumption
  - Use as many cores as available
  - Minimize access time to shared data structure
- Strategy: use CV to synchronize
  - Make producers wait if buffer is full
  - Make consumers wait if buffer is empty (nothing to consume)

Solving Producer/Consumer Problem

- **Setup:**
  - Buffer holds a single item
  - One producer and one consumer

Solving Producer/Consumer Problem

- **Setup:**
  - Buffer holds a single item
  - One producer and one consumer

```
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;
}
```

```
void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    put(i);
  }
}

void *consumer(void *arg) {
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

Solving Producer/Consumer Problem

- **Setup:**
  - Buffer holds a single item
  - One producer and one consumer

```
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;
}
```

```
void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    put(i);
  }
}

void *consumer(void *arg) {
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

- **Problems with this solution**
  - Critical sections in `put()` and `get()`. **Use locks...**

Solving Producer/Consumer Problem

- **Setup:**
  - Buffer holds a single item
  - One producer and one consumer

```
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;
}
```

```
void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    put(i);
  }
}

void *consumer(void *arg) {
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

- **Problems with this solution**
  - Critical sections in `put()` and `get()`. **Use locks...**
- Producer-Consumer dependency for fetching. **Needs CV!**

Solving Producer/Consumer Problem

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    if (count == 1)
      Pthread_cond_wait(&cond, &mutex);
    put(i);
    Pthread_cond_signal(&cond);
    Pthread_mutex_unlock(&mutex);
  }
}
```

```
void *consumer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    if (count == 0)
      Pthread_cond_wait(&cond, &mutex);
    int tmp = get();
    Pthread_cond_signal(&cond);
    Pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

Solving Producer/Consumer Problem

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    if (count == 1)
      Pthread_cond_wait(&cond, &mutex);
    put(i);
    Pthread_cond_signal(&cond);
    Pthread_mutex_unlock(&mutex);
  }
}
```

```
void *consumer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    if (count == 0)
      Pthread_cond_wait(&cond, &mutex);
    int tmp = get();
    Pthread_cond_signal(&cond);
    Pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

Does it work?

- Fine for single producer and single consumer.
- Change the setup to accommodate multiple producers and/or multiple consumers. How about now?

Solving Producer/Consumer Problem (2)

- **Setup:**
    - Buffer holds a single item
    - Multiple producers and/or multiple consumers
- Previous code doesn't work.

Solving Producer/Consumer Problem (2)

- **Setup:**
  - Buffer holds a single item
  - Multiple producers and/or multiple consumers
- Previous code doesn't work.

No recheck after waking up.

- Consider a consumer thread (C1) is waiting for an item
- What if a second consumer thread (C2) sneaks in just after an item is produced? . . . skipping the `wait()` call.
- Producer's `signal()` wakes C1 up, but C2 already fetched the item!
- Solution: Use `while` instead of `if` to recheck upon waking up.

Solving Producer/Consumer Problem (2)

- **Setup:**
  - Buffer holds a single item
  - Multiple producers and/or multiple consumers
- Previous code doesn't work.

No recheck after waking up.

- Consider a consumer thread (C1) is waiting for an item
- What if a second consumer thread (C2) sneaks in just after an item is produced? ... skipping the `wait()` call.
- Producer's `signal()` wakes C1 up, but C2 already fetched the item!
- Solution: Use `while` instead of `if` to recheck upon waking up.

Producers and consumers both waiting on the same CV.

- Two consumers C1 and C2 runs and sleeps by calling `wait()`.
- Producer runs and `signal()` wakes up C1 (or C2).
- After consuming the item C1 can wake up producer again.
- But what if C1's `signal()` wakes up C2 instead?
- Solution: Use separate conditions for directed signaling.

Solving Producer/Consumer Problem (2)

```c
cond_t empty, full;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    while (count == 1)
      Pthread_cond_wait(&empty, &mutex);
    put(i);
    Pthread_cond_signal(&full);
    Pthread_mutex_unlock(&mutex);
  }
}
```

```c
void *consumer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    while (count == 0)
      Pthread_cond_wait(&full, &mutex);
    int tmp = get();
    Pthread_cond_signal(&empty);
    Pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

Producer/Consumer Buffer with Multiple Slots

```c
int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0;

void put(int value) {
  buffer[fill_ptr] = value;
  fill_ptr = (fill_ptr + 1) % MAX;
  count++;
}

int get() {
  int tmp = buffer[use_ptr];
  use_ptr = (use_ptr + 1) % MAX;
  count--;
  return tmp;
}


cond_t empty, fill;
mutex_t mutex;
```

```c
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    while (count == MAX)
      Pthread_cond_wait(&empty, &mutex);
    put(i);
    Pthread_cond_signal(&fill);
    Pthread_mutex_unlock(&mutex);
  }
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    Pthread_mutex_lock(&mutex);
    while (count == 0)
      Pthread_cond_wait(&fill, &mutex);
    int tmp = get();
    Pthread_cond_signal(&empty);
    Pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

[Self-study] Semaphore

- A semaphore extends a CV with an integer as internal state
- `int sem_init(sem_t *sem, unsigned int value)`: creates a new semaphore with `value` slots
- `int sem_wait(sem_t *sem)`: waits until the semaphore has at least one slot, decrements the number of slots
- `int sem_post(sem_t *sem)`: increments the semaphore (and wakes one waiting thread)
- `int sem_destroy(sem_t *sem)`: destroys the semaphore and releases any waiting threads