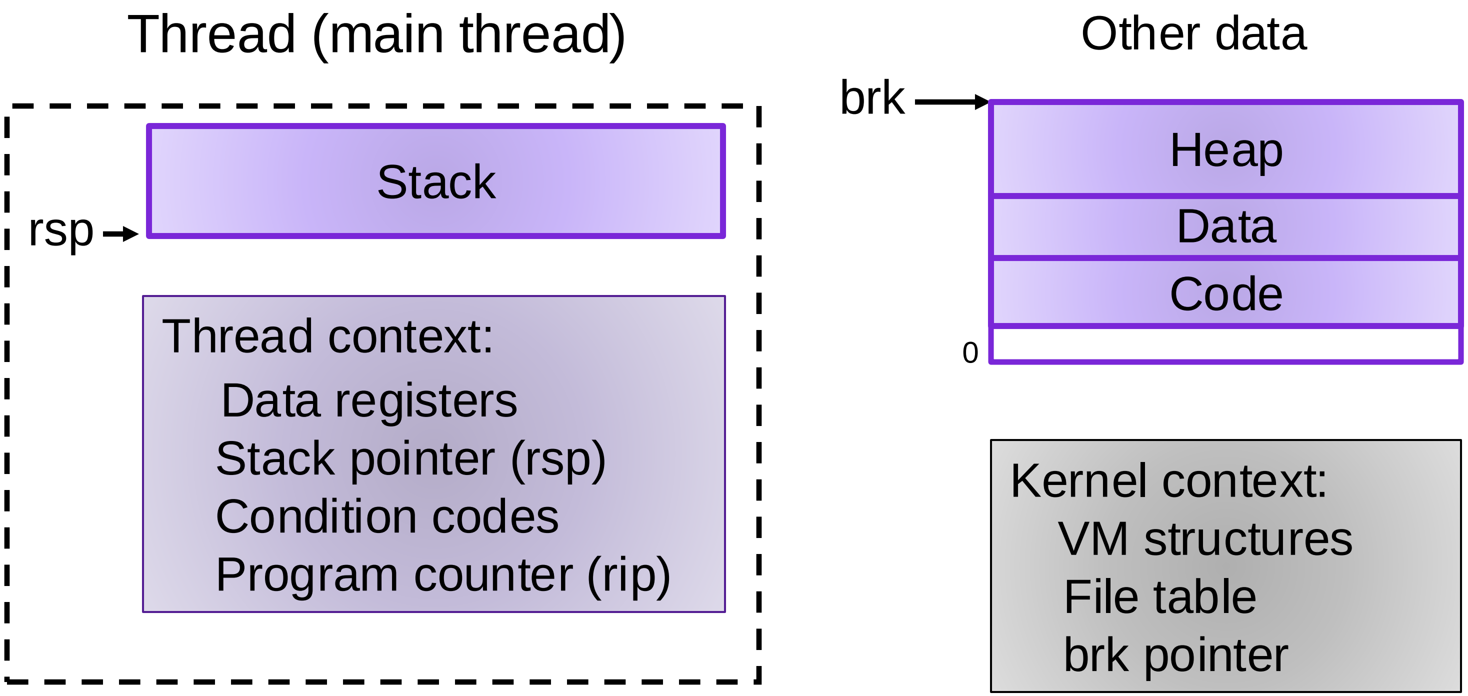
Lecture 20: Synchronization

CS 105 Fall 2024

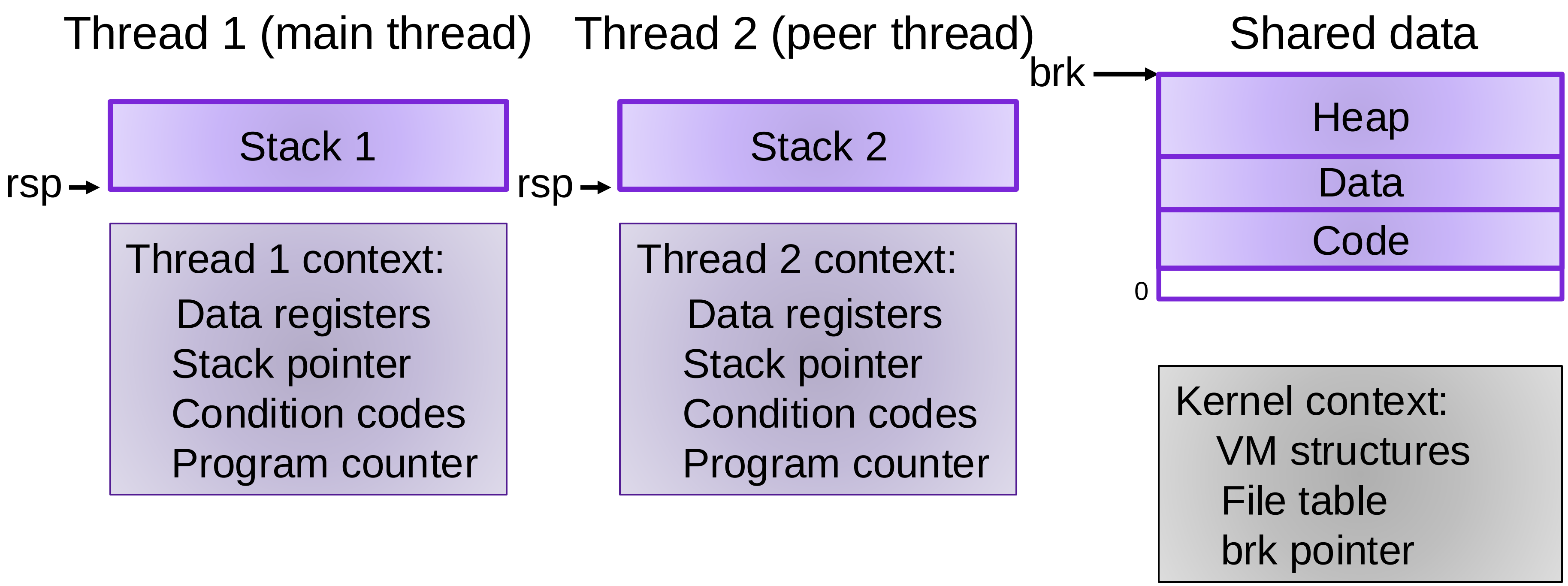
Review: Alternate View of a Process

- Process = thread + other state



Review: Multi-threading

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread has its own stack for local variables
  - Each thread has its own thread id (TID)
  - Each thread shares the same code, data, and kernel context



Review: Locks

- A **lock** (aka a mutex) is a synchronization primitive that provides mutual exclusion. When one thread holds a lock, no other thread can hold it.
  - a lock can be in one of two states: locked or unlocked
  - a lock is initially unlocked

  - function acquire(&lock) waits until the lock is unlocked, then atomically sets it to locked
  - function release(&lock) sets the lock to unlocked

Review: use a lock

- You and your roommate share a refrigerator. Being good roommates, you both try to make sure that the refrigerator is always stocked with milk.

Algorithm 6:

```
acquire(&lock)
if (milk == 0) {      // no milk
  milk++;             // buy milk
}
release(&lock)
```

Review: Locks

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */
pthread_mutex_t lock; /* Lock */

int main(int argc, char** argv){
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    pthread_create(&tid1, NULL,
        thread, &niters);
    pthread_create(&tid2, NULL,
        thread, &niters);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

```c
/* Thread routine */
void* count_func(void* vargp){
  long i, niters;
  niters = *((long*)vargp);

  for (i = 0; i < niters; i++){
    pthread_mutex_lock(&lock);
    cnt++;
    pthread_mutex_unlock(&lock);
  }

  return NULL;
}
```

Problems with Locks

1. Locks are slow
   - threads that fail to acquire a lock on the first attempt must "spin", which wastes CPU cycles
   - threads get scheduled and de-scheduled while the lock is still locked

2. Using locks correctly is hard
   - hard to ensure all race conditions are eliminated
   - easy to introduce synchronization bugs (deadlock, livelock)

Blocking Lock (aka mutex)

- Initial state of lock is 0 ("available")

- acquire(&lock)
  - while value == 1, block (**suspend thread**)
  - when value == 0, set value to 1

```
acquire(&lock){
  while(lock == 1){
    ;
  }
  lock == 1
}
```

- release(&lock)
  - set value to 0
  - **resume a thread waiting on lock (if any)**

```
release(&lock){
  lock == 0
}
```

Example: Bounded Buffers

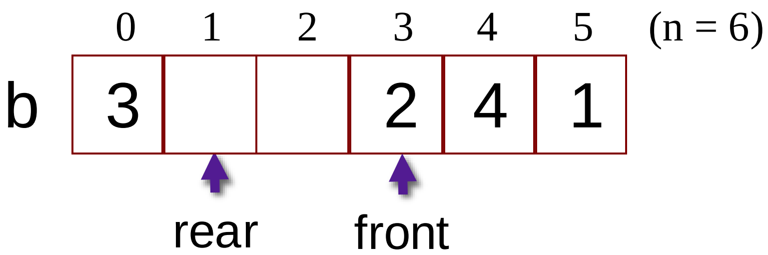
finite capacity (e.g. 20 loaves)
implemented as a queue

1. How do you implement a bounded buffer (0 <= count <= n)?
2. How do you synchronize concurrent access to a bounded buffer?

Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

Example: Bounded Buffers



Values wrap around!!

Exercise: What can go wrong?

```
typedef struct {
    int* b;        // ptr to buffer containing the queue
    int n;         // length of array (max # slots)
    int count;     // number of elements in array
    int front;     // index of first element, 0 <= front < n
    int rear;      // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```
void put(bbuf_t* ptr, int val){
  ptr->b[ptr->rear]= val;
  ptr->rear= ((ptr->rear)+1)%(ptr->n);
  ptr->count++;
}
```

```
int get(bbuf_t* ptr){
  int val= ptr->b[ptr->front];
  ptr->front= ((ptr->front)+1)%(ptr->n);
  ptr->count--;
  return val;
}
```

```
void init(bbuf_t* ptr, int n){
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->count = 0;
  ptr->front = 0;
  ptr->rear = 0;
}
```

Example: Bounded Buffers

	0	1	2	3	4	5	(n = 6)
b	3	2			4	1	
			↑ rear		↑ front		

```
typedef struct {
    int* b;
    int n;
    int count;
    int front;
    int rear;
   pthread_mutex_t lock;
} bbuf_t
```

```
void init(bbuf_t* ptr, int n){
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->count = 0;
  ptr->front = 0;
  ptr->rear = 0;
  init(&(ptr->lock));
}
```

```
void put(bbuf_t* ptr, int val){
  acquire(&(ptr->lock))
  while(ptr->count==ptr->n){
    release(&lock)
    acquire(&lock)
  }
  ptr->b[ptr->rear]= val;
  ptr->rear= ((ptr->rear)+1)%(ptr->n);
  ptr->count++;
  release(&(ptr->lock))
}
```

```
int get(bbuf_t* ptr){
  acquire(&(ptr->lock))
  while(ptr->count==0){
    release(&lock)
    acquire(&lock)
  }
  int val= ptr->b[ptr->front];
  ptr->front= ((ptr->front)+1)%(ptr->n);
  ptr->count--;
  release(&(ptr->lock))
  return val;
```

Condition Variables

- A condition variable cv is a stateless synchronization primitive that is used in combination with locks (mutexes)
  - condition variables allow threads to efficiently wait for a change to the shared state protected by the lock
  - a condition variable is comprised of a waitlist

- Interface:
  - **wait(CV* cv, Lock* lock):** Atomically releases the lock, suspends execution of the calling thread, and places that thread on cv's waitlist; after the thread is awoken, it re-acquires the lock before wait returns
  - **signal(CV* cv):** takes one thread off of cv's waitlist and marks it as eligible to run. (No-op if waitlist is empty.)

Example: Bounded Buffers

	0	1	2	3	4	5	(n = 6)
b	3	2			4	1	
			↑ rear		↑ front		

```
typedef struct {
    int* b;
    int n;
    int count;
    int front;
    int rear;
   pthread_mutex_t lock;
   CV bread_bought;
   CV bread_added;
} bbuf_t
void init(bbuf_t* ptr, int n){
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->count = 0;
  ptr->front = 0;
  ptr->rear = 0;
  init(&(ptr->lock));
  init(&(ptr->bread_bought));
  init(&(ptr->bread_added));
}
```

```
void put(bbuf_t* ptr, int val){
   acquire(&(ptr->lock))
   while(ptr->count == ptr->n)
     wait(&(ptr->bread_bought))
   ptr->b[ptr->rear]= val;
   ptr->rear= ((ptr->rear)+1)%(ptr->n);
   ptr->count++;
   signal(&(ptr->bread_added))
   release(&(ptr->lock))
}
```

```
int get(bbuf_t* ptr){
   acquire(&(ptr->lock))
   while(ptr->count == 0)
     wait(&(ptr->bread_added))
   int val= ptr->b[ptr->front];
   ptr->front= ((ptr->front)+1)%(ptr->n);
   ptr->count--;
   signal(&(ptr->bread_bought))
   release(&(ptr->lock))
   return val;
}
```

Using Condition Variables

1. Declare a lock. Each shared value needs a lock to enforce mutually exclusive access to the shared value.
2. Add code to acquire and release the lock. All code that accesses the shared value must hold the lock.
3. Identify each place something could go wrong if the next line is executed under the wrong circumstances
    - Declare a condition variable that corresponds to when it is safe to proceed with the function.
    - Add a wait for that condition to ensure the critical line is only executed under the right conditions.
    - Add a signal when the condition becomes true.
4. Add loops around your waits. Even though a condition was true when signal was called, it might not still be true when a thread resumes execution.

Exercise: Synchronization Barrier

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data. Once all threads have completed, they can safely use each others results.

What can go wrong?

Use locks and condition variables to synchronize this code.

```
int done_count = 0;
```

```
/* Thread routine */
void* thread(void* args)
{
    parallel_computation(args)

    done_count++;




    use_results();
}
```

Condition Variables

- A condition variable cv is a stateless synchronization primitive that is used in combination with locks (mutexes)
  - condition variables allow threads to efficiently wait for a change to the shared state protected by the lock
  - a condition variable is comprised of a waitlist

- Interface:
  - **wait(CV * cv, Lock * lock):** Atomically releases the lock, suspends execution of the calling thread, and places that thread on cv's waitlist; after the thread is awoken, it re-acquires the lock before wait returns
  - **signal(CV * cv):** takes one thread off of cv's waitlist and marks it as eligible to run. (No-op if waitlist is empty.)
  - **broadcast(CV * cv):** takes all threads off cv's waitlist and marks them as eligible to run. (No-op if waitlist is empty.)

Exercise: Readers/Writers

- Consider a collection of concurrent threads that have access to a shared object
- Some threads are readers, some threads are writers
  - a unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```
int num_readers = 0;
int num_writers = 0;
```

```
int reader(void* shared){



    num_readers++;

    int x =  read(shared);

    num_readers--;


    return x
}
```

```
void writer(void* shared, int val){




    num_writers=1;

    write(shared, val);

    num_writers=0;



}
```

Programming with CVs

C

- Initialization:

```
pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv =
    PTHREAD_COND_INITIALIZER;
```

- Lock acquire/release:

```
pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);
```

- CV operations:

```
pthread_cond_wait(&cv, &lock);
pthread_cond_signal(&cv);
pthread_cond_broadcast(&cv);
```

Python

- Initialization:

```
lock = Lock()
cv = Condition(lock)
```

- Lock acquire/release:

```
lock.acquire()
lock.release()
```

- V

```
cv.wait()
cv.notify()
cv.notify_all()
```