Operating Systems (A)
(Honor Track)

Lecture 16: Synchronization (III)

Yao Guo (郭耀)

Peking University

Fall 2021

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

This Lecture

Advanced Synchronization Topics

Messages and barriers

Concurrency mechanisms in Unix and Linux

Classic synchronization problems

Buzz Words

- Message
- Barrier
- Interprocess communication (IPC)
- Pipe
- Signal
- Memory barrier
- Interlock

This Lecture

Advanced Synchronization Topics

Messages and barriers

Concurrency mechanisms in Unix and Linux

Classic synchronization problems

Messages

- Simple model of communication and synchronization based on atomic transfer of data across a channel
  - send(destination, &message);
  - receive(source, &message);
- Messages for synchronization are straightforward
- Issues
  - Message lost, ack lost
  - Order
  - Naming
  - Authentication
  - …

Producer-Consumer with Message Passing

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}
```

```
void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                  /* get message containing item */
          item = extract_item(&m);                /* extract item from message */
          send(producer, &m);                     /* send back empty reply */
          consume_item(item);                     /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with N messages.

Barriers

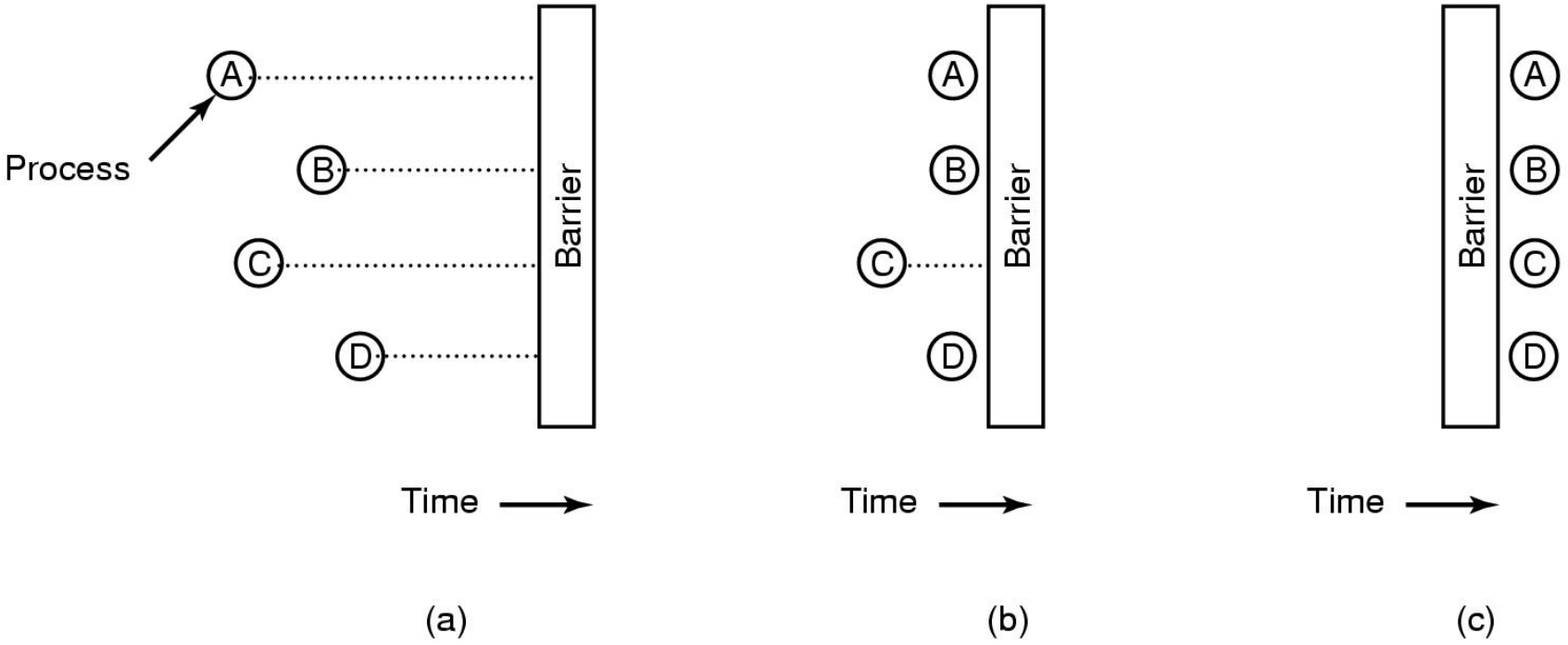


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

This Lecture

Advanced Synchronization Topics

Messages and barriers

Concurrency mechanisms in Unix and Linux

Classic synchronization problems

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocess communication (IPC) and synchronization including:
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals

Pipes

- A circular buffer allowing two processes to communicate on the producer-consumer model
  - first-in-first-out queue, written by one process and read by another.
- Two types:
  - Named: mkfifo my_pipe; rm my_pipe
  - Unnamed

Messages

- A block of bytes with an accompanying type
- UNIX provides ***msgsnd*** and ***msgrcv*** system calls for processes to engage in message passing
- Associated with each process is a message queue, which functions like a mailbox

Shared Memory

- A common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
  - determined on a per-process basis
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

Semaphores

- System V Release 4 (SVR4) uses a generalization of the ***semWait*** and ***semSignal*** primitives
- Associated with the semaphore are queues of processes blocked on that semaphore

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events.
  - Similar to a hardware interrupt, without priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent

Signals Defined for UNIX SVR4

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX plus
  - Atomic operations
  - Spinlocks
  - Semaphores (slightly different to SVR4)
  - Memory barriers

Atomic Operations

- Atomic operations execute without interruption and without interference
- Two types:
  - Integer operations – operating on an integer variable
  - Bitmap operations – operating on one bit in a bitmap

Linux Atomic Operations

Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic_set(atomic_t *v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i, atomic_t *v)`	Subtract i from v
`void atomic_inc(atomic_t *v)`	Add 1 to v
`void atomic_dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise

Linux Atomic Operations

Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr

Spinlock

- Only one thread at a time can acquire a spinlock
  - Any other thread will keep trying (spinning) until it can acquire the lock
- A spinlock is an integer
  - If 0, the thread sets the value to 1 and enters its critical section
  - If the value is nonzero, the thread continually checks the value until it is zero

Linux Spinlocks

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise

Semaphores

- Similar to UNIX SVR4 but also provides an implementation of semaphores for its own use
- Three types of kernel semaphores
  - Binary semaphores
  - Counting semaphores
  - Reader-writer semaphores
    - Allows multiple readers into the critical section but only one writer at any given time.
    - "struct rw_semaphore"

Linux Kernel Semaphores

Traditional Semaphores	
`void sema_init(struct semaphore *sem, int count)`	Initializes the dynamically created semaphore to the given count
`void init_MUTEX(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
`void init_MUTEX_LOCKED(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 0 (initially locked)
`void down(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
`int down_interruptible(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received.
`int down_trylock(struct semaphore *sem)`	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
`void up(struct semaphore *sem)`	Releases the given semaphore
Reader-Writer Semaphores	
`void init_rwsem(struct rw_semaphore, *rwsem)`	Initalizes the dynamically created semaphore with a count of 1
`void down_read(struct rw_semaphore, *rwsem)`	Down operation for readers
`void up_read(struct rw_semaphore, *rwsem)`	Up operation for readers
`void down_write(struct rw_semaphore, *rwsem)`	Down operation for writers
`void up_write(struct rw_semaphore, *rwsem)`	Up operation for writers

Memory Barriers

- To enforce the order in which instructions are executed, Linux provides the memory barrier facility

Table 6.6 Linux Memory Barrier Operations

`rmb()`	Prevents loads from being reordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents loads and stores from being reordered across the barrier
`Barrier()`	Prevents the compiler from reordering loads or stores across the barrier
`smp_rmb()`	On SMP, provides a `rmb()` and on UP provides a `barrier()`
`smp_wmb()`	On SMP, provides a `wmb()` and on UP provides a `barrier()`
`smp_mb()`	On SMP, provides a `mb()` and on UP provides a `barrier()`

SMP = symmetric multiprocessor
UP = uniprocessor

This Lecture

Advanced Synchronization Topics

Messages and barriers

Concurrency mechanisms in Unix and Linux

Classic synchronization problems

Classic Synchronization Problems

- *Bounded Buffer* problem
- *Dining Philosophers* problem
- *Sleeping Barber* problem

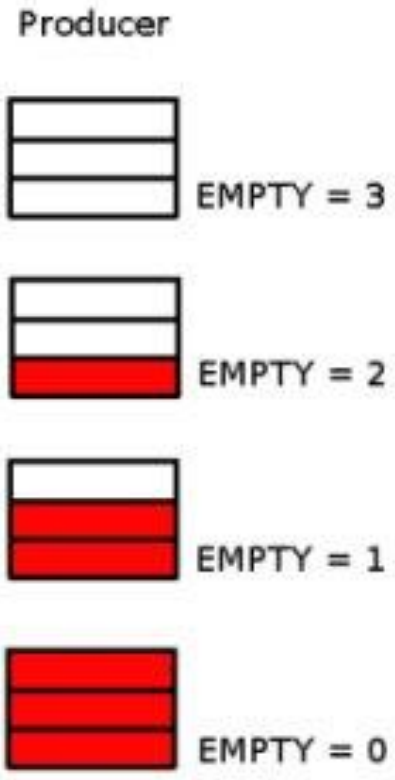
Bounded Buffer Problem (1/5)

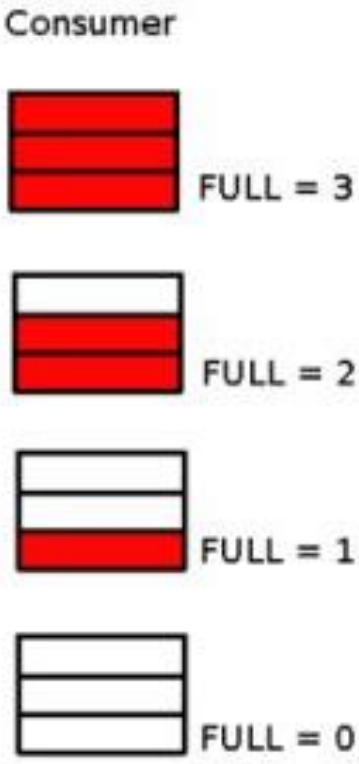
- Problem: there is a set of resource buffers shared by producer and consumer threads
  - Producer inserts resources into the buffer set
    - Output, disk blocks, memory pages, processes, etc.
  - Consumer removes resources from the buffer set
    - Whatever is generated by the producer
- Producer and consumer execute at different rates
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff
- Safety:
  - If *nc* is number consumed, *np* number produced, and N the size of the buffer, then $0 \leq np - nc \leq N$

Bounded Buffer Problem (2/5)

- Use three semaphores:
  - empty – count of empty buffers
    - Counting semaphore
    - empty = N – (np-nc)
  - full – count of full buffers
    - Counting semaphore
    - *np* - *nc* = full
  - mutex – mutual exclusion to shared set of buffers
    - Binary semaphore

Bounded Buffer Problem (3/5)





Producer decrements EMPTY and blocks when buffer is full since the semaphore is at 0

Consumer decrements FULL and Blocks when buffer has no item Since the semaphore FULL is at 0

Bounded Buffer Problem (4/5)

```
Semaphore mutex = 1;   // mutual exclusion to shared set of buffers
Semaphore empty = N;   // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)
```

```
producer {
  while (1) {
    Produce new resource;
    P(empty); // wait for empty buffer
    P(mutex); // lock buffer list
    Add resource to an empty buffer;
    V(mutex); // unlock buffer list
    V(full);  // note a full buffer
  }
}
```

```
consumer {
  while (1) {
    P(full);  // wait for a full buffer
    P(mutex); // lock buffer list
    Remove resource from a full buffer;
    V(mutex); // unlock buffer list
    V(empty); // note an empty buffer
    Consume resource;
  }
}
```

Bounded Buffer Problem (5/5)

- Why need the mutex at all?
- Where are the critical sections?

- What happens if operations on mutex and full/empty are switched around?
  - The pattern of P/V on full/empty is a common construct often called an interlock
- Why V(full) and V(empty)?

- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems

Dining Philosophers: An Intellectual Game

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- Possible deadlock?
- How to prevent deadlock?

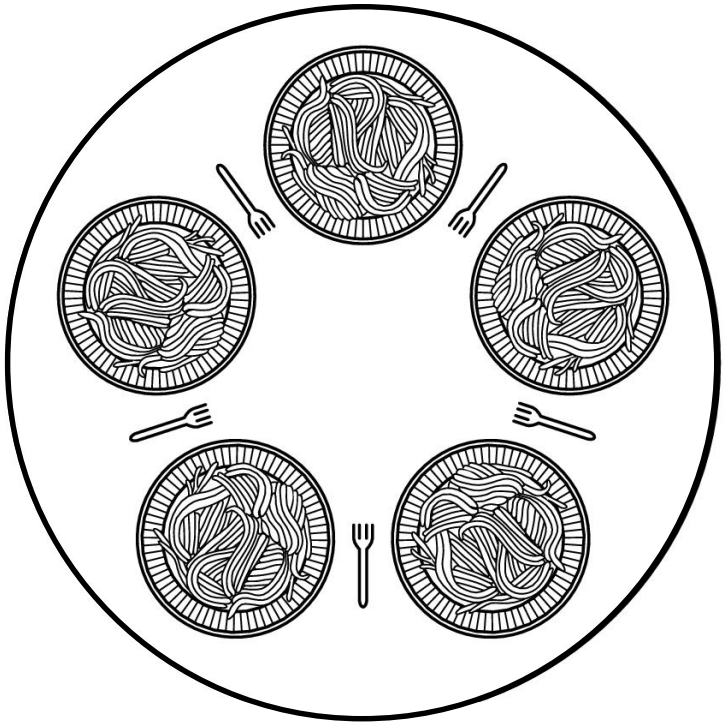


Figure 2-44. Lunch time in the Philosophy Department.

Does It Solve the Dining Philosophers Problem?


```
#define N 5                                   /* number of philosophers */

void philosopher(int i)                       /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                     /* philosopher is thinking */
                take_fork(i);                 /* take left fork */
                take_fork((i+1) % N);         /* take right fork; % is modulo operator */
                eat( );                       /* yum-yum, spaghetti */
                put_fork(i);                  /* put left fork back on the table */
                put_fork((i+1) % N);          /* put right fork back on the table */
        }
}
```

Figure 2-45. A nonsolution to the dining philosophers problem.

Dining Philosophers Solution (1/2)

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N−1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                          /* repeat forever */
        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 */
    }
}
```

Figure 2-46. A solution to the dining philosophers problem.

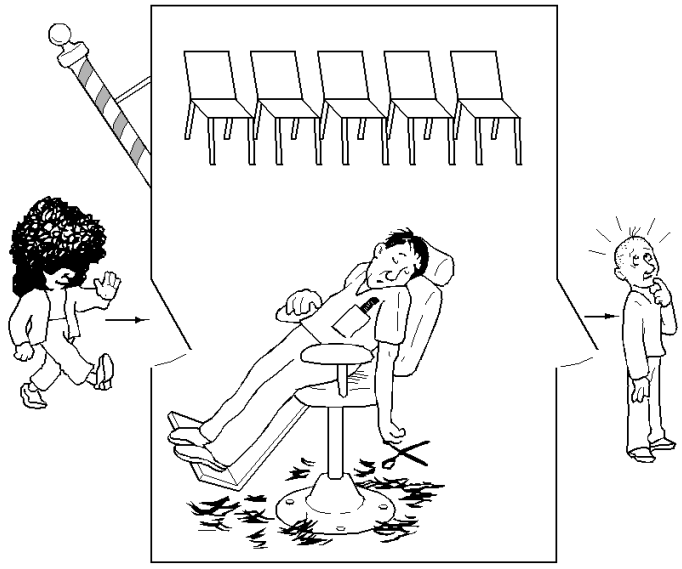
Dining Philosophers Solution (2/2)

```
void take_forks(int i)                    /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                       /* enter critical region */
      state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
      test(i);                            /* try to acquire 2 forks */
      up(&mutex);                         /* exit critical region */
      down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                         /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                       /* enter critical region */
      state[i] = THINKING;                /* philosopher has finished eating */
      test(LEFT);                         /* see if left neighbor can now eat */
      test(RIGHT);                        /* see if right neighbor can now eat */
      up(&mutex);                         /* exit critical region */
}

void test(i)                              /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Figure 2-46. A solution to the dining philosophers problem.

The Sleeping Barber Problem

- *N* customer chairs
  - Customer waiting if there is a free chair
  - Otherwise leave the shop
- One barber
  - can cut one customer's hair at any time
  - If no customer, goes to sleep



The Sleeping Barber Solution 1/3

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */
```

The Sleeping Barber Solution 2/3

```
void barber(void)
{
    while (TRUE) {
        down(&customers);          /* go to sleep if # of customers is 0 */
        down(&mutex);              /* acquire access to 'waiting' */
        waiting = waiting – 1;     /* decrement count of waiting customers */
        up(&barbers);              /* one barber is now ready to cut hair */
        up(&mutex);                /* release 'waiting' */
        cut_hair( );               /* cut hair (outside critical region) */
    }
}
```

The Sleeping Barber Solution 3/3

```
void customer(void)
{
      down(&mutex);                    /* enter critical region */
      if (waiting < CHAIRS) {          /* if there are no free chairs, leave */
            waiting = waiting + 1;     /* increment count of waiting customers */
            up(&customers);            /* wake up barber if necessary */
            up(&mutex);                /* release access to 'waiting' */
            down(&barbers);            /* go to sleep if # of free barbers is 0 */
            get_haircut( );            /* be seated and be serviced */
      } else {
            up(&mutex);                /* shop is full; do not wait */
      }
}
```

Summary

- Messages and barriers
- Concurrency mechanisms in Unix and Linux
- Classic synchronization problems

- Next lecture: Deadlocks