

ANNOUNCEMENTS

Exam 2: Monday 10/26 7:15 – 9:15 Ingraham B10

- Fill out form on webpage if have academic conflict
- Covers all of Concurrency Piece (lecture and book)
 - Light on chapter 29, nothing from chapter 33
 - Very few questions from Virtualization Piece (< 10%)
- Multiple choice (fewer pure true/false)
- Look at two concurrency homeworks

Project 3: New project partners matched

- Only xv6 part; watch two videos early
- Due Wed 10/28

Today's Reading: Chapter 30

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

LOCKS AND CONDITION VARIABLES

Questions answered in this lecture:

How can threads **block** instead of **spin-waiting** while waiting for a lock?

When should a waiting thread block and when should it spin?

How can threads enforce **ordering** across operations?

How can **thread_join()** be implemented?

How can **condition** variables be used to support **producer/consumer** apps?

REVIEW: TICKET LOCK

```

typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn)
        yield();
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

```

Are both FAA() instructions needed or can replace with simple ++?

REVIEW: TICKET LOCK

```

typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn)
        yield();
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

```

FAA() used in textbook → conservative
Try this modification in Homework simulations

LOCK EVALUATION

After correctness,
how to tell if a lock implementation is good?

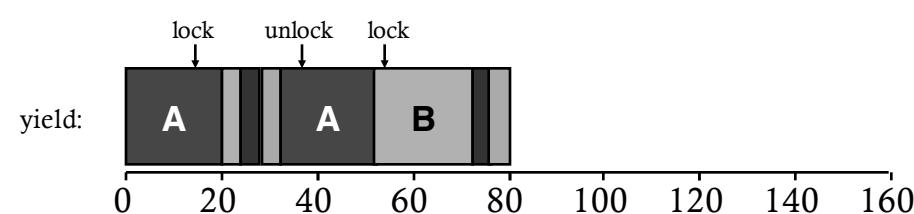
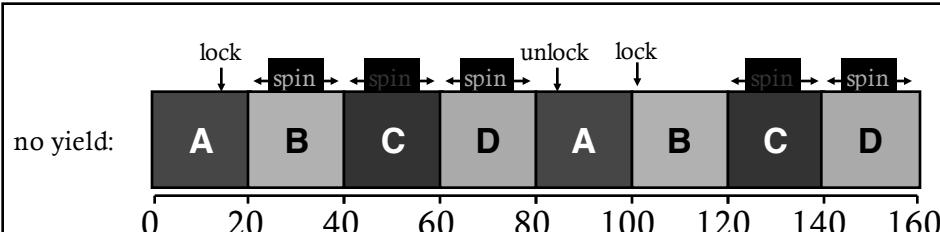
Fairness:

- Do processes acquire lock in same order as requested?

Performance

Two scenarios:

- low contention (fewer threads, lock usually available)
- high contention (many threads per CPU, each contending)



Why is yield useful?
Why doesn't yield solve all performance problems?

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

Lock implementation removes waiting threads from scheduler ready queue (e.g., park() and unpark())

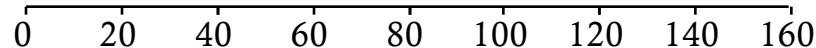
Scheduler runs any thread that is **ready**

Good separation of concerns

RUNNABLE: A, B, C, D

RUNNING: <empty>

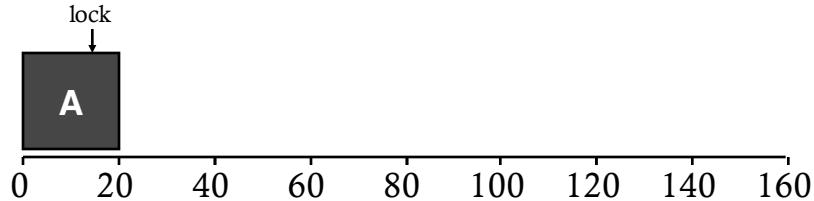
WAITING: <empty>



RUNNABLE: B, C, D

RUNNING: A

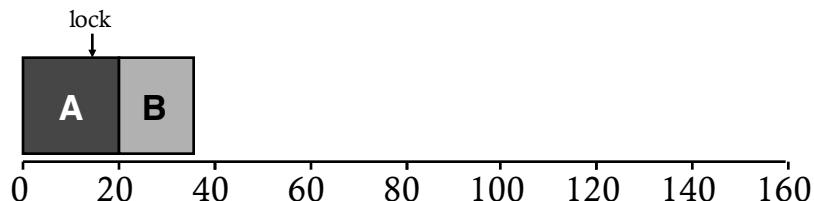
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING: B

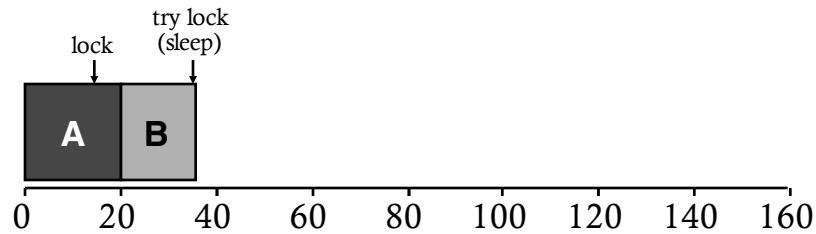
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING:

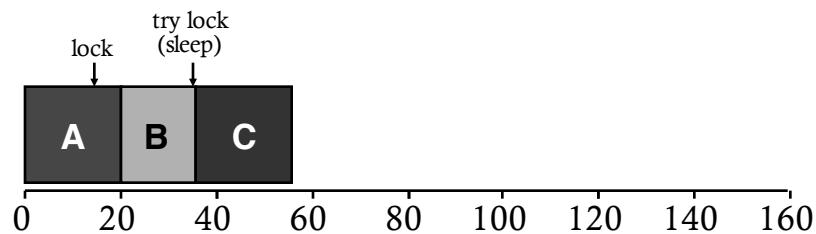
WAITING: B



RUNNABLE: D, A

RUNNING: C

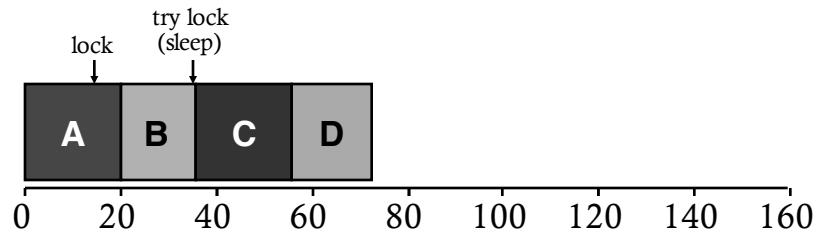
WAITING: B



RUNNABLE: A, C

RUNNING: D

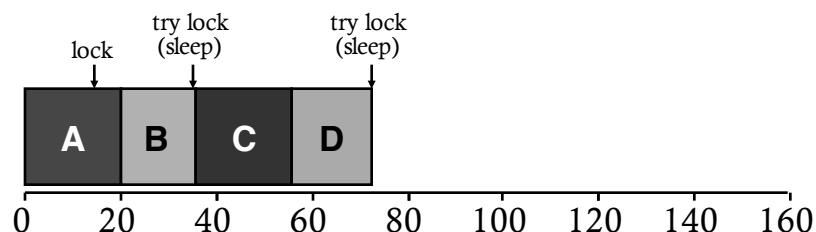
WAITING: B



RUNNABLE: A, C

RUNNING:

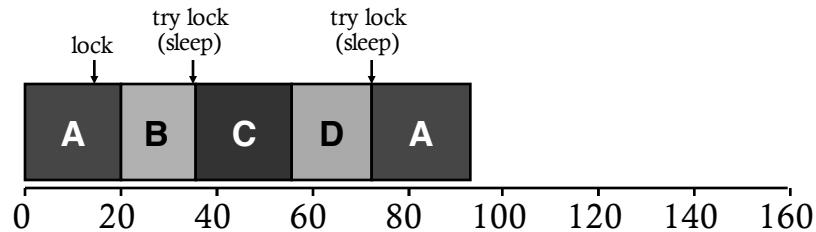
WAITING: B, D



RUNNABLE: C

RUNNING: A

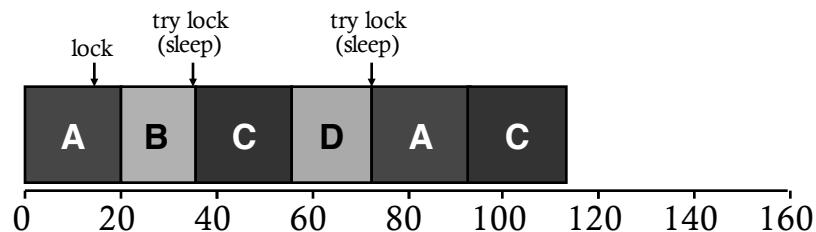
WAITING: B, D



RUNNABLE: A

RUNNING: C

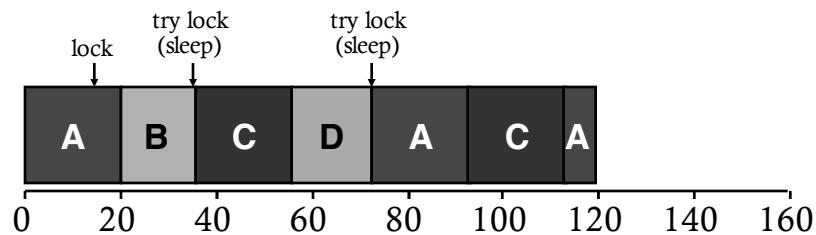
WAITING: B, D



RUNNABLE: C

RUNNING: A

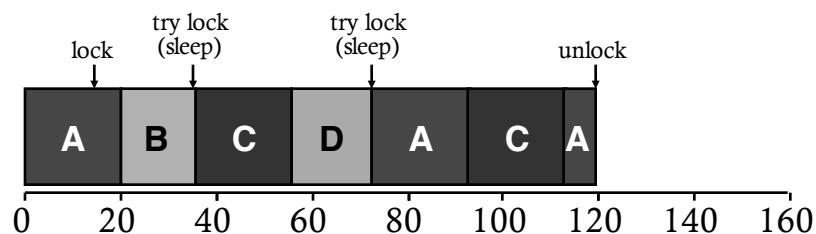
WAITING: B, D



RUNNABLE: B, C

RUNNING: A

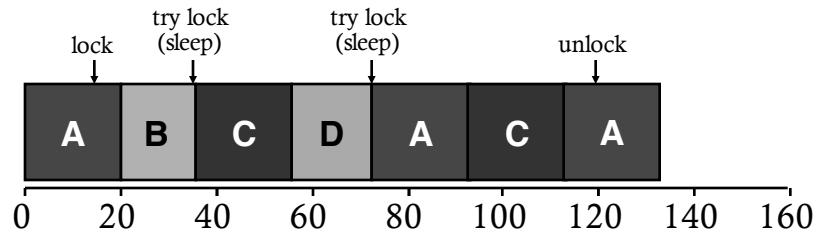
WAITING: D



RUNNABLE: B, C

RUNNING: A

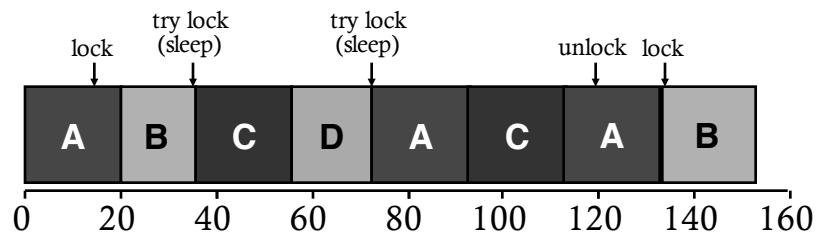
WAITING: D



RUNNABLE: C, A

RUNNING: B

WAITING: D



LOCK IMPLEMENTATION: BLOCK WHEN WAITING

```

typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;

void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;
        park(); // blocked
    } else {
        l->lock = true;
        l->guard = false;
    }
}

(a) Why is guard used?
(b) Why okay to spin on guard?
(c) In release(), why not set lock=false
when unpark?
(d) What is the race condition?

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}

```

RACE CONDITION

Thread 1 (in lock)	Thread 2 (in unlock)
<pre> if (l->lock) { qadd(l->q, tid); l->guard = false; park(); // block } </pre>	<pre> while (TAS(&l->guard, true)); if (qempty(l->q)) // false!! else unpark(qremove(l->q)); l->guard = false; </pre>

Problem: Guard not held when call park()
 Unlocking thread may unpark() before other park()

BLOCK WHEN WAITING: FINAL CORRECT LOCK

```

Typedef struct {
    void acquire(LockT *l) {
        bool lock = false;
        while (TAS(&l->guard, true));
        if (l->lock) {
            qadd(l->q, tid);
            setpark(); // notify of plan
            l->guard = false;
            park(); // unless unpark()
        } else {
            l->lock = true;
            l->guard = false;
        }
    }
}

setpark() fixes race condition
}
}

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
}

```

SPIN-WAITING VS BLOCKING

Each approach is better under different circumstances

Uniprocessor

Waiting process is scheduled --> Process holding lock isn't

Waiting process should always relinquish processor

Associate queue of waiters with each lock (as in previous implementation)

Multiprocessor

Waiting process is scheduled --> Process holding lock might be

Spin or block depends on how long, t, before lock is released

Lock released quickly --> Spin-wait

Lock released slowly --> Block

Quick and slow are relative to context-switch cost, C

WHEN TO SPIN-WAIT? WHEN TO BLOCK?

If know how long, t , before lock released, can determine optimal behavior

How much CPU time is wasted when spin-waiting?

t

How much wasted when block?

C

What is the best action when $t < C$?

spin-wait

When $t > C$?

block

Problem:

Requires knowledge of future; too much overhead to do any special prediction

TWO-PHASE WAITING

Theory: Bound worst-case performance; ratio of actual/optimal

When does worst-possible performance occur?

Spin for very long time $t \gg C$

Ratio: t/C (unbounded)

Algorithm: Spin-wait for C then block --> Factor of 2 of optimal

Two cases:

$t < C$: optimal spin-waits for t ; we spin-wait t too

$t > C$: optimal blocks immediately (cost of C); we pay spin C then block
(cost of $2C$); $2C / C \rightarrow 2$ -competitive algorithm

Example of competitive analysis

CONDITION VARIABLES

CONCURRENCY OBJECTIVES

Mutual exclusion (e.g., A and B don't run at same time)

- solved with *locks*

Ordering (e.g., B runs after A does something)

- solved with *condition variables* and *semaphores*

ORDERING EXAMPLE: JOIN

```

pthread_t p1,p2;
Pthread_create(&p1,NULL,mythread,"A");
Pthread_create(&p2,NULL,mythread,"B");
// join waits for the threads to finish
Pthread_join(p1,NULL);           how to implement join()?
Pthread_join(p2,NULL);
printf("main: done\n [balance: %d]\n [should: %d]\n",
       balance, max*2);
return 0;

```

CONDITION VARIABLES

Condition Variable: queue of waiting threads

B waits for a signal on CV before running

- wait(CV, ...)

A sends signal to CV when time for **B** to run

- signal(CV, ...)

CONDITION VARIABLES

wait(cond_t *cv, mutex_t *lock)

- assumes the lock is held when wait() is called
 - puts caller to sleep + releases the lock (atomically)
 - 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

JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```
void thread_join() {
    Mutex_lock(&m);           // x
    Cond_wait(&c, &m);        // y
    Mutex_unlock(&m);         // z
}
```

Child:

```
void thread_exit() {
    Mutex_lock(&m);           // a
    Cond_signal(&c);          // b
    Mutex_unlock(&m);         // c
```

Example schedule:

Parent: x y z

Child: a b c

Works!

JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```
void thread_join() {
    Mutex_lock(&m);      // x
    Cond_wait(&c, &m);   // y
    Mutex_unlock(&m);   // z
}
```

Child:

```
void thread_exit() {
    Mutex_lock(&m);      // a
    Cond_signal(&c);     // b
    Mutex_unlock(&m);   // c
}
```

Can you construct ordering that does not work?

Example broken schedule:

Parent:	x	y
---------	---	---

Child:	a	b
--------	---	---

Parent waits forever!

RULE OF THUMB 1

Keep state in addition to CV's!

CV's are used to signal threads when state changes

If state is already as needed, thread doesn't wait for a signal!

JOIN IMPLEMENTATION: ATTEMPT 2

Parent:

```
void thread_join() {
    Mutex_lock(&m);      // w
    if(done == 0)          // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m);    // z
}
```

Child:

```
void thread_exit() {
    done = 1;           // a
    Cond_signal(&c);   // b
}
```

Fixes previous broken ordering:

Parent:

w x y z

Child:

a b

JOIN IMPLEMENTATION: ATTEMPT 2

Parent:

```
void thread_join() {
    Mutex_lock(&m);      // w
    if(done == 0)          // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m);    // z
}
```

Child:

```
void thread_exit() {
    done = 1;           // a
    Cond_signal(&c);   // b
}
```

Can you construct ordering that does not work?

Parent: w x

y ... sleep forever ...

Child: a b

JOIN IMPLEMENTATION: CORRECT

Parent:

```
void thread_join() {
    Mutex_lock(&m);      // w
    if(done == 0)          // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m);    // z
}
```

Child:

```
void thread_exit() {
    Mutex_lock(&m);      // a
    done = 1;              // b
    Cond_signal(&c);     // c
    Mutex_unlock(&m);    // d
}
```

Parent: w x y z

Child: a b c

Use mutex to ensure no race between interacting with state
and wait/signal

PRODUCER/CONSUMER PROBLEM

EXAMPLE: UNIX PIPES

A pipe may have many writers and readers

Internally, there is a finite-sized buffer

Writers add data to the buffer

Writers have to wait if buffer is full

Readers remove data from the buffer

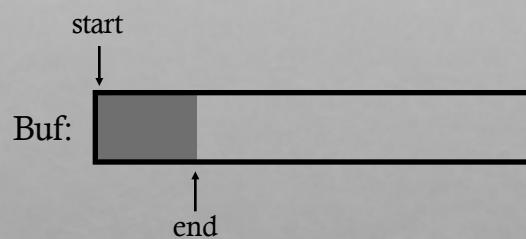
Readers have to wait if buffer is empty

EXAMPLE: UNIX PIPES

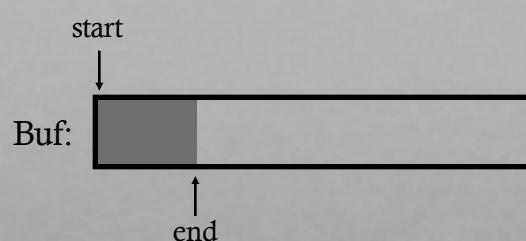


EXAMPLE: UNIX PIPES

write!

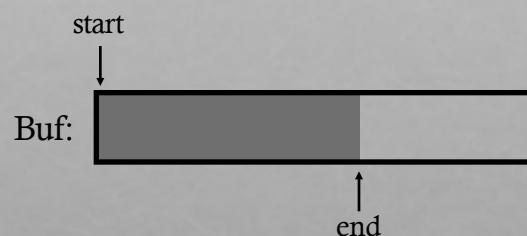


EXAMPLE: UNIX PIPES

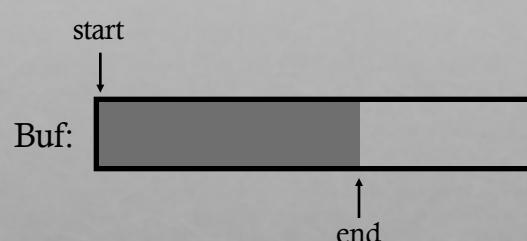


EXAMPLE: UNIX PIPES

write!

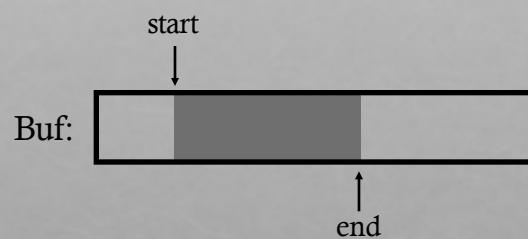


EXAMPLE: UNIX PIPES

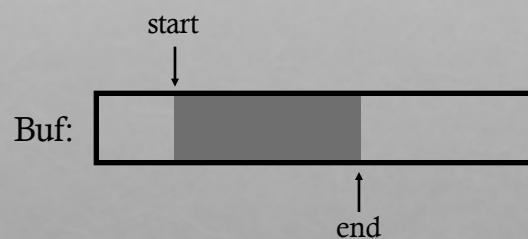


EXAMPLE: UNIX PIPES

read!

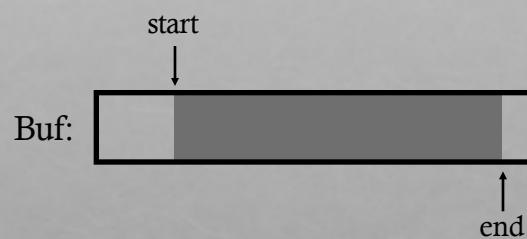


EXAMPLE: UNIX PIPES

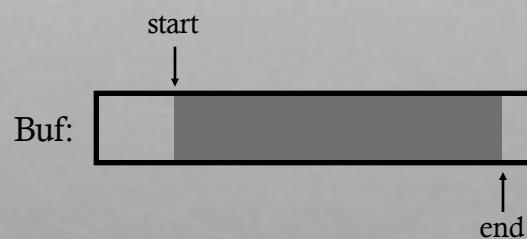


EXAMPLE: UNIX PIPES

write!

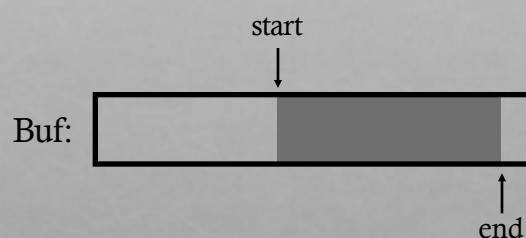


EXAMPLE: UNIX PIPES

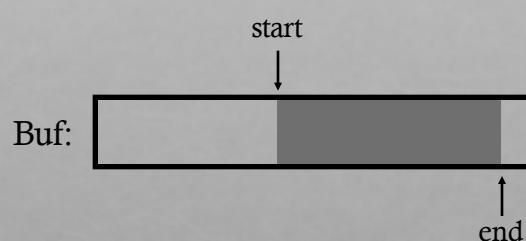


EXAMPLE: UNIX PIPES

read!

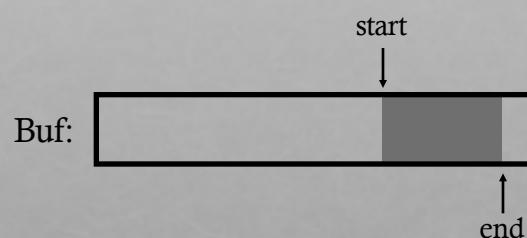


EXAMPLE: UNIX PIPES

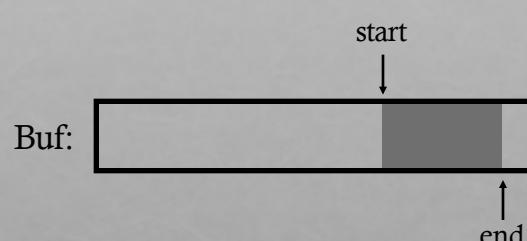


EXAMPLE: UNIX PIPES

read!



EXAMPLE: UNIX PIPES



EXAMPLE: UNIX PIPES

read!

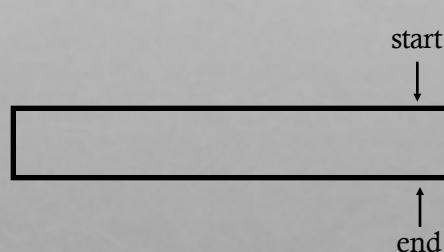
Buf:



EXAMPLE: UNIX PIPES

read!

Buf:



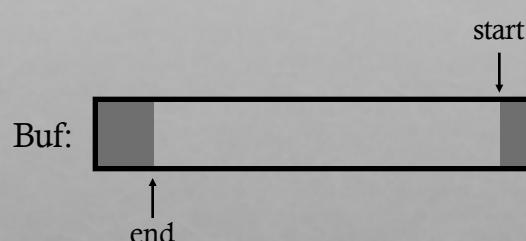
note: readers must wait

EXAMPLE: UNIX PIPES

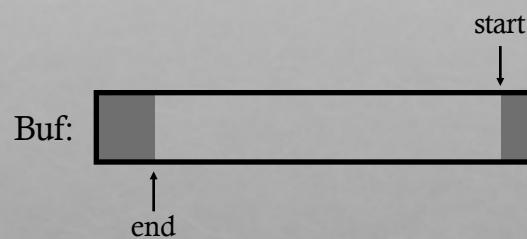


EXAMPLE: UNIX PIPES

write!

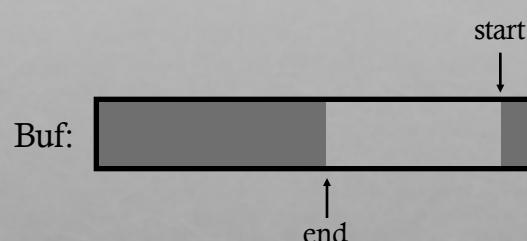


EXAMPLE: UNIX PIPES

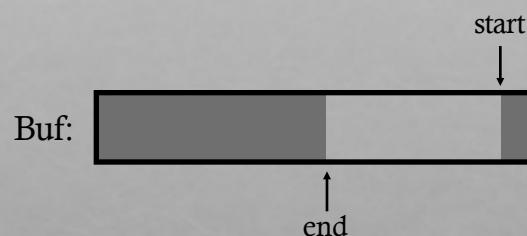


EXAMPLE: UNIX PIPES

write!



EXAMPLE: UNIX PIPES



EXAMPLE: UNIX PIPES

write!



EXAMPLE: UNIX PIPES

write!

Buf:



note: writers must wait

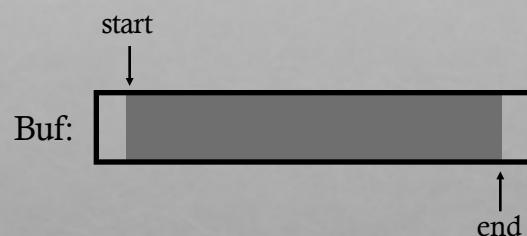
EXAMPLE: UNIX PIPES

Buf:



EXAMPLE: UNIX PIPES

read!



EXAMPLE: UNIX PIPES

Implementation:

- reads/writes to buffer require locking
- when buffers are full, writers must wait
- when buffers are empty, readers must wait

PRODUCER/CONSUMER PROBLEM

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems

- Web servers

General strategy use condition variables to:

make producers wait when buffers are full

make consumers wait when there is nothing to consume

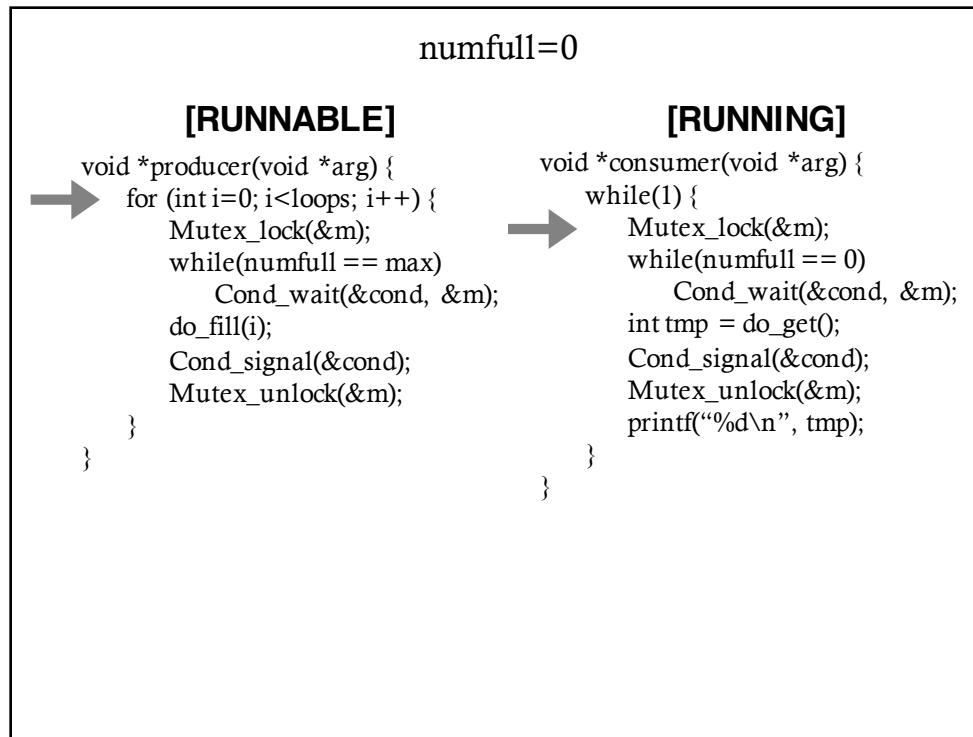
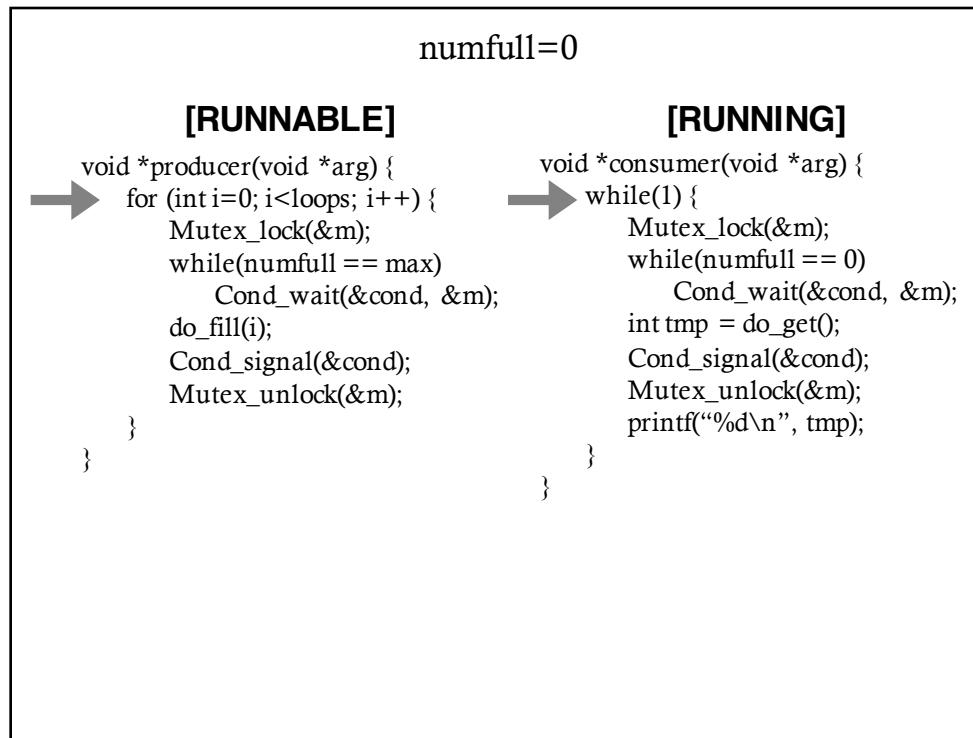
PRODUCE/CONSUMER EXAMPLE

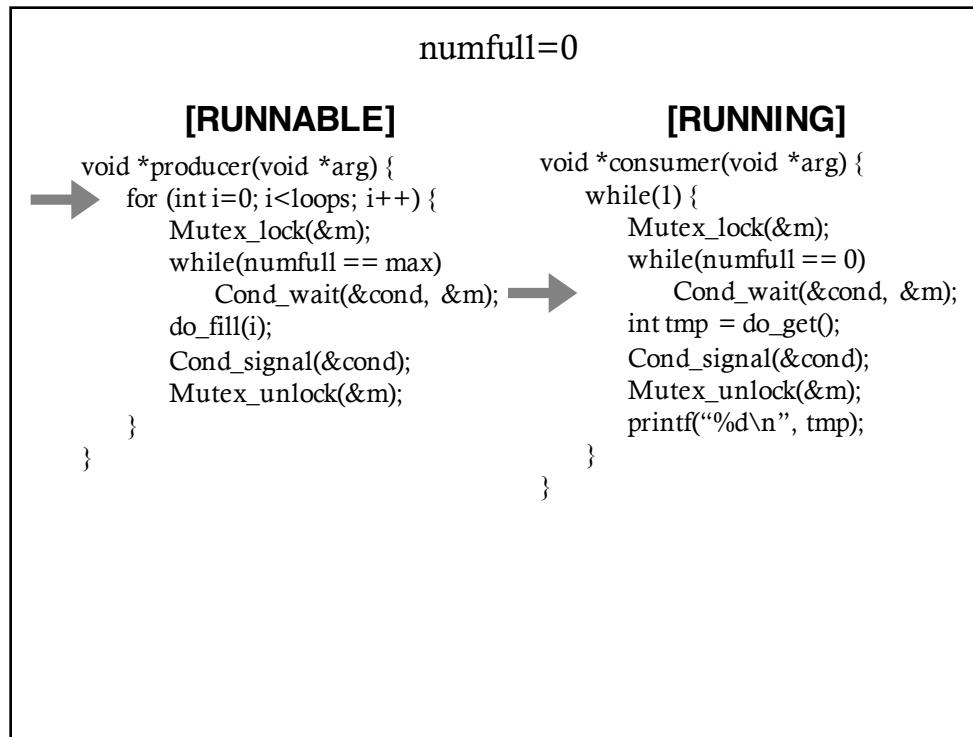
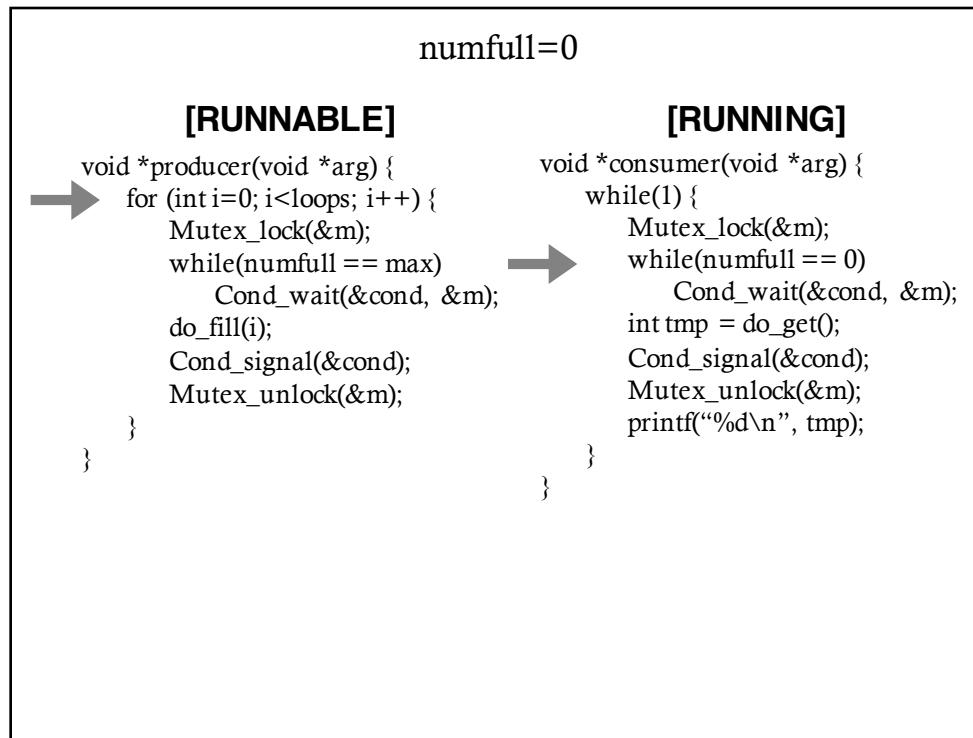
Start with easy case:

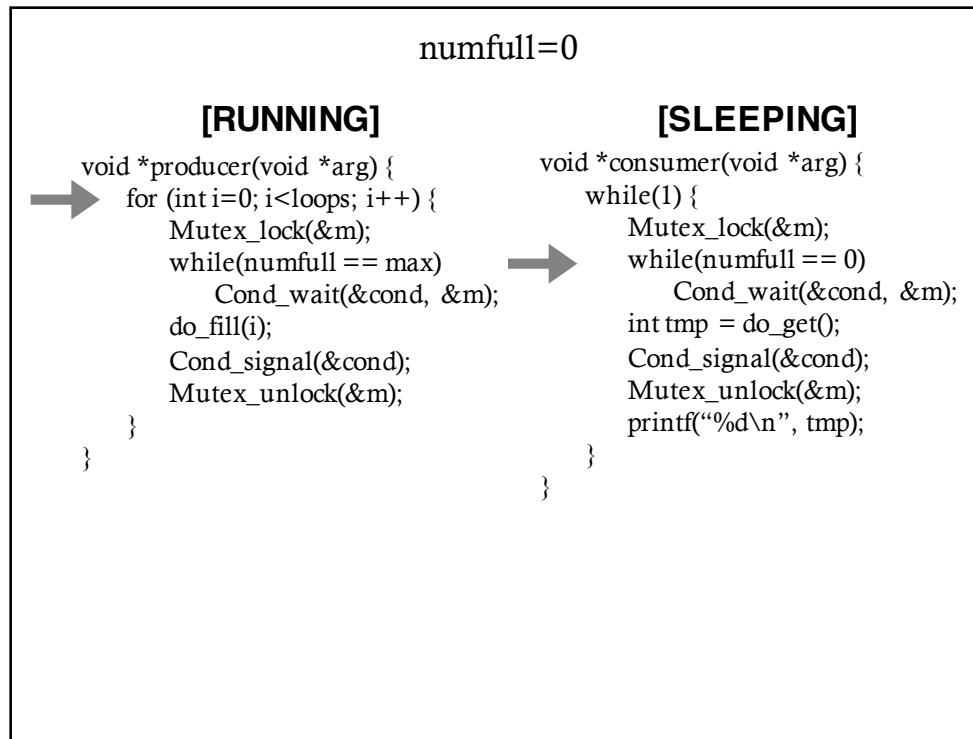
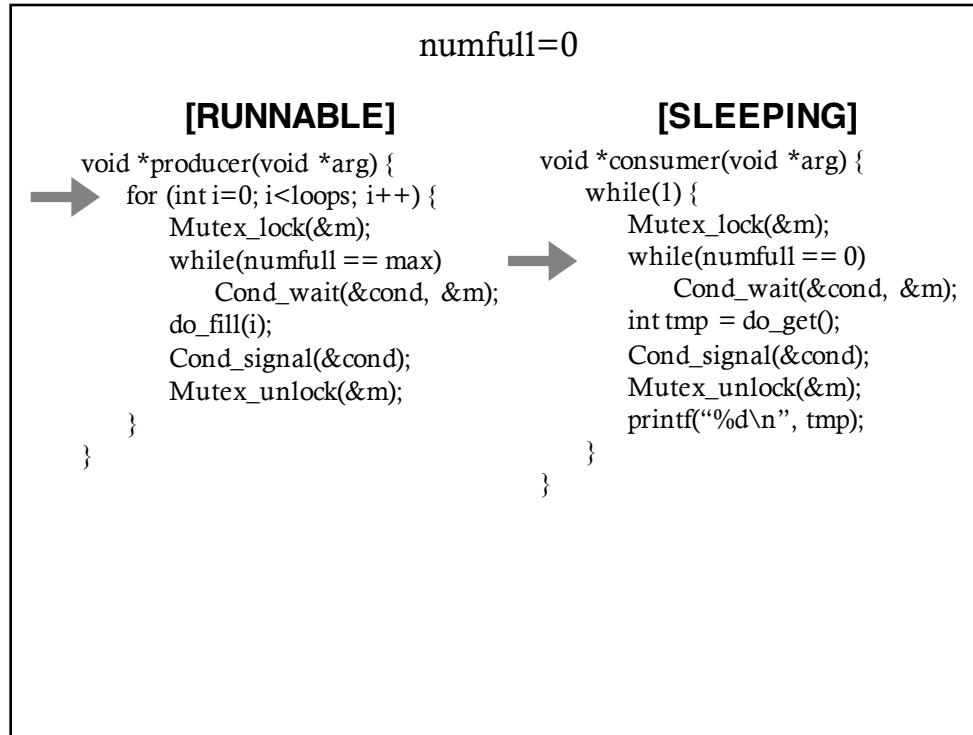
- 1 producer thread
- 1 consumer thread
- 1 shared buffer to fill/consume (max = 1)

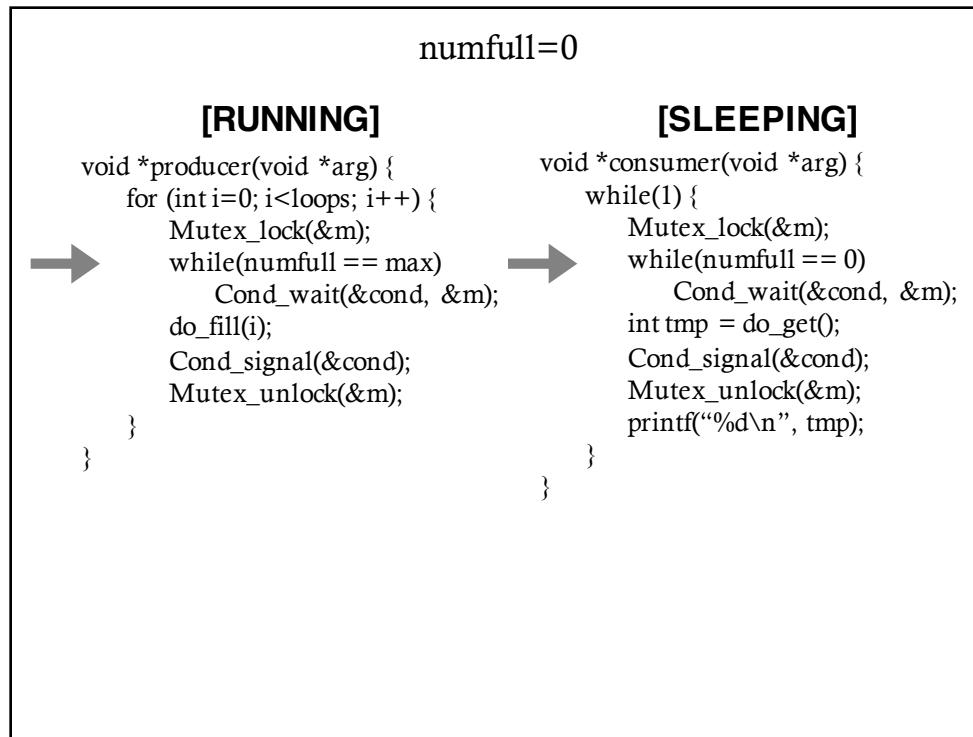
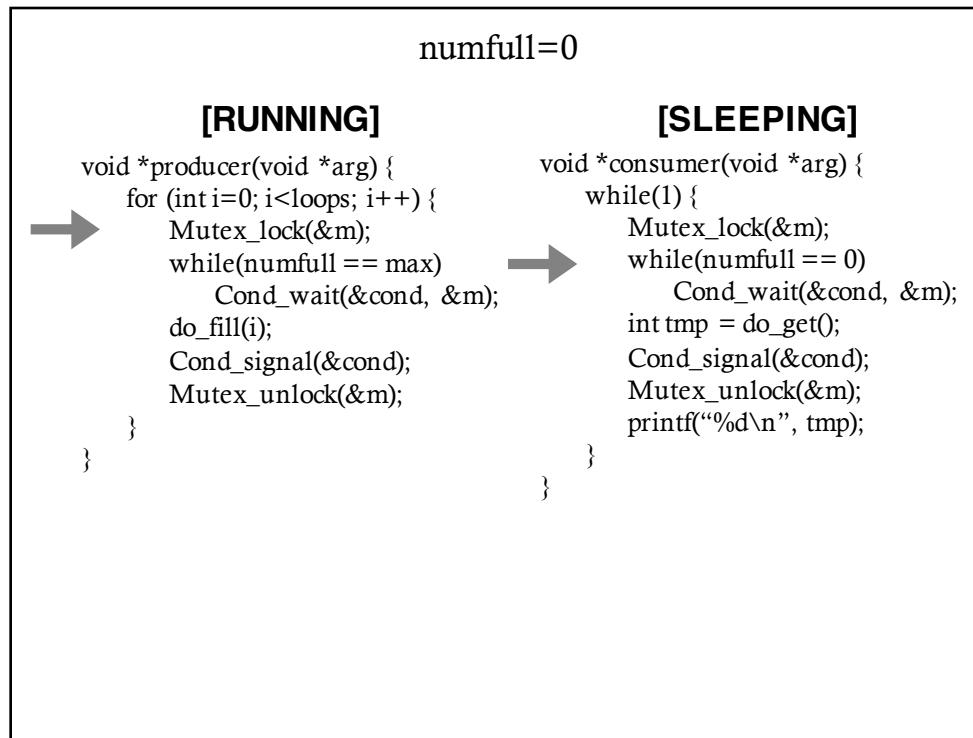
Numfill = number of buffers currently filled

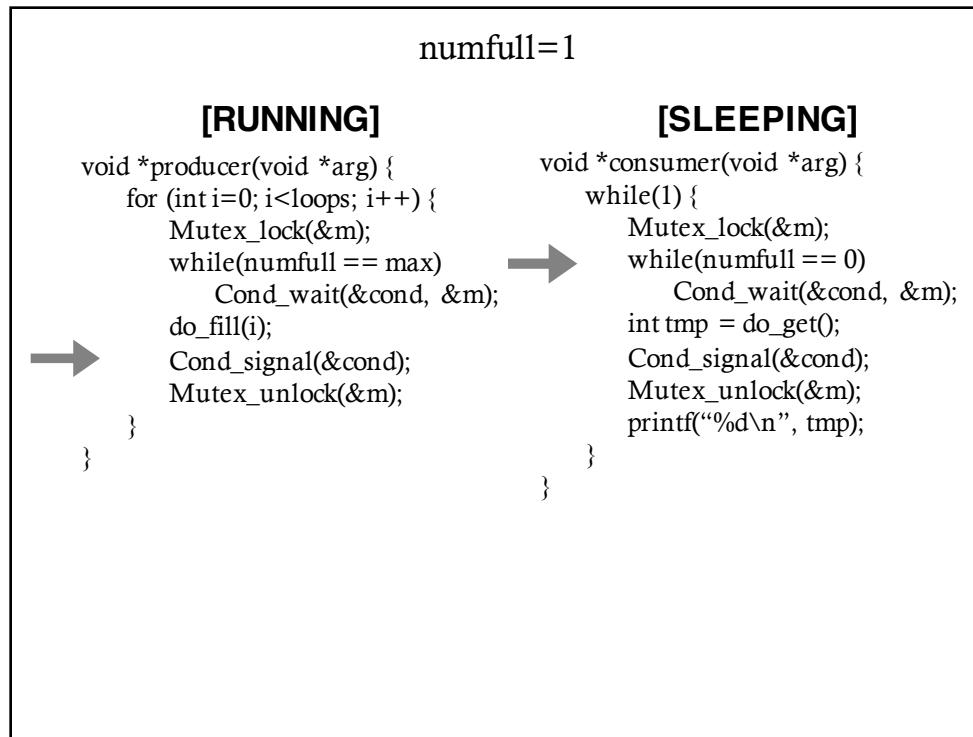
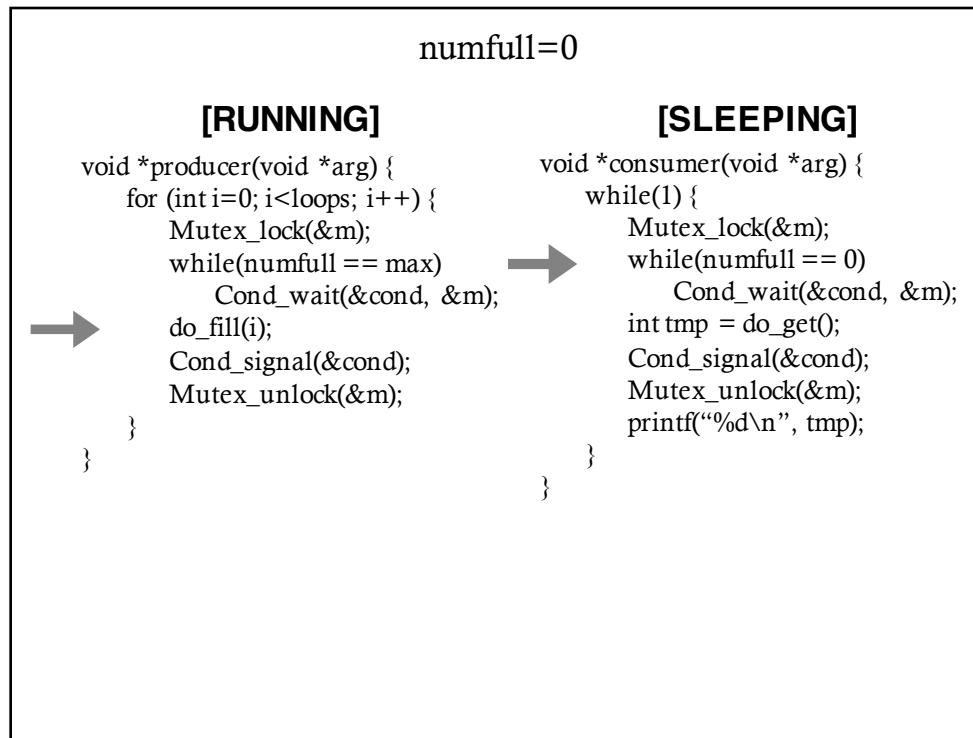
Examine slightly broken code to begin...

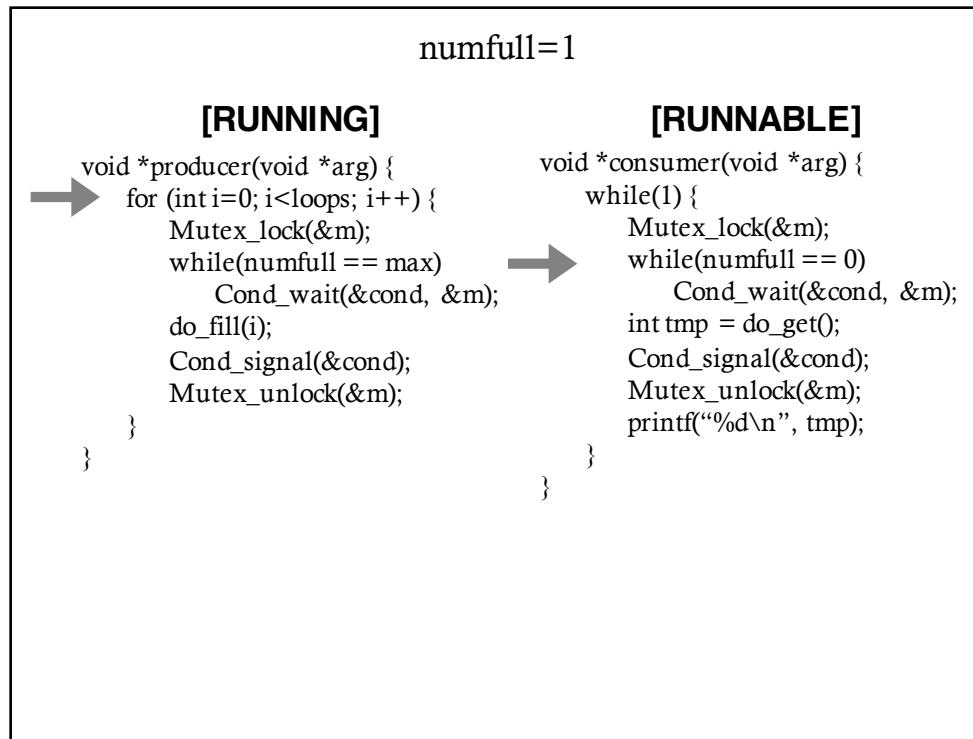
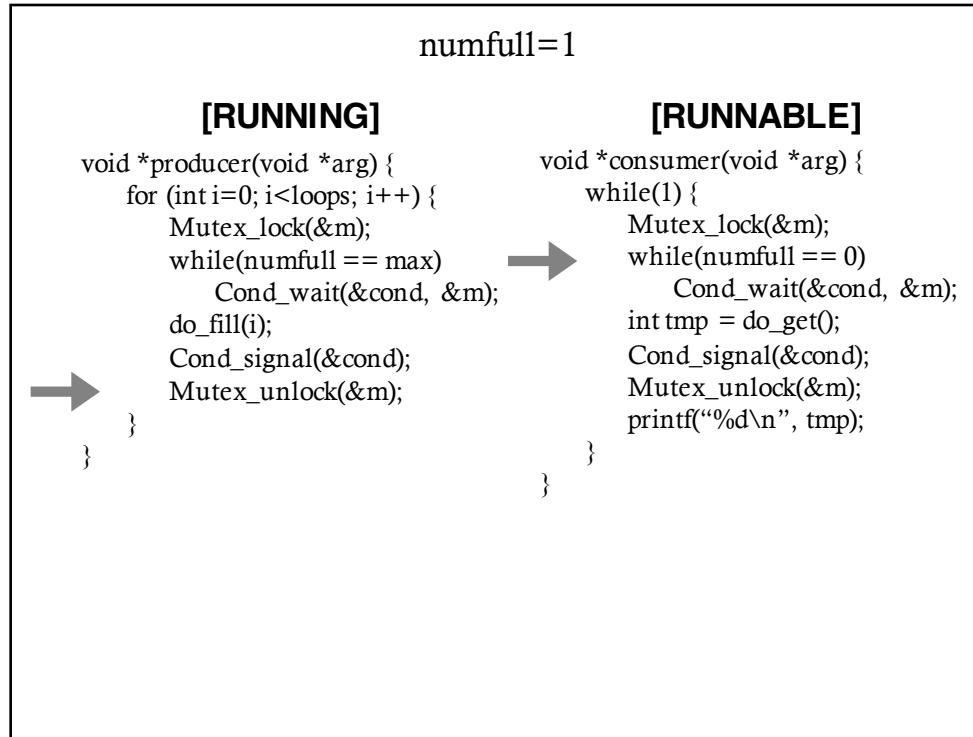


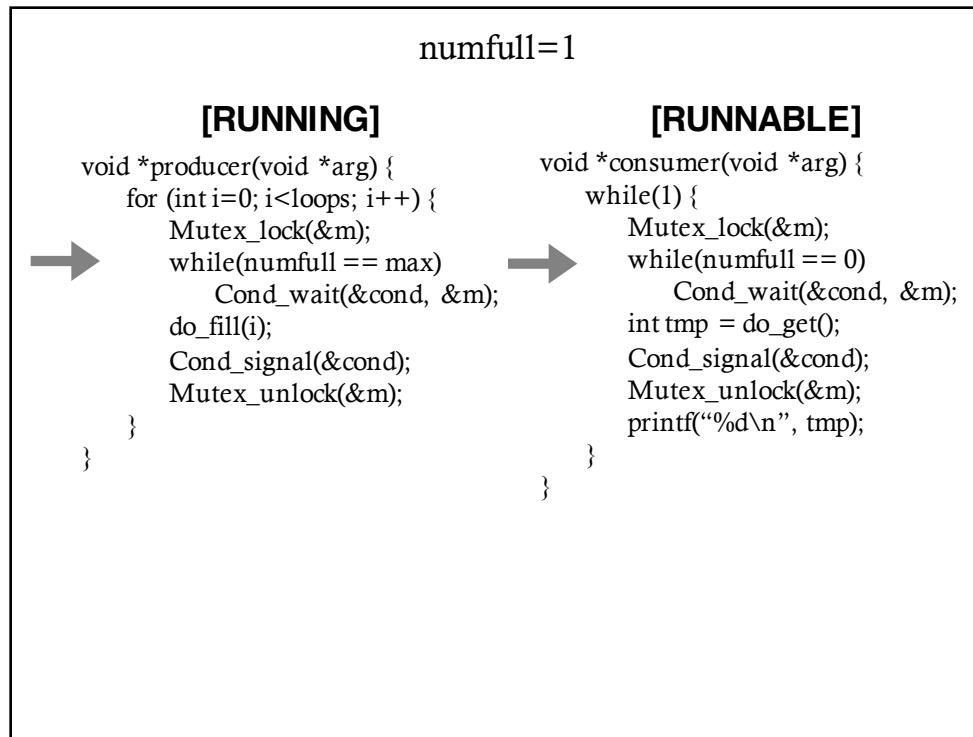
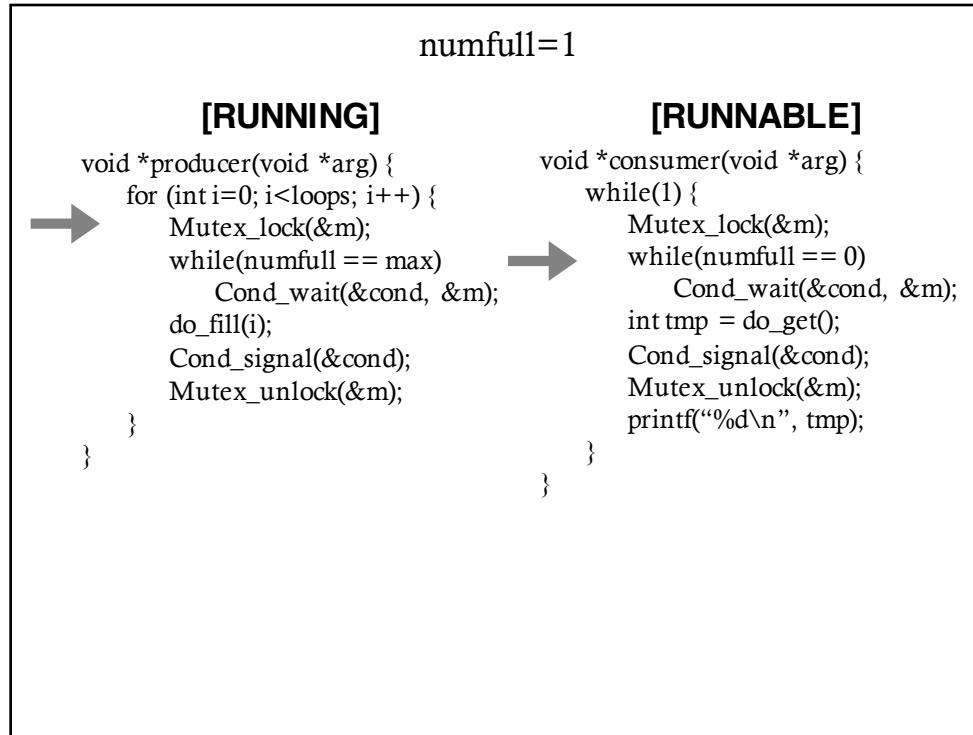


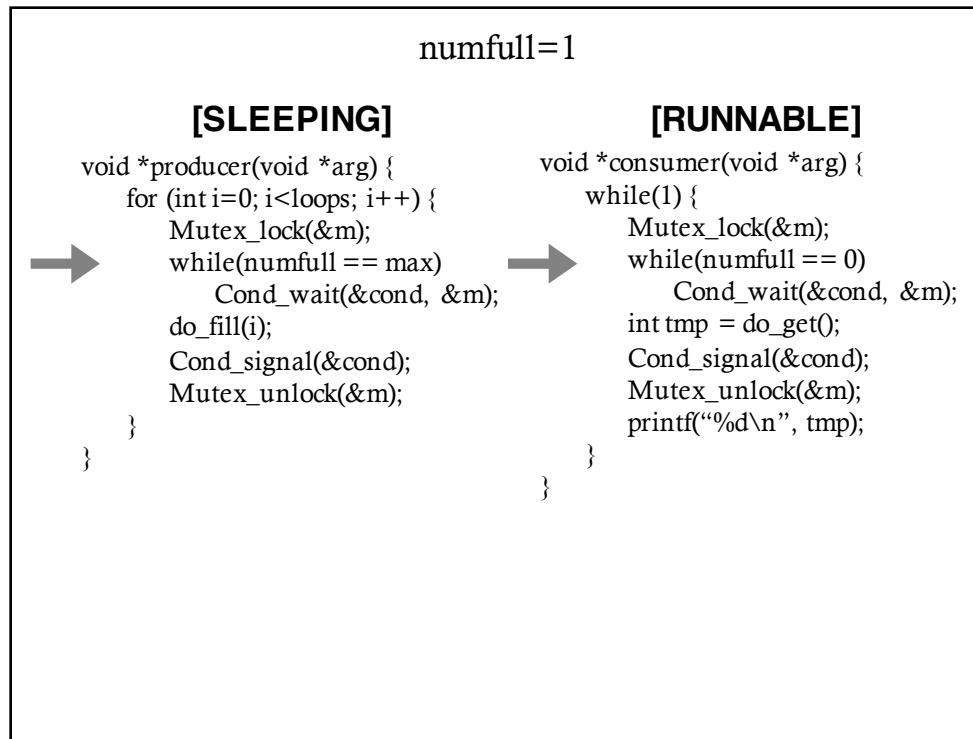
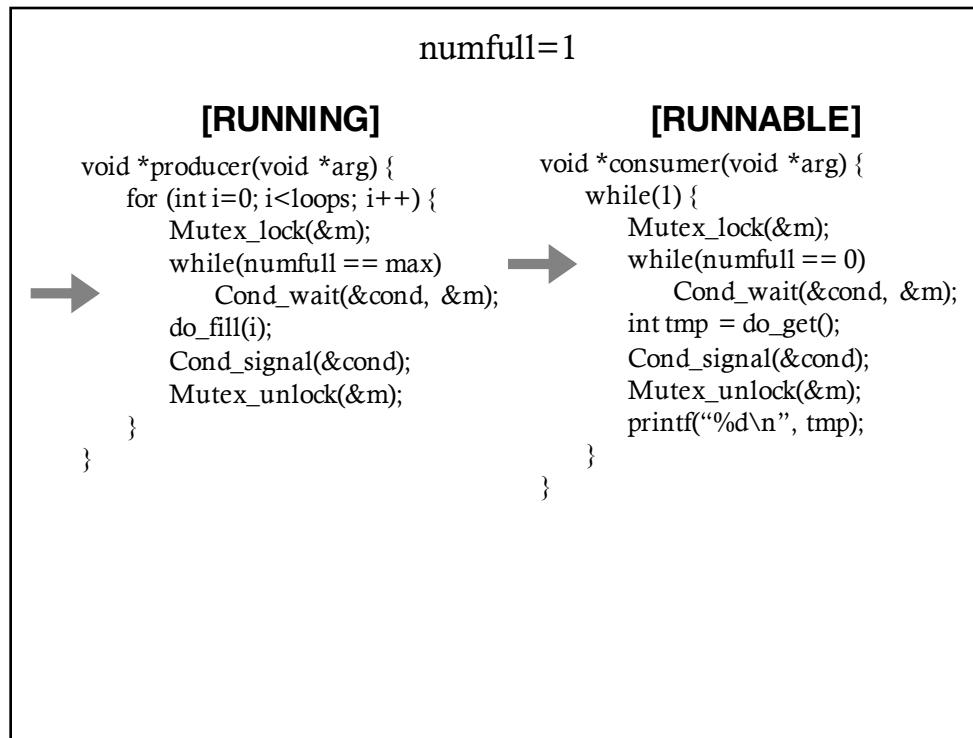


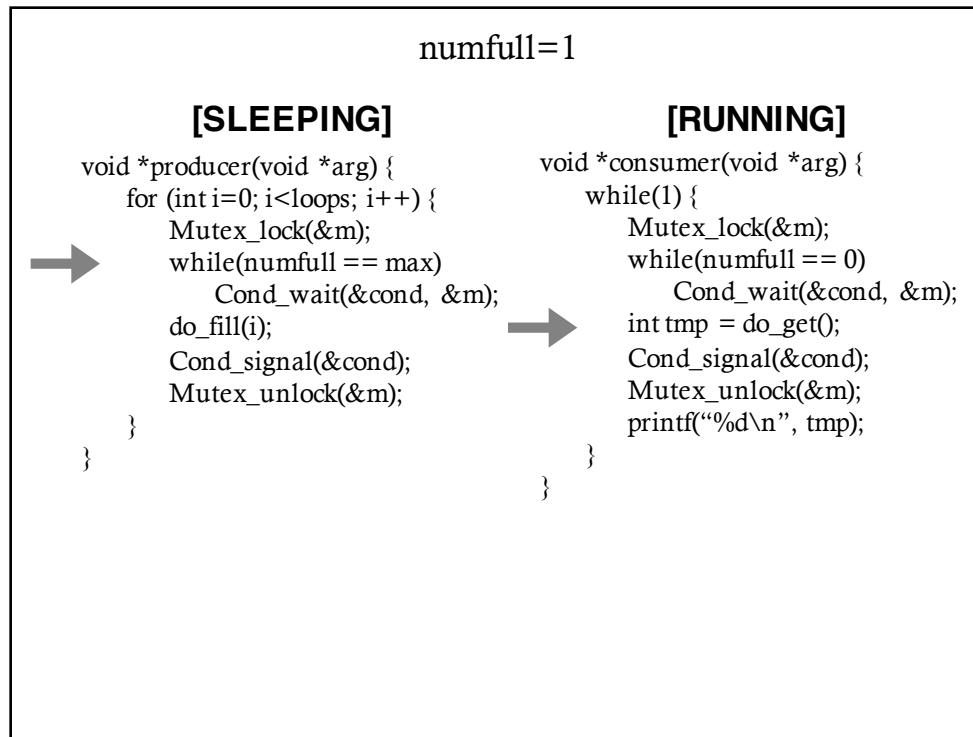
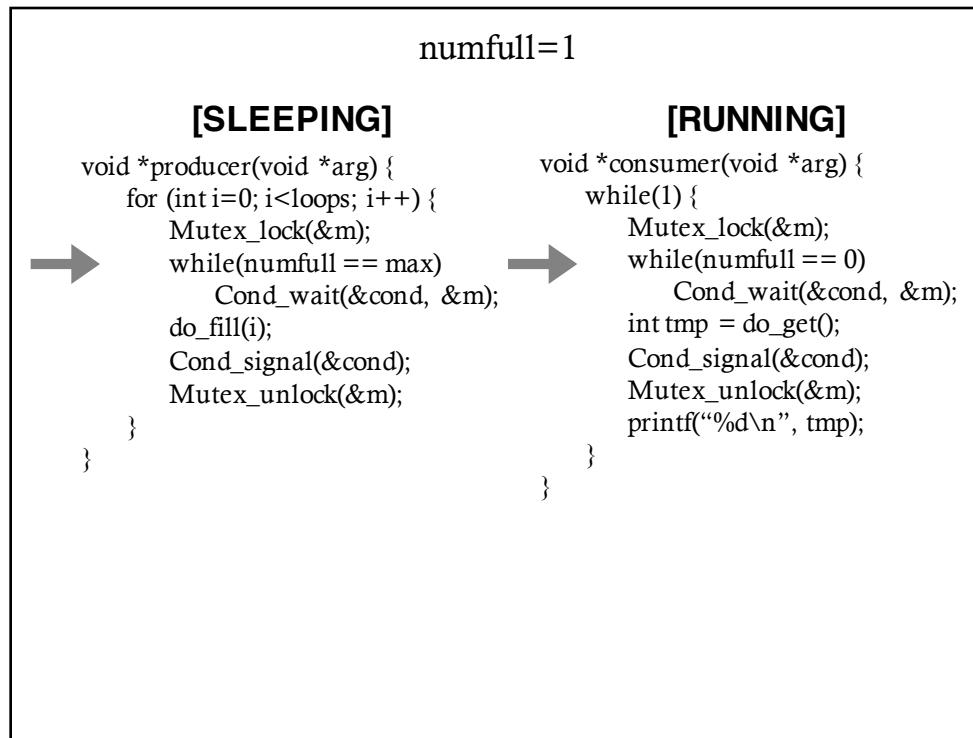


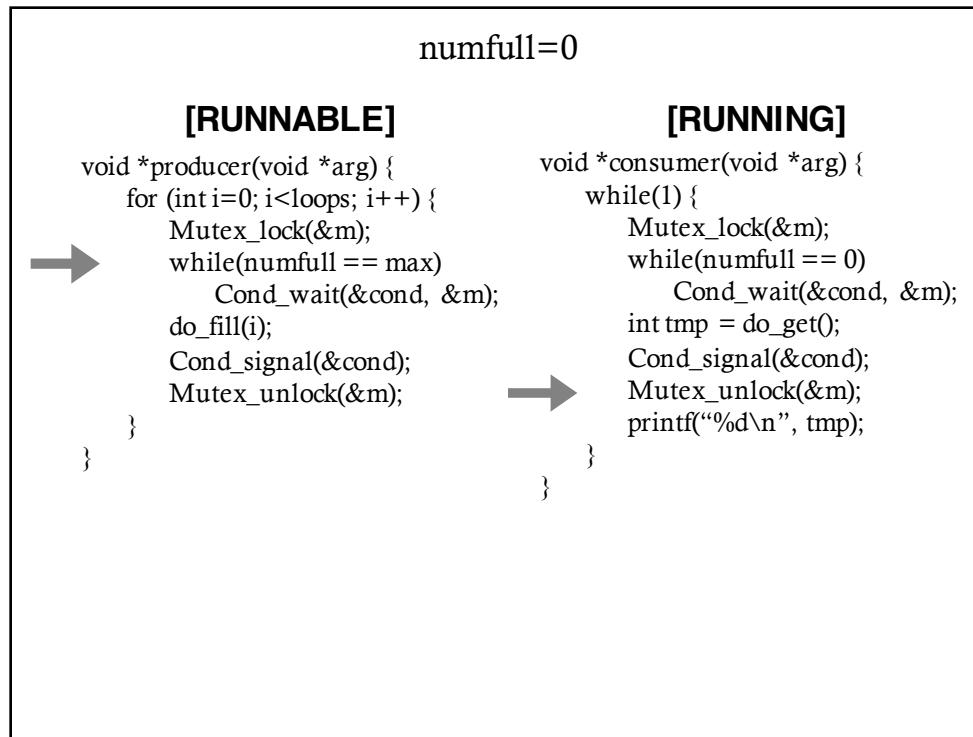
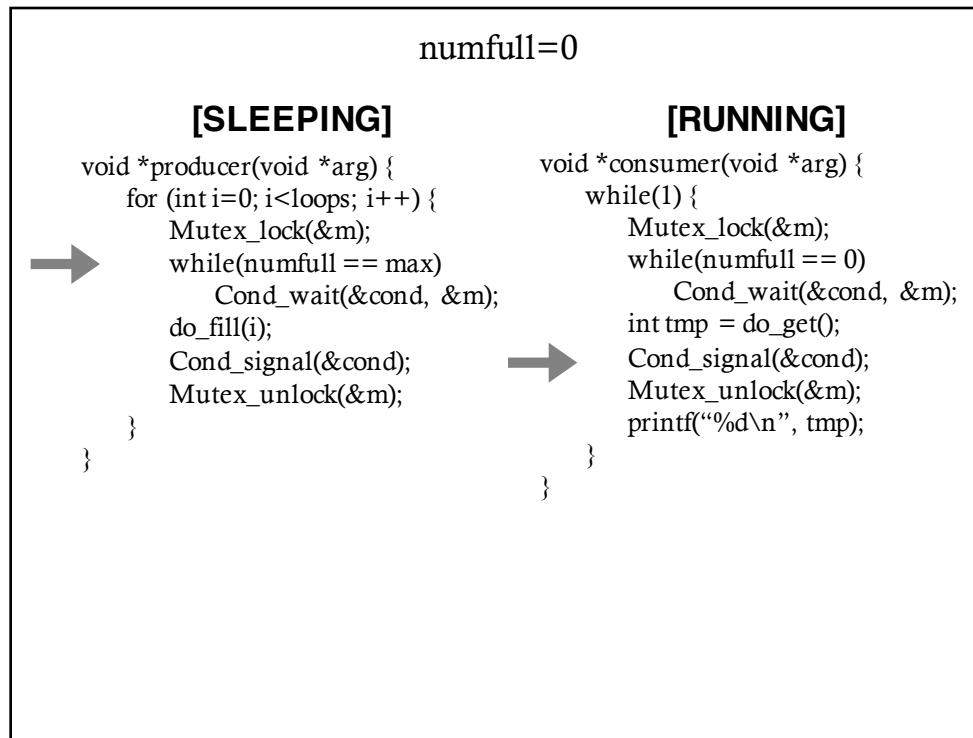


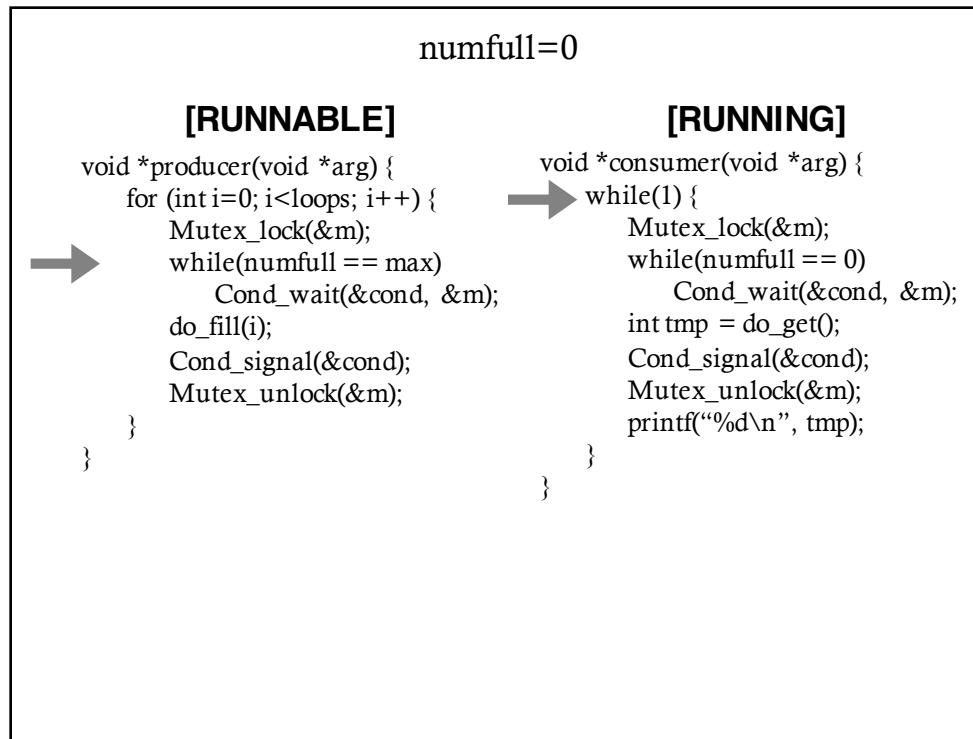
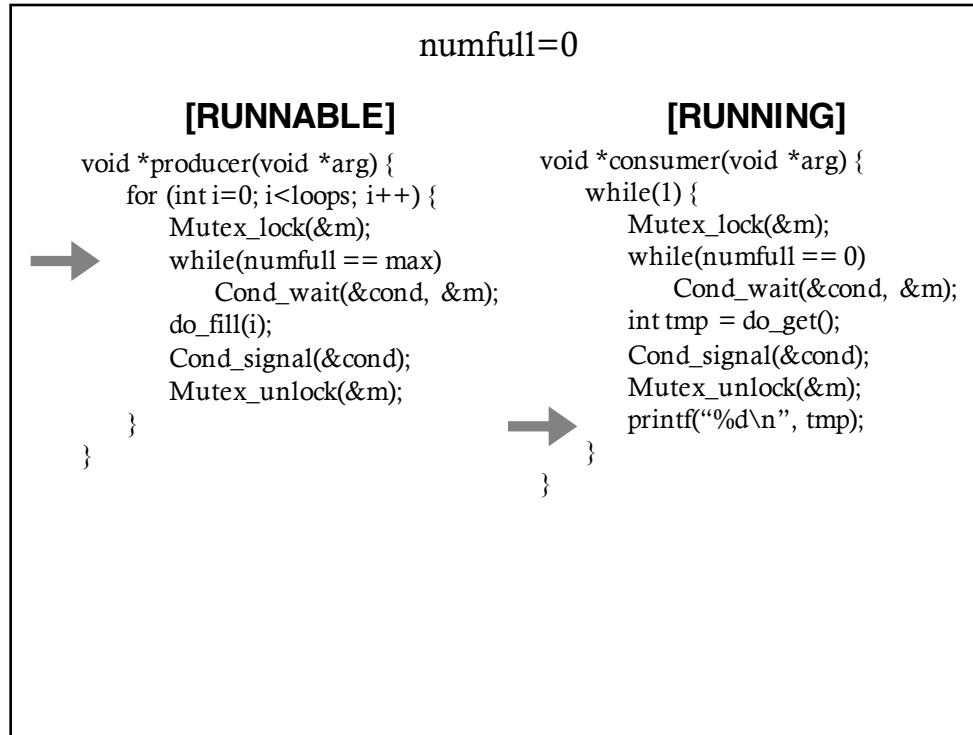


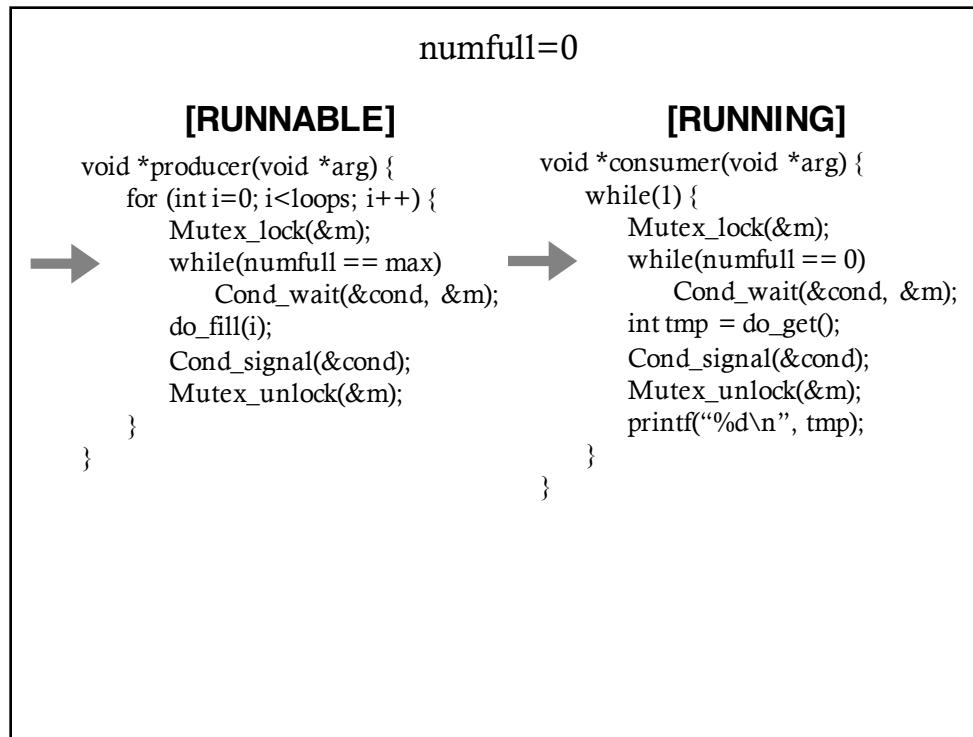
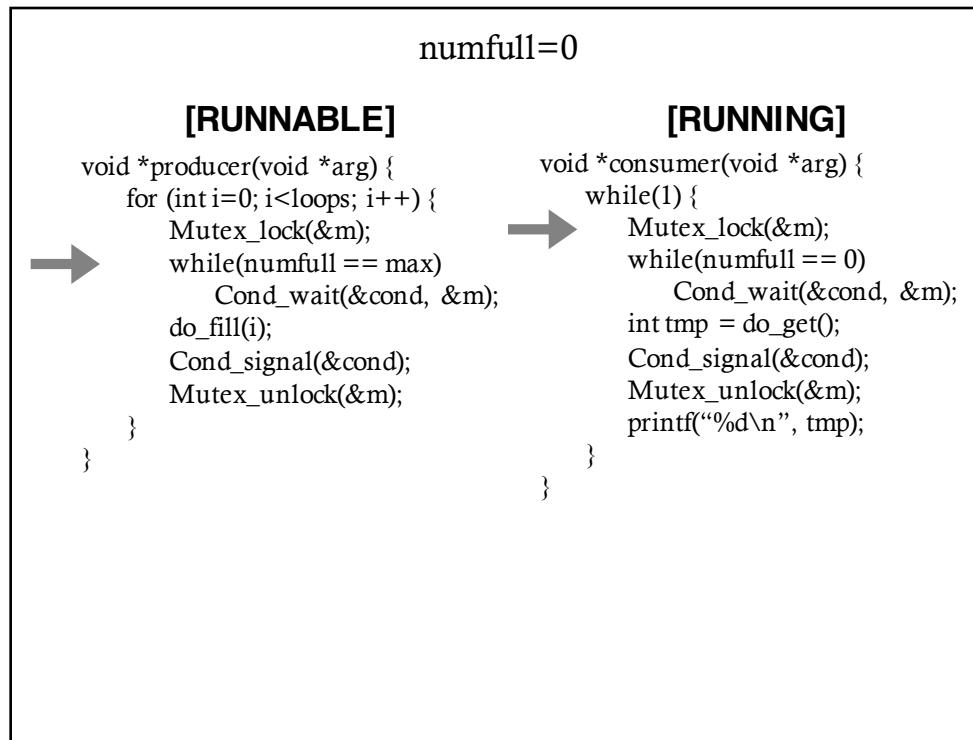


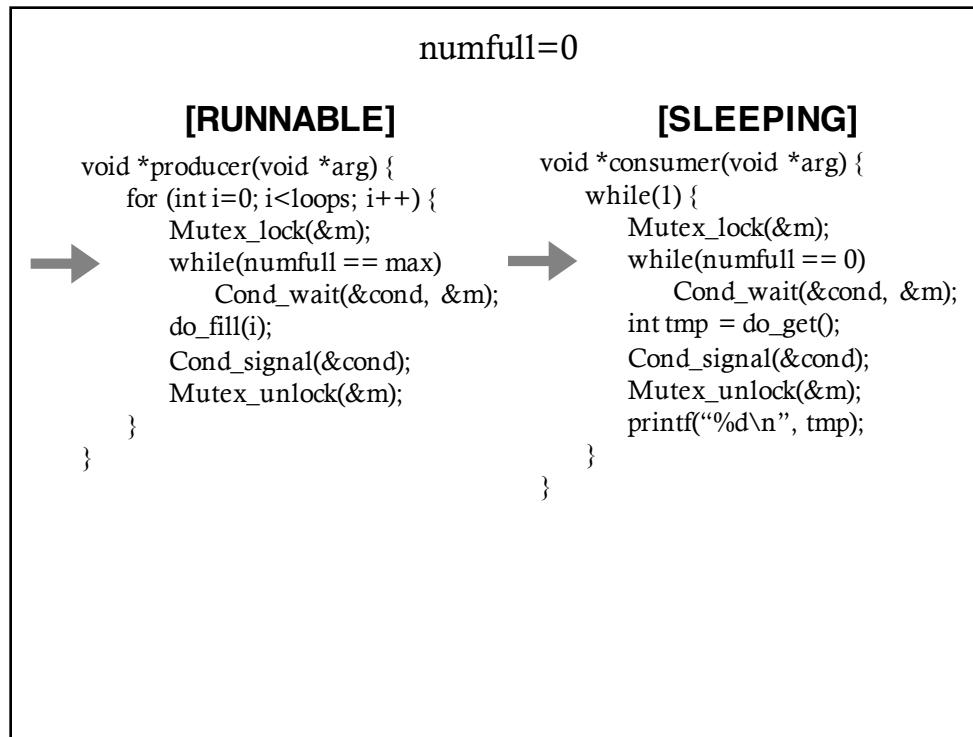
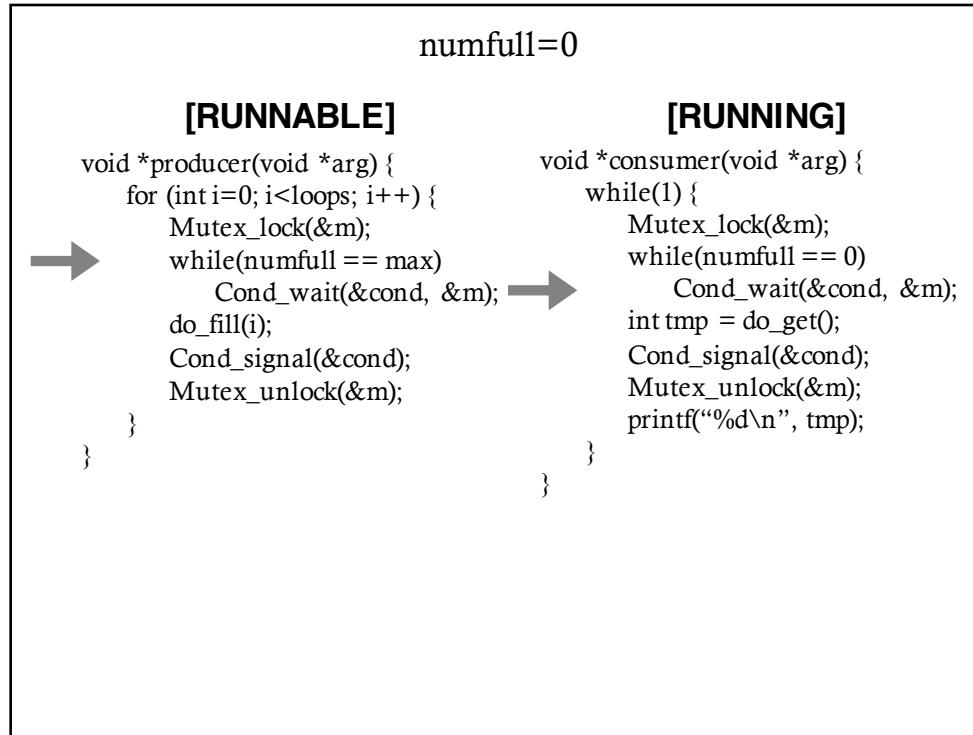


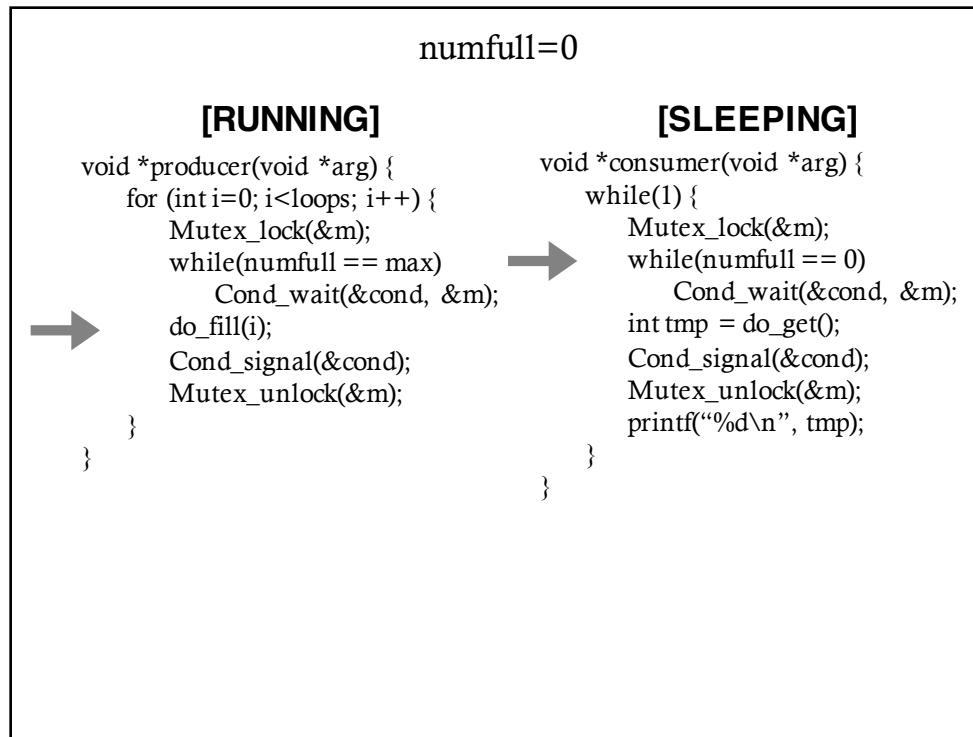
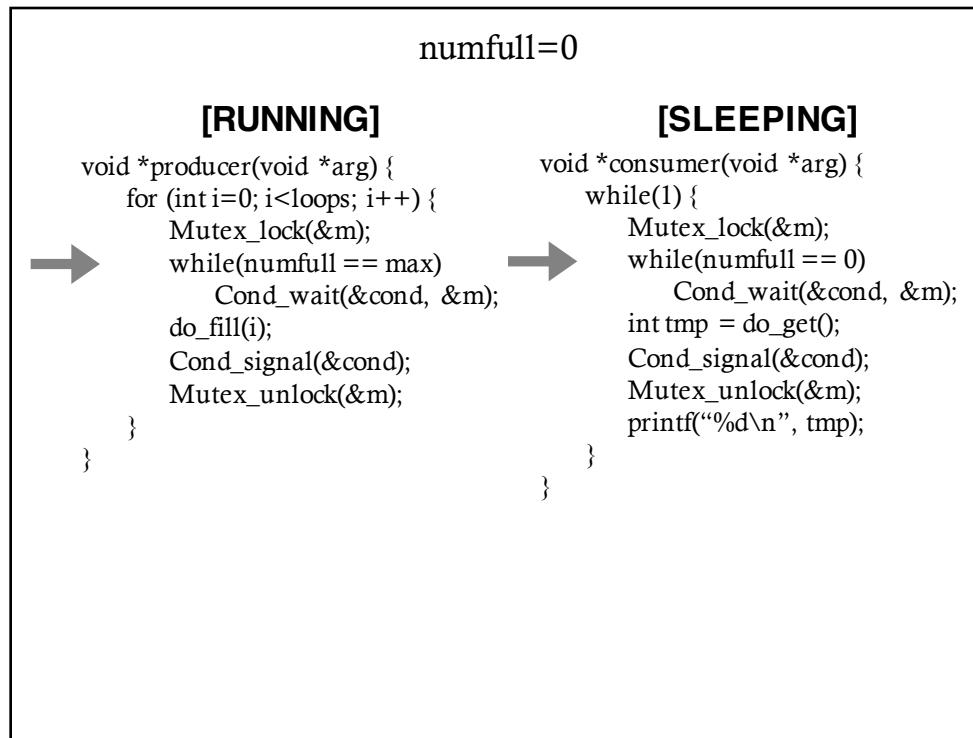


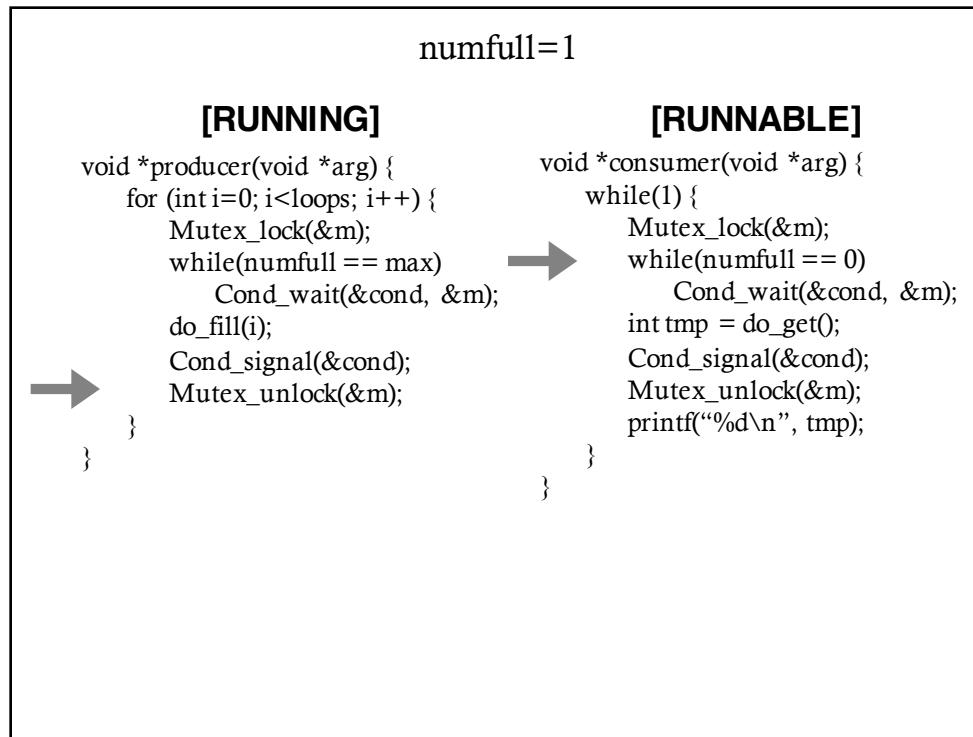
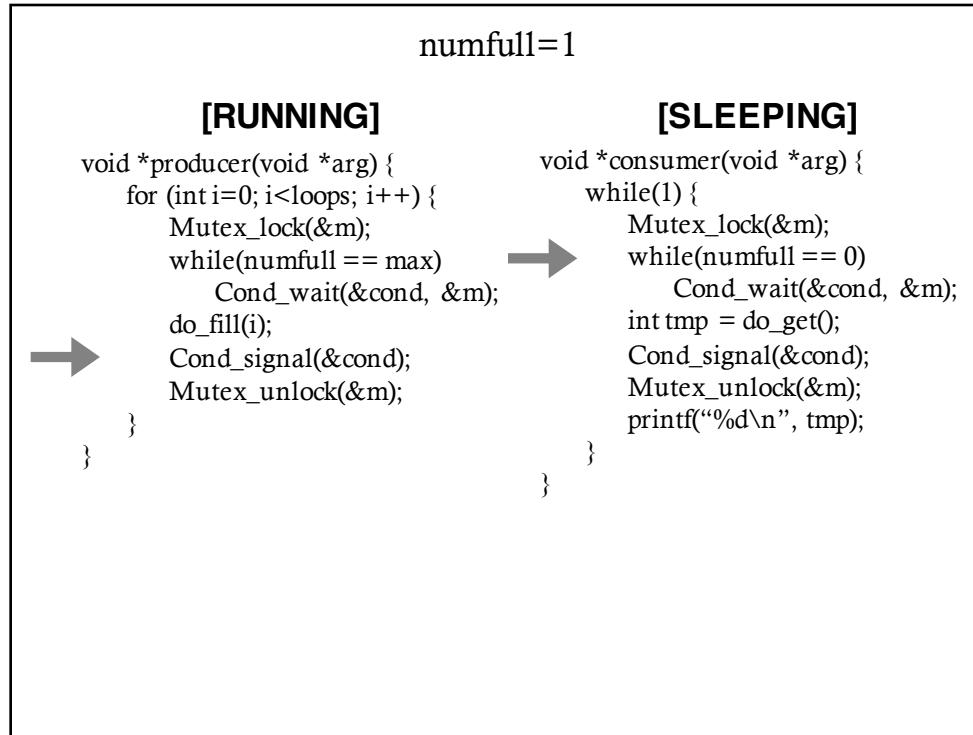












WHAT ABOUT 2 CONSUMERS?

Can you find a problematic timeline with 2 consumers
(still 1 producer)?

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

Producer: wait() wait() signal() wait() signal()
 Consumer1: c1 c2 c3 c1 c2 c3 p1 p2 p4 p5 p6 p1 p2 p3
 Consumer2: c1 c2 c3 c1 c2 c3 c2 c4 c5

does last signal wake producer or consumer2?

HOW TO WAKE THE RIGHT THREAD?

One solution:

wake all the threads!

WAKING ALL WAITING THREADS

- **wait**(cond_t *cv, mutex_t *lock)
 - - assumes the lock is held when wait() is called
 - - puts caller to sleep + releases the lock (atomically)
 - - 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

- **broadcast**(cond_t *cv) any disadvantage?
 - - wake **all** waiting threads (if >= 1 thread is waiting)
 - - if there are no waiting thread, just return, doing nothing

EXAMPLE NEED FOR BROADCAST

```
void *allocate(int size) {           void free(void *ptr, int size) {  
    mutex_lock(&m);                ...  
    while (bytesLeft < size)        cond_broadcast(&c)  
        cond_wait(&c);            ...  
    ...  
}
```

HOW TO WAKE THE RIGHT THREAD?

- One solution:
wake all the threads!
- Better solution (usually): use two condition variables

PRODUCER/CONSUMER: TWO CVS

```

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}

```

Is this correct? Can you find a bad schedule?

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.

GOOD RULE OF THUMB 3

Whenever a lock is acquired, recheck assumptions about state!

Possible for another thread to grab lock in between signal and wakeup from wait

Note that some libraries also have “spurious wakeups” (may wake multiple waiting threads at signal)

PRODUCER/CONSUMER: TWO CVS AND WHILE

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); // p6
    }
}

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);
    }
}
```

Is this correct? Can you find a bad schedule?

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()`

SUMMARY: CV RULES OF THUMB

Keep state in addition to CV's

Always do wait/signal with lock held

Whenever thread wakes from waiting, recheck state