

15-213

“The course that gives CMU its Zip!”

Concurrency II: Synchronization

Nov 14, 2000

Topics

- Progress graphs
- Semaphores
- Mutex and condition variables
- Barrier synchronization
- Timeout waiting

A version of badcnt.c with a simple counter loop

```
int ctr = 0; /* shared */

/* main routine creates*/
/* two count threads */

/* count thread */
void *count(void *arg) {
    int i;

    for (i=0; i<NITERS; i++)
        ctr++;
    return NULL;
}
```

note: counter should be equal to 200,000,000

```
linux> badcnt
BOOM! ctr=198841183

linux> badcnt
BOOM! ctr=198261801

linux> badcnt
BOOM! ctr=198269672
```

What went wrong?

Assembly code for counter loop

C code for counter loop

```
for (i=0; i<NITERS; i++)  
    ctr++;
```

Corresponding asm code (gcc -O0 -fforce-mem)

Head (H_i)	{	.L9:	<code>movl -4(%ebp),%eax</code>
		<code>cmpl \$99999999,%eax</code>	
		<code>jle .L12</code>	
		<code>jmp .L10</code>	
Load ctr (L_i) Update ctr (U_i) Store ctr (S_i)	{	.L12:	<code>movl ctr,%eax # Load</code>
		<code>leal 1(%eax),%edx # Update</code>	
		<code>movl %edx,ctr # Store</code>	
Tail (T_i)	{	.L11:	<code>movl -4(%ebp),%eax</code>
		<code>leal 1(%eax),%edx</code>	
		<code>movl %edx,-4(%ebp)</code>	
		<code>jmp .L9</code>	
		.L10:	

Concurrent execution

Key thread idea: In general, any sequentially consistent interleaving is possible, but some are incorrect!

- I_i denotes that thread i executes instruction I
- $\%eax_i$ is the contents of $\%eax$ in thread i 's context

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	ctr
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

Concurrent execution (cont)

Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2.

i (thread)	instr _i	%eax ₁	%eax ₂	ctr
1	H ₁	-	-	0
1	L ₁	0	-	0
1	U ₁	1	-	0
2	H ₂	-	-	0
2	L ₂	-	0	0
1	S ₁	1	-	1
1	T ₁	1	-	1
2	U ₂	-	1	1
2	S ₂	-	1	1
2	T ₂	-	1	1

Oops!

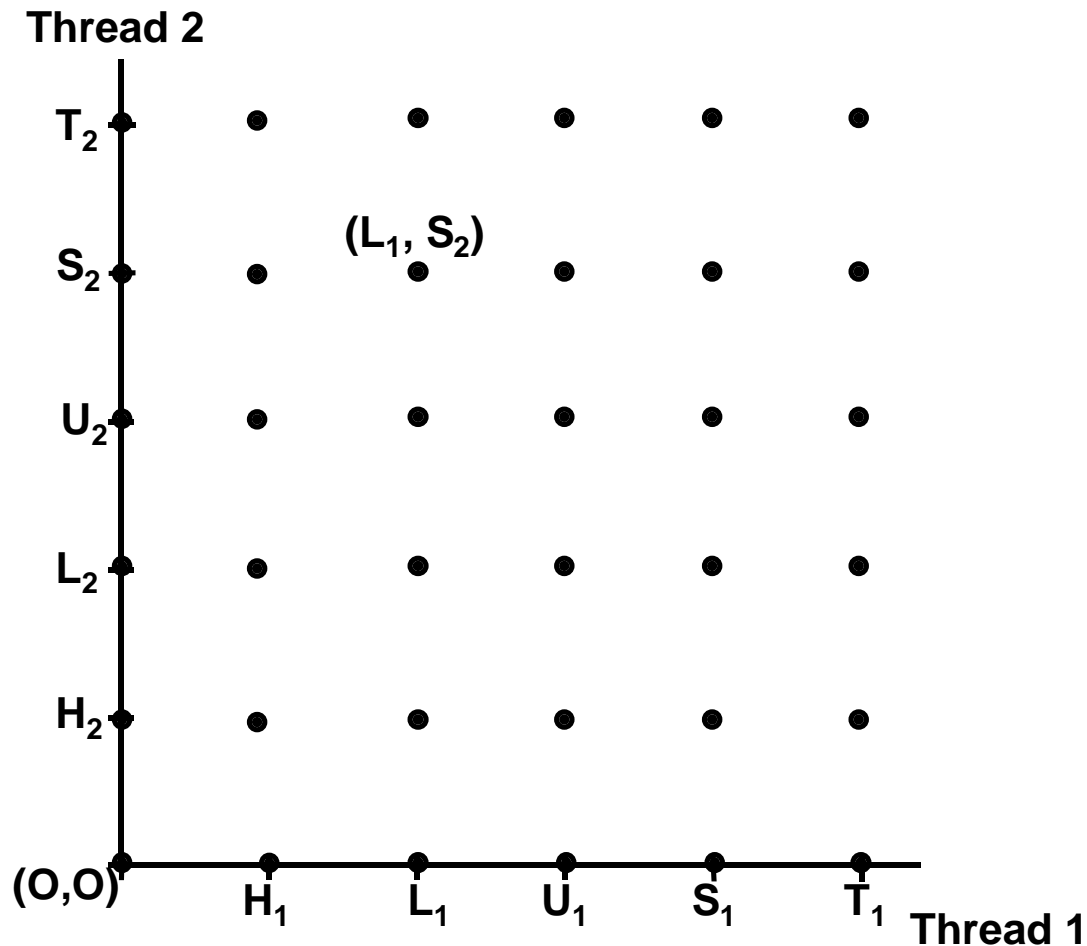
Concurrent execution (cont)

How about this ordering?

i (thread)	instr _i	%eax ₁	%eax ₂	ctr
1	H ₁			
1	L ₁			
2	H ₂			
2	L ₂			
2	U ₂			
2	S ₂			
1	U ₁			
1	S ₁			
1	T ₁			
2	T ₂			

We can clarify our understanding of concurrent execution with the help of the *progress graph*

Progress graphs



A *progress graph* depicts the discrete *execution state space* of concurrent threads.

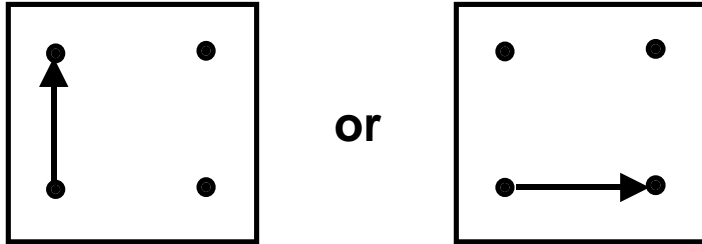
Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible *execution state* $(Inst_1, Inst_2)$.

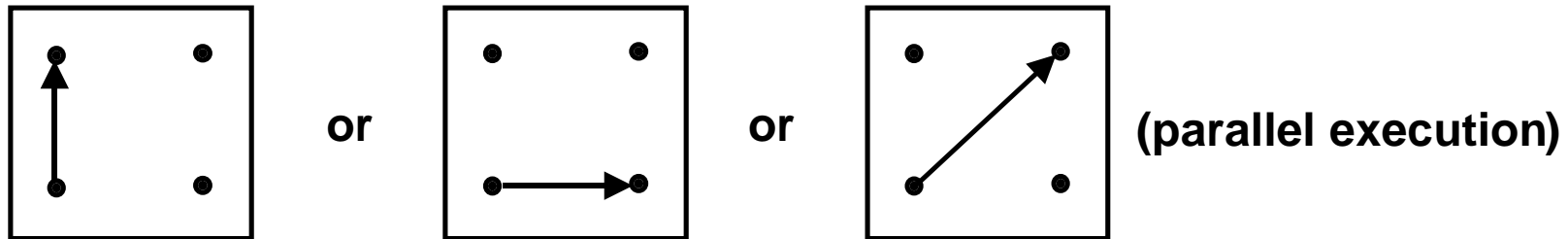
E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2 .

Legal state transitions

Interleaved concurrent execution (one processor):

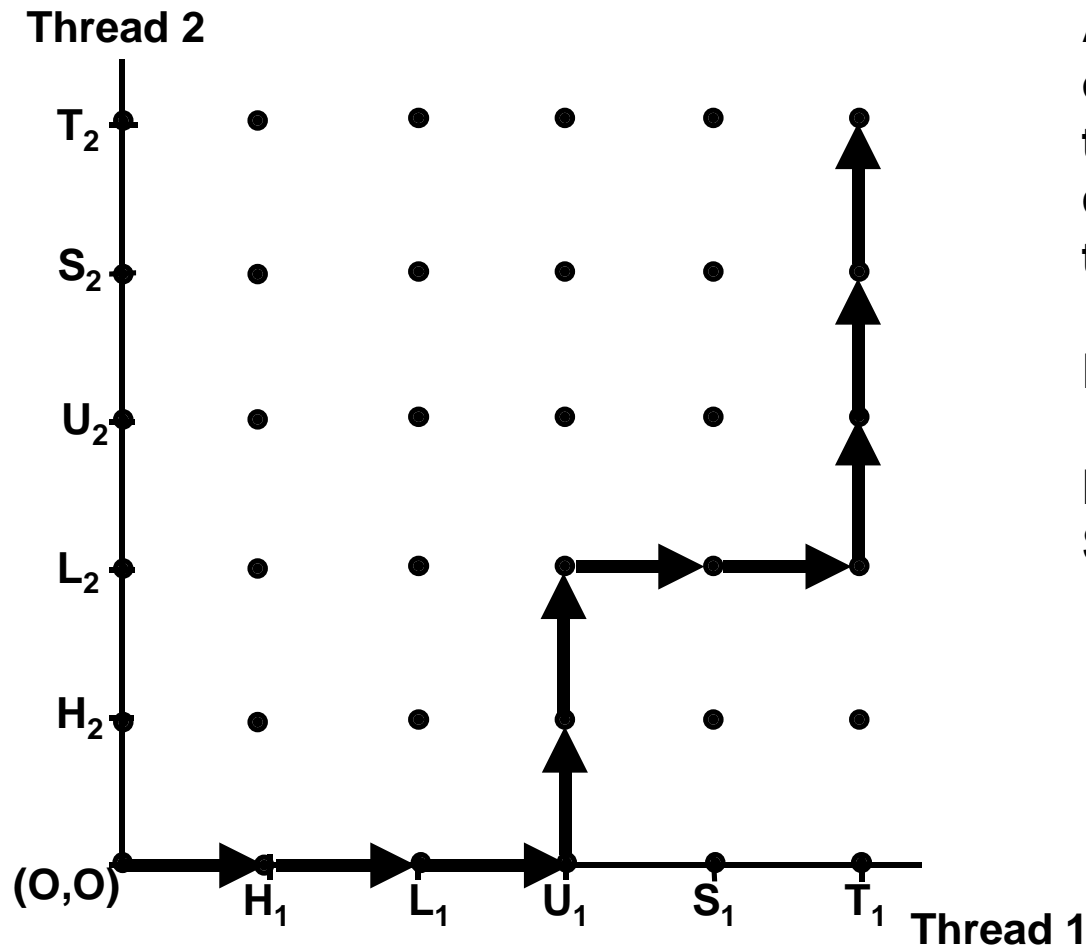


Parallel concurrent execution (multiple processors)



Key point: Always reason about concurrent threads as if each thread had its own CPU.

Trajectories

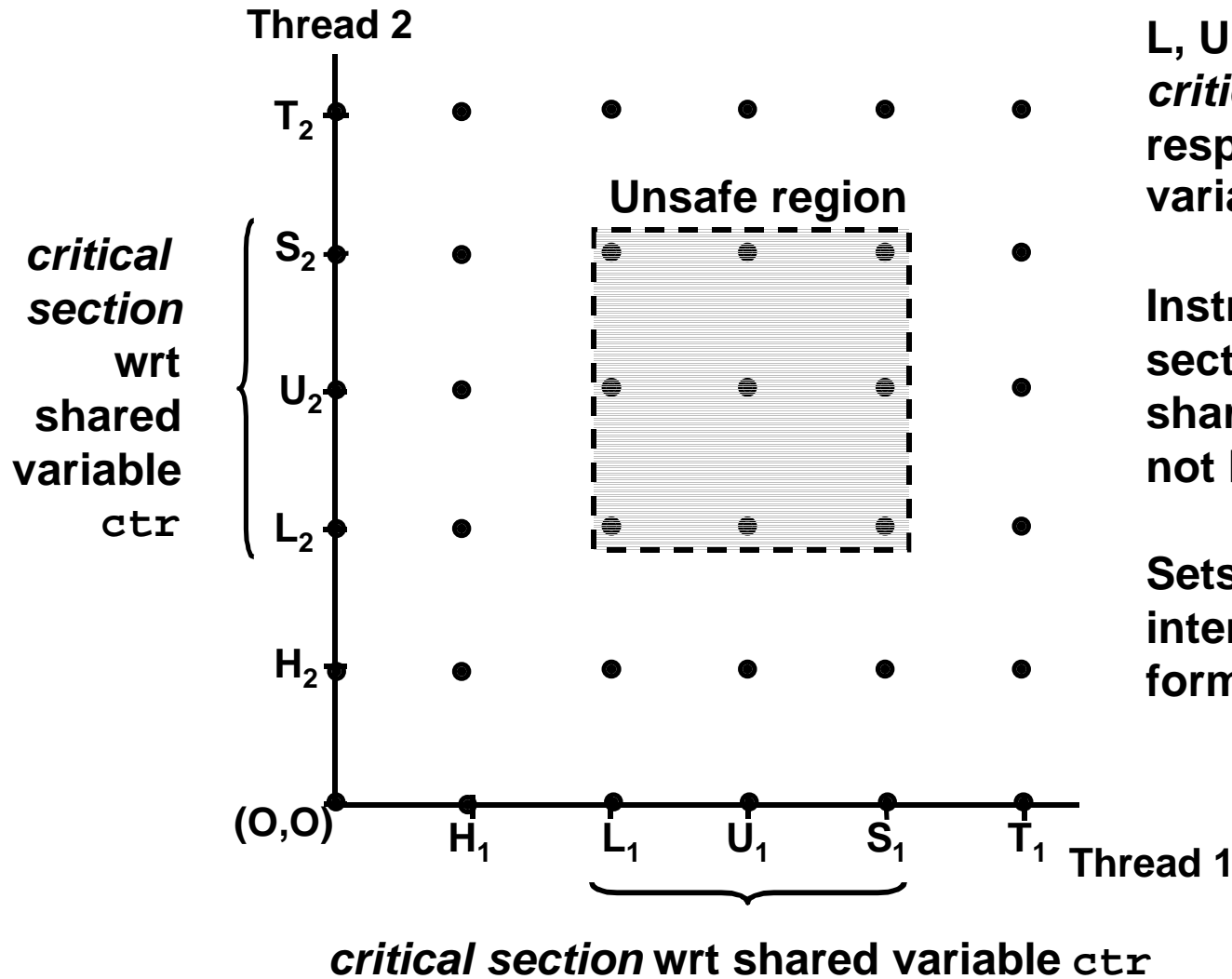


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L2, U1, H2, L2,
S1, T1, U2, S2, T2

Critical sections and unsafe regions

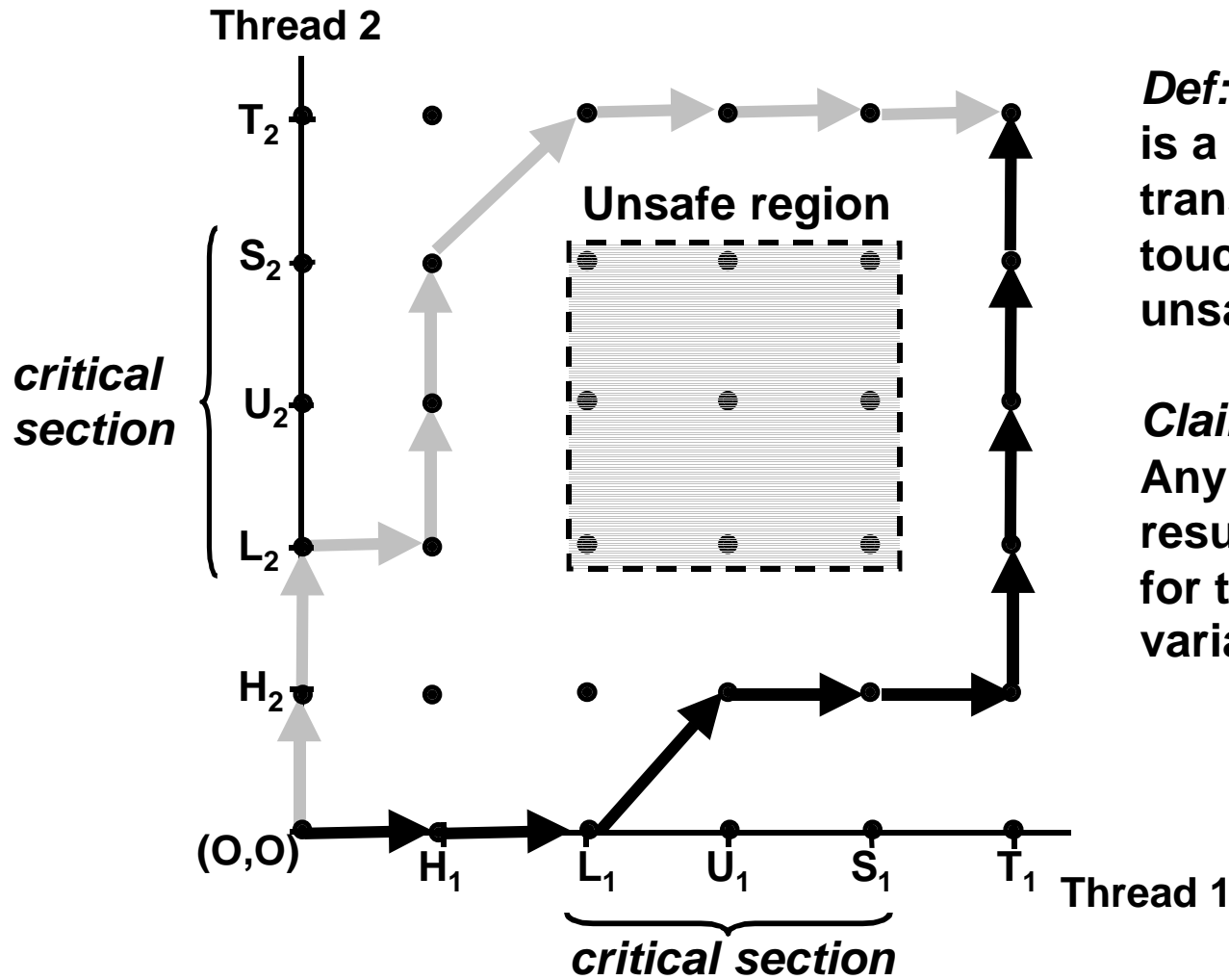


L, U, and S form a *critical section* with respect to the shared variable `ctr`.

Instructions in critical sections (wrt to some shared variable) should not be interleaved.

Sets of states where such interleaving occurs form *unsafe regions*.

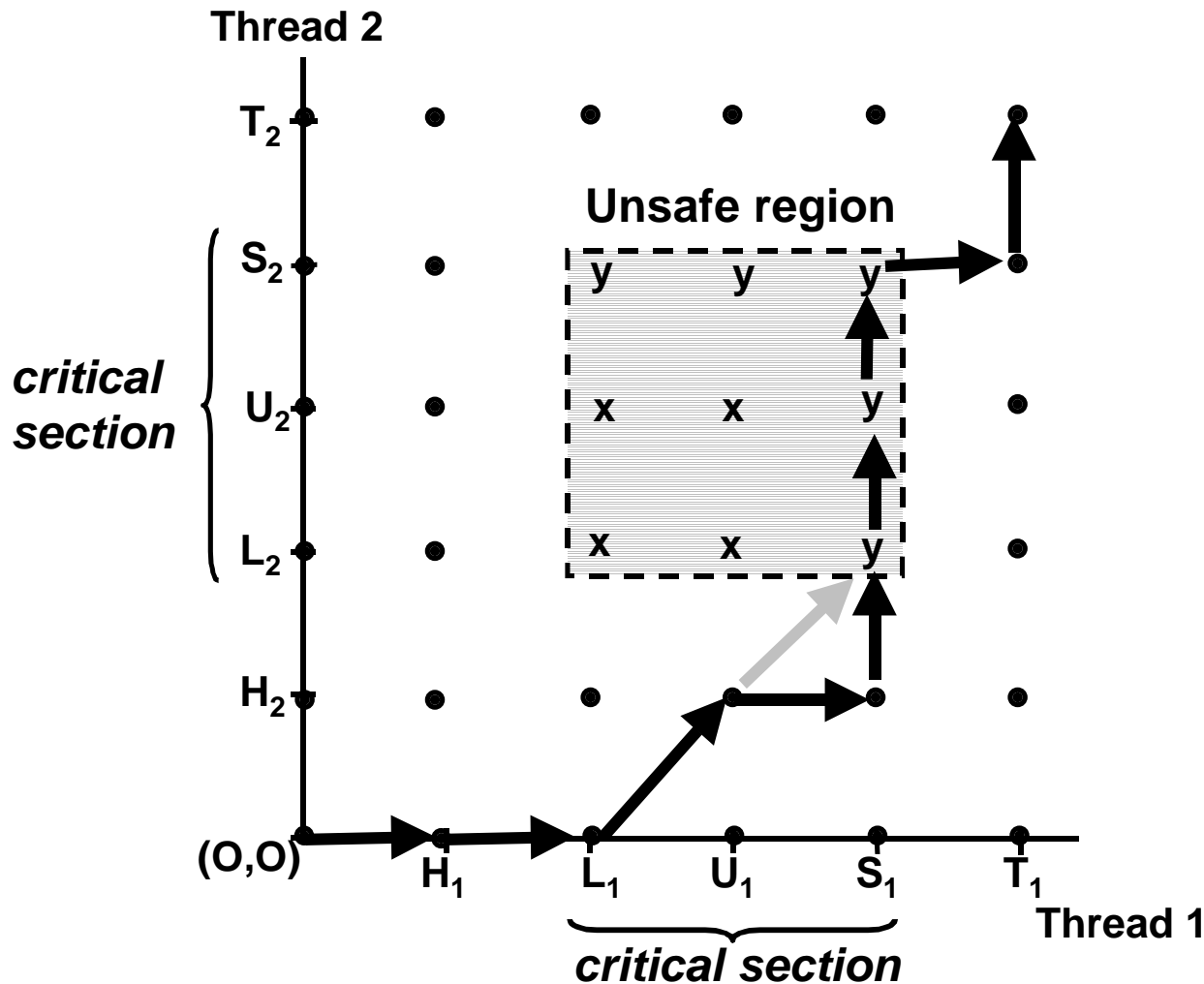
Safe trajectories



Def: A *safe trajectory* is a sequence of legal transitions that does not touch any states in an unsafe region.

Claim: Any safe trajectory results in a correct value for the shared variable `ctr`.

Unsafe trajectories



Touching a state of type x is always incorrect.

Touching a state of type y may or may not be OK:

→ correct because store completes before load.

→ incorrect because order of load and store are indeterminate.

Moral: be conservative and disallow all unsafe trajectories.

Semaphore operations

Question: How can we guarantee a safe trajectory?

- We must *synchronize* the threads so that they never enter an unsafe state.

Classic solution: Dijkstra's P and V operations on *semaphores*.

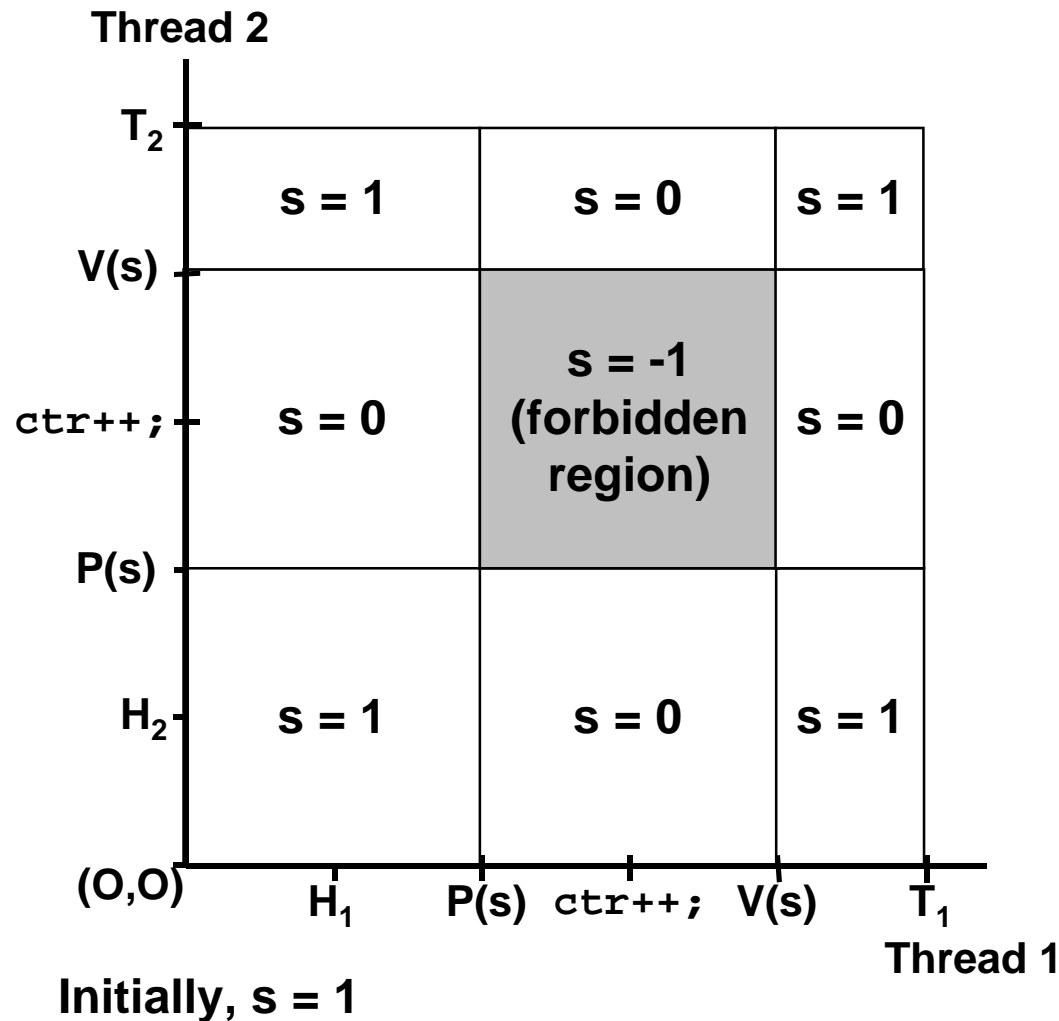
- **semaphore:** non-negative integer synchronization variable.
- **P(s):** [while (s == 0) wait(); s--;]
 - Dutch for "Proberen" (test)
- **V(s):** [s++;]
 - Dutch for "Verhogen" (increment)
- **OS guarantees that operations between brackets [] are executed indivisibly.**
 - Only one P or V operation at a time can modify s.
 - When while loop in P terminates, only that P can decrement s.
- **Semaphore invariant:** (s >= 0)

Sharing with semaphores

Provide mutually exclusive access to shared variable by surrounding critical section with P and V operations on semaphore s (initially set to 1).

Semaphore invariant creates a *forbidden region* that encloses unsafe region and is never touched by any trajectory.

Semaphore used in this way is often called a *mutex* (mutual exclusion).



Posix semaphores

```
/* initialize semaphore sem to value */
/* pshared=0 if thread, pshared=1 if process */
void Sem_init(sem_t *sem, int pshared, unsigned int value) {
    if (sem_init(sem, pshared, value) < 0)
        unix_error("Sem_init");
}

/* P operation on semaphore sem */
void P(sem_t *sem) {
    if (sem_wait(sem))
        unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
    if (sem_post(sem))
        unix_error("V");
}
```

Sharing with Posix semaphores

```
/* goodcnt.c - properly synch'd */
/* version of badcnt.c */
#include <ics.h>
#define NITERS 10000000

void *count(void *arg);

struct {
    int ctr;      /* shared ctr */
    sem_t mutex; /* semaphore */
} shared;

int main() {
    pthread_t tid1, tid2;

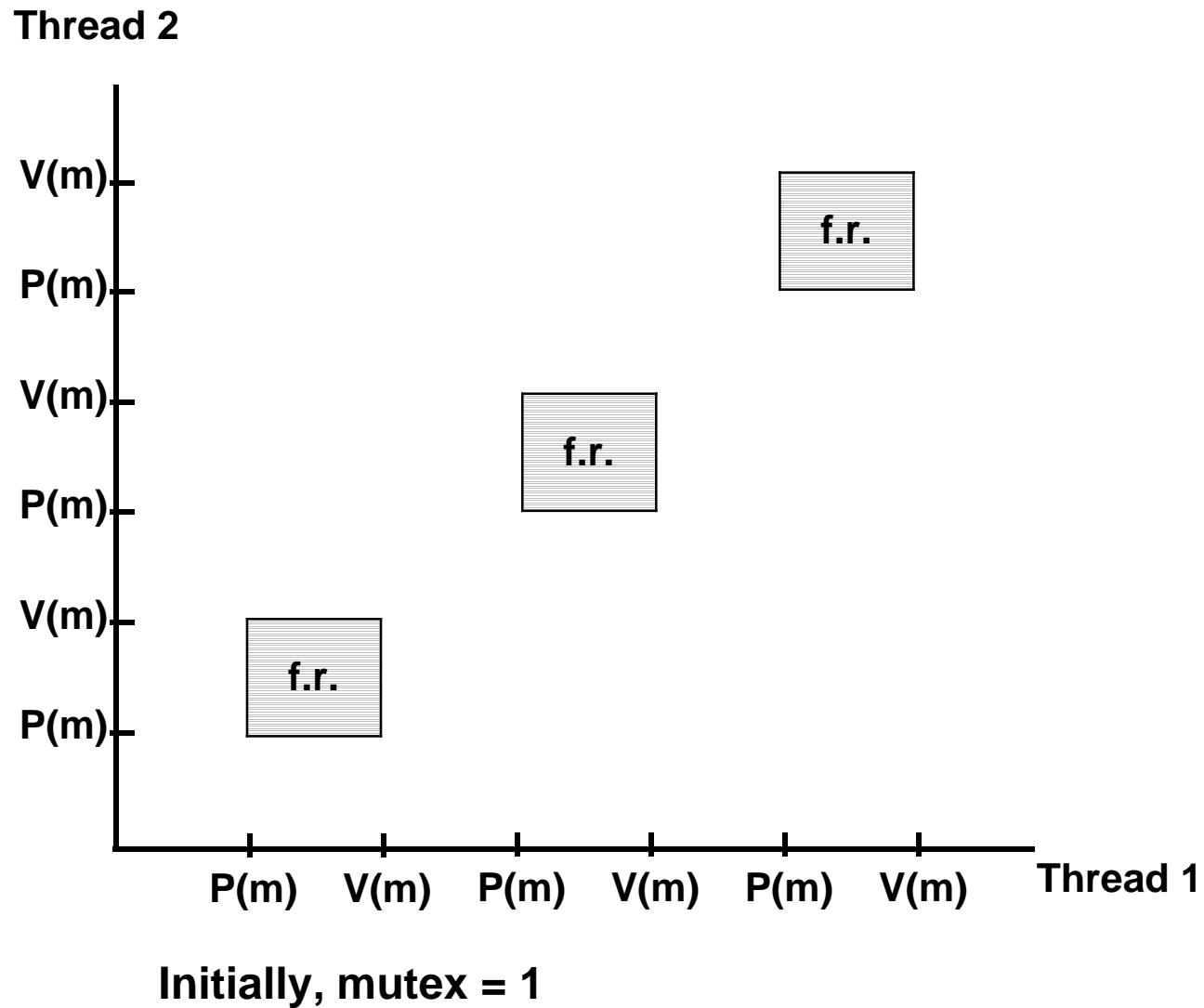
    /* init mutex semaphore to 1 */
    Sem_init(&shared.mutex, 0, 1);

    /* create 2 ctr threads and wait */
    ...
}
```

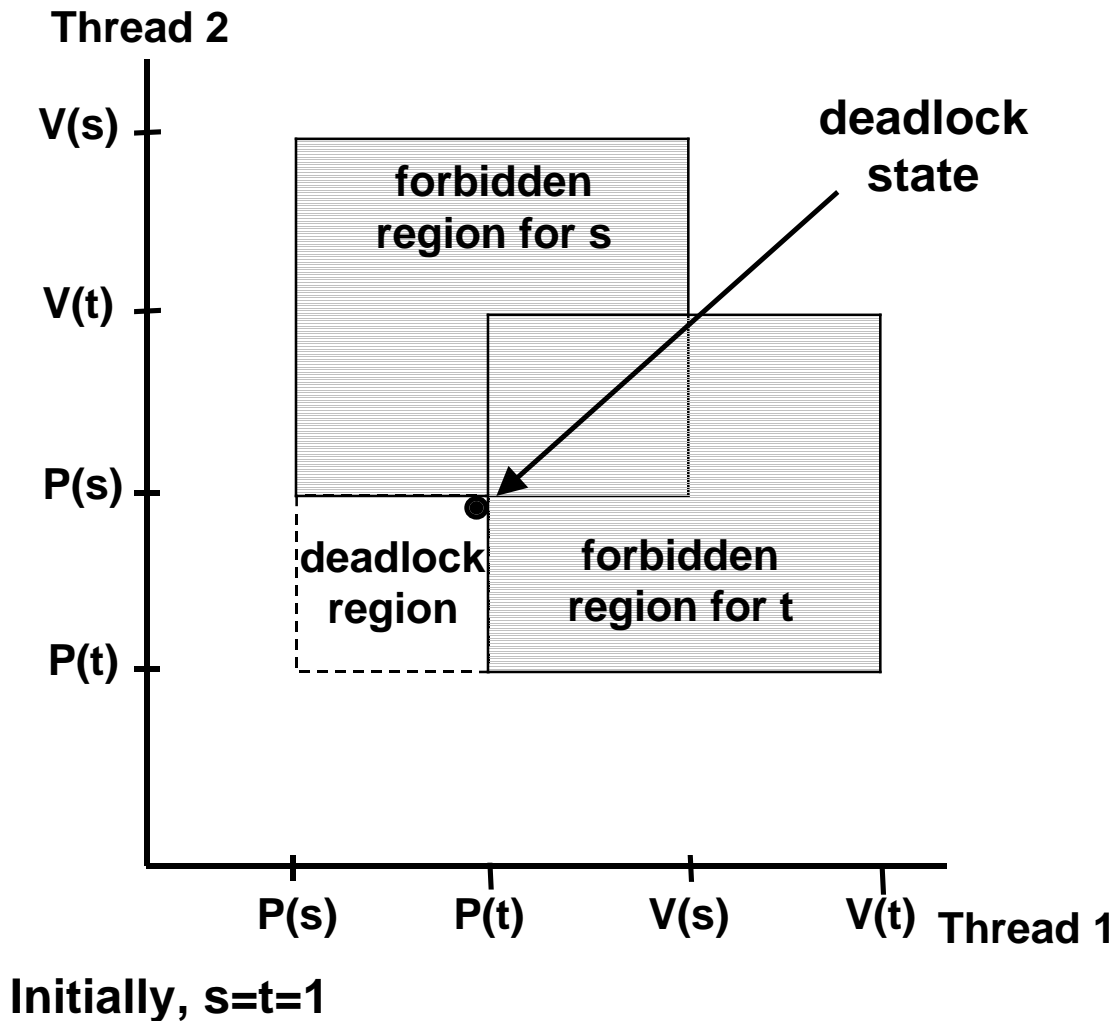
```
/* counter thread */
void *count(void *arg) {
    int i;

    for (i=0; i<NITERS; i++) {
        P(&shared.mutex);
        shared.ctr++;
        V(&shared.mutex);
    }
    return NULL;
}
```


Progress graph for goodcnt.c



Deadlock



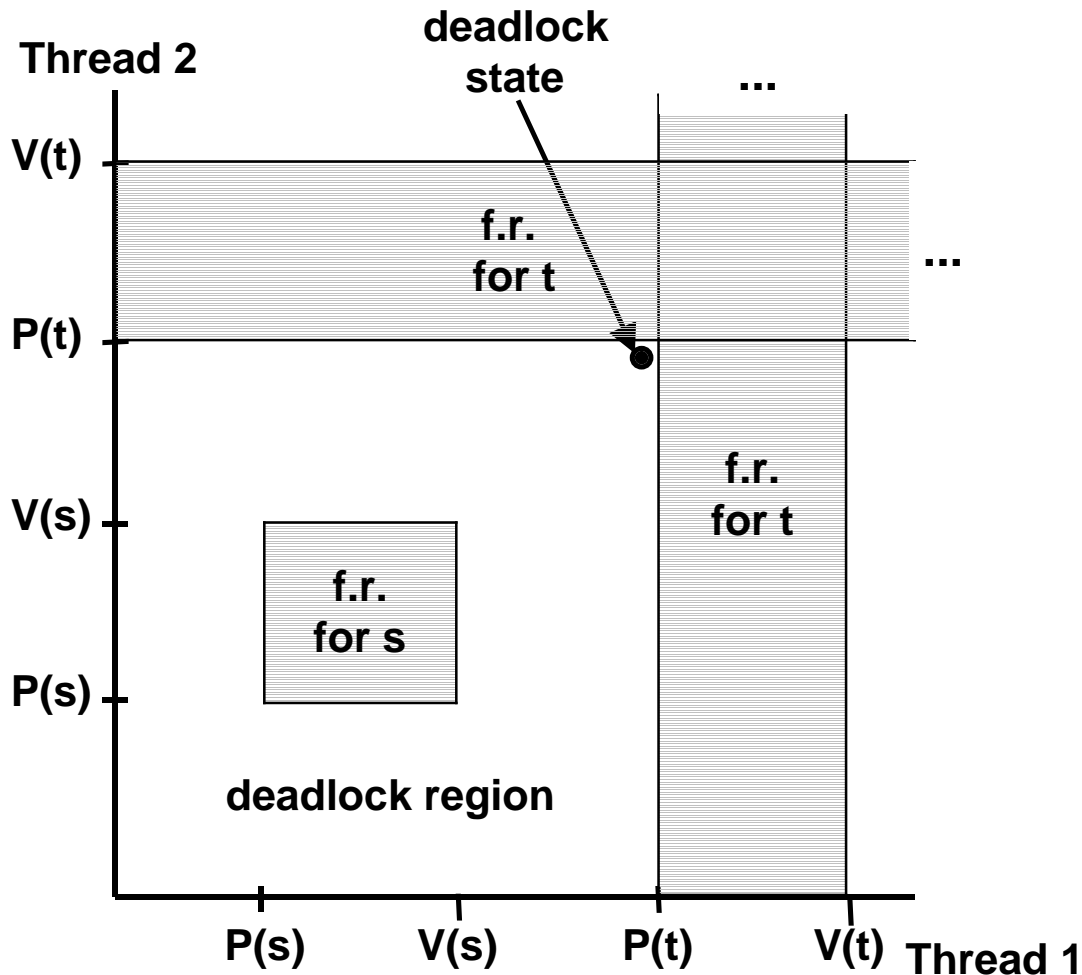
Semaphores introduce the potential for *deadlock*: waiting for a condition that will never be true.

Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state*, waiting for either s or t to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunate fact: deadlock is often non-deterministic.

A deterministic deadlock

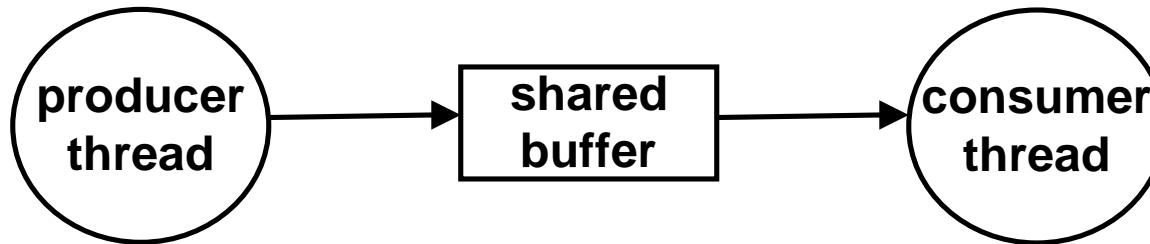


Sometimes though, we get "lucky" and the deadlock is deterministic.

Here is an example of a deterministic deadlock caused by improperly initializing semaphore t .

Problem: correct this program and draw the resulting forbidden regions.

Signaling with semaphores



Common synchronization pattern:

- **Producer** waits for slot, inserts item in buffer, and signals consumer.
- **Consumer** waits for item, removes it from buffer, and signals producer.

Examples

- **Multimedia processing:**
 - producer creates MPEG video frames, consumer renders the frames
- **Graphical user interfaces**
 - producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer.
 - consumer retrieves events from buffer and paints the display.

Producer-consumer (1-buffer)

```
/* buf1.c - producer-consumer
on 1-element buffer */
#include <ics.h>

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
    int buf; /* shared var */
    sem_t full; /* sems */
    sem_t empty;
} shared;
```

```
int main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;

    /* initialize the semaphores */
    Sem_init(&shared.empty, 0, 1);
    Sem_init(&shared.full, 0, 0);

    /* create threads and wait */
    Pthread_create(&tid_producer, NULL,
                  producer, NULL);
    Pthread_create(&tid_consumer, NULL,
                  consumer, NULL);
    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);

    exit(0);
}
```

Producer-consumer (cont)

Initially: empty = 1, full = 0.

```
/* producer thread */
void *producer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* produce item */
        item = i;
        printf("produced %d\n",
            item);

        /* write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}
```

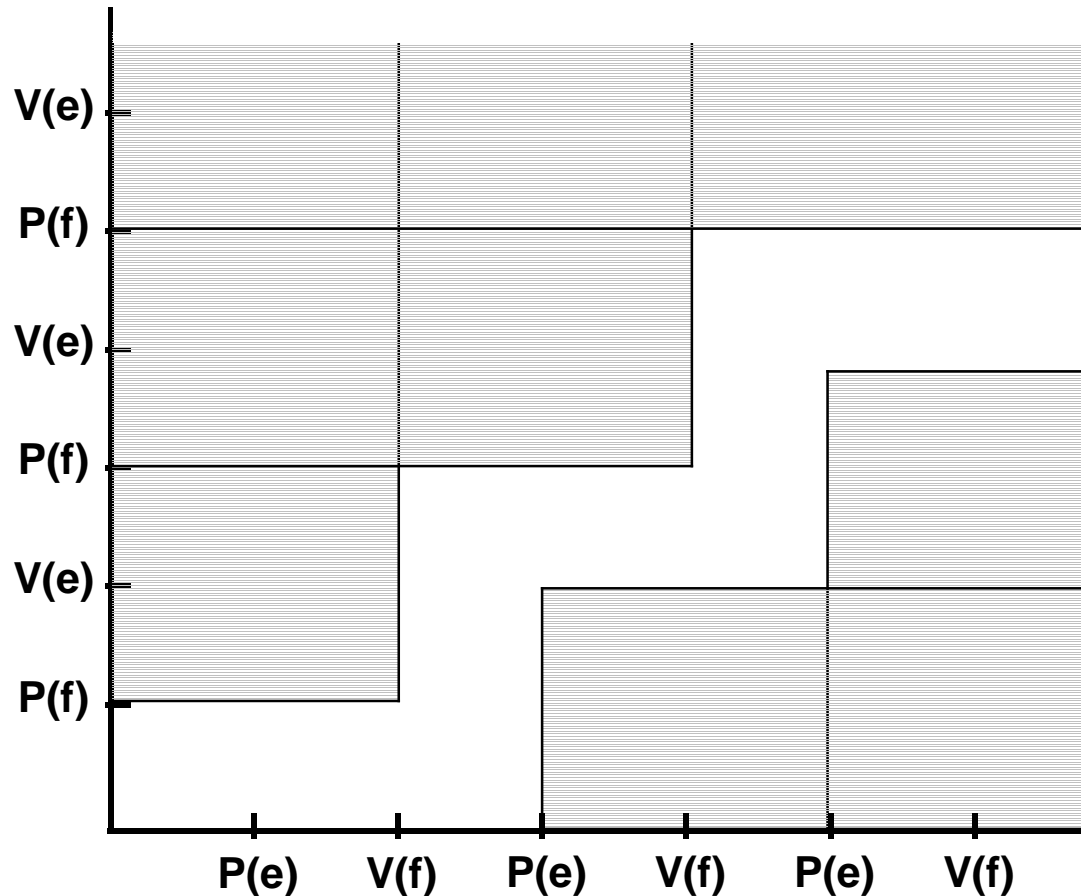
```
/* consumer thread */
void *consumer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);

        /* consume item */
        printf("consumed %d\n",
            item);
    }
    return NULL;
}
```

Producer-consumer progress graph

Consumer



Initially, empty = 1, full = 0.

Producer

The forbidden regions prevent the producer from writing into a full buffer.

They also prevent the consumer from reading an empty buffer.

Problem: Write version for n-element buffer with multiple producers and consumers.

Limitations of semaphores

Semaphores are sound and fundamental, but they have limitations.

- **Difficult to broadcast a signal to a group of threads.**
 - e.g., *barrier synchronization*: no thread returns from the barrier function until every other thread has called the barrier function.
- **Impossible to do timeout waiting.**
 - e.g., wait for at most 1 second for a condition to become true.

For these we must use Pthreads mutex and condition variables.

Basic operations on mutex variables

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                       pthread_mutexattr_t *attr)
```

Initializes a mutex variable (`mutex`) with some attributes (`attr`).

- attributes are usually `NULL`.
- like initializing a mutex semaphore to 1.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Indivisibly waits for `mutex` to be unlocked and then locks it.

- like `P(mutex)`

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Unlocks `mutex`.

- like `V(mutex)`

Basic operations on condition variables

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *attr)
```

Initializes a condition variable (`cond`) with some attributes (`attr`).

- attributes are usually `NULL`.

```
int pthread_cond_signal(pthread_cond_t *cond)
```

Awakens one thread waiting on condition `cond`.

- if no threads waiting on condition, then it does nothing.
- key point: signals are not queued!

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Indivisibly unlocks `mutex` and waits for signal on condition `cond`

- When awakened, indivisibly locks `mutex`.

Advanced operations on condition variables

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Awakens *all* threads waiting on condition `cond`.

- if no threads waiting on condition, then it does nothing.

```
int pthread_cond_timedwait(pthread_cond_t *cond,  
                           pthread_mutex_t *mutex,  
                           struct timespec *abstime)
```

Waits for condition `cond` until absolute wall clock time is `abstime`

- Unlocks `mutex` on entry, locks `mutex` on awakening.
- Use of absolute time rather than relative time is strange.

Signaling and waiting on conditions

Basic pattern for signaling

```
Pthread_mutex_lock(&mutex);  
Pthread_cond_signal(&cond);  
Pthread_mutex_unlock(&mutex);
```

A mutex is always associated with a condition variable.

Guarantees that the condition cannot be signaled (and thus ignored) in the interval when the waiter locks the mutex and waits on the condition.

Basic pattern for waiting

```
Pthread_mutex_lock(&mutex);  
Pthread_cond_wait(&cond, &mutex);  
Pthread_mutex_unlock(&mutex);
```

Barrier synchronization

```
#include <ics.h>

static pthread_mutex_t mutex;
static pthread_cond_t cond;
static int nthreads;
static int barriercnt = 0;

void barrier_init(int n) {
    nthreads = n;
    Pthread_mutex_init(&mutex, NULL);
    Pthread_cond_init(&cond, NULL);
}

void barrier() {
    Pthread_mutex_lock(&mutex);
    if (++barriercnt == nthreads) {
        barriercnt = 0;
        Pthread_cond_broadcast(&cond);
    }
    else
        Pthread_cond_wait(&cond, &mutex);
    Pthread_mutex_unlock(&mutex);
}
```

Call to barrier will not return until every other thread has also called barrier.

Needed for tightly-coupled parallel applications that proceed in phases. E.g., physical simulations.

timebomb.c: timeout waiting example

A program that explodes unless the user hits a key within 5 seconds.

```
#include <ics.h>
#define TIMEOUT 5

/* function prototypes */
void *thread(void *vargp);
struct timespec *maketimeout(int secs);

/* condition variable and
its associated mutex */
pthread_cond_t cond;
pthread_mutex_t mutex;

/* thread id */
pthread_t tid;
```

timebomb.c (cont)

A routine for building a timeout structure for `pthread_cond_timewait`.

```
/*
 * maketimeout - builds a timeout object that times out
 *               in secs seconds
 */
struct timespec *maketimeout(int secs) {
    struct timeval now;
    struct timespec *tp =
        (struct timespec *)malloc(sizeof(struct timespec));

    gettimeofday(&now, NULL);
    tp->tv_sec = now.tv_sec + secs;
    tp->tv_nsec = now.tv_usec * 1000;
    return tp;
}
```

Main routine for timebomb.c

```
int main() {
    int i, rc;

    /* initialize the mutex and condition variable */
    Pthread_cond_init(&cond, NULL);
    Pthread_mutex_init(&mutex, NULL);

    /* start getchar thread and wait for it to timeout */
    Pthread_mutex_lock(&mutex);
    Pthread_create(&tid, NULL, thread, NULL);
    for (i=0; i<TIMEOUT; i++) {
        printf("BEEP\n");
        rc = pthread_cond_timedwait(&cond, &mutex, maketimeout(1));
        if (rc != ETIMEDOUT) {
            printf("WHEW!\n");
            exit(0);
        }
    }
    printf("BOOM!\n");
    exit(0);
}
```


Thread routine for timebomb.c

```
/*  
 * thread - executes getchar in a separate thread  
 */  
void *thread(void *vargp) {  
  
    (void) getchar();  
  
    Pthread_mutex_lock(&mutex);  
    Pthread_cond_signal(&cond);  
    Pthread_mutex_unlock(&mutex);  
    return NULL;  
}
```

Threads summary

Threads provide another mechanism for writing concurrent programs.

Threads are growing in popularity

- Somewhat cheaper than processes.
- Easy to share data between threads.

However, the ease of sharing has a cost:

- Easy to introduce subtle synchronization errors.

For more info:

- man pages (`man -k pthreads`)
- D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997.