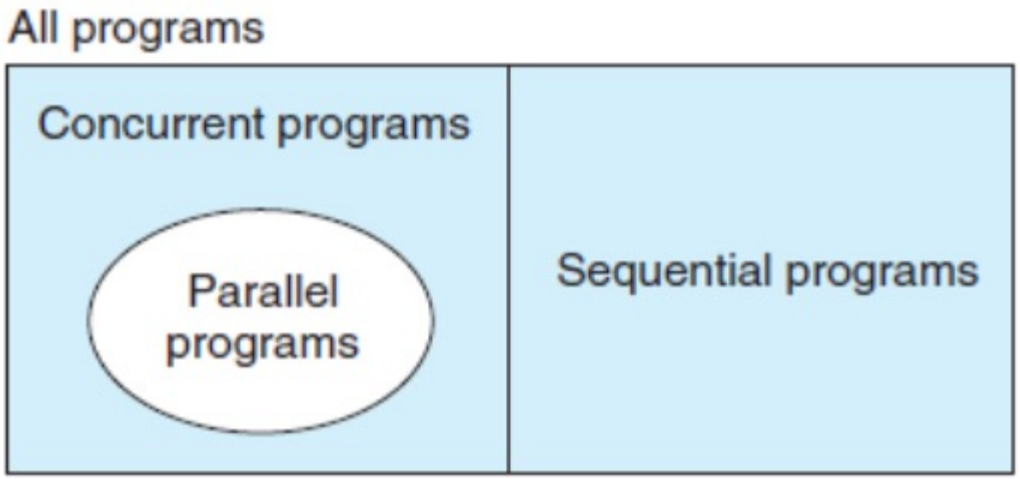
CSE320 System Fundamentals II

Semaphores II

YOUNGMIN KWON / TONY MIONE

Using Threads for Parallelism

A parallel program is a concurrent program running on multiple processors



Synchronization Overhead

```
void *sum_mutex(void *vargp);
void *sum_array(void *vargp);
void *sum_local(void *vargp);

long gsum = 0;
long nelements_per_thread;
sem_t mutex;

int main(int argc, char **argv) {
...
    nelems_per_thread = nelems / nthreads;
    sem_init(&mutex, 0, 1);
    for(i = 0; i < nthreads; i++) {
        myid[i] = i;
        pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
    }

    for (i = 0; i < nthreads; i++)
        pthread_join(tid[i], NULL);
...
}
```

Synchronization Overhead

```
void *sum_mutex(void *vargp) {
    long myid = *(long*)vargp;
    long start = myid * nelements_per_thread;
    long end = start + nelements_per_thread;
    long i;
    for (i = start; i < end; i++) {
        sem_wait(&mutex);
        gsum += i;
        sem_post(&mutex);
    }
    return NULL;
}

// # of threads    :    1    2    3    4    5
// psum-mutex (sec):   68  432  719  552  599
```

Synchronization Overhead

```
long psum[nthreads];

void *sum_array(void *vargp) {
    long myid = *(long*)vargp;
    long start = myid * nelements_per_thread;
    long end = start + nelements_per_thread;
    long i;
    for (i = start; i < end; i++) {
        //sem_wait(&mutex);
        psum[myid] += i;
        //sem_post(&mutex);
    }
    return NULL;
}
// # of threads    :     1     2     3     4     5
// psum-mutex (sec):    68   432   719   552   599
// psum-array (sec):  7.26  3.64  1.91  1.85  1.84
```

Synchronization Overhead

```
void *sum_local(void *vargp) {
    long myid = *(long*)vargp;
    long start = myid * nelements_per_thread;
    long end = start + nelements_per_thread;
    long i, sum = 0;
    for (i = start; i < end; i++) {
        //sem_wait(&mutex);
        sum += i;
        //sem_post(&mutex);
    }
    psum[myid] = sum;
    return NULL;
}
// # of threads    :     1     2     3     4     5
// psum-mutex (sec):    68   432   719   552   599
// psum-array (sec):  7.26  3.64  1.91  1.85  1.84
// psum-local (sec):  1.06  0.54  0.28  0.29  0.30
```

Thread-Unsafe Functions

Four classes of thread-unsafe functions

Class 1

- Functions that do not protect shared variables

```
volatile long cnt = 0;

void* thread(void *vargp) {
    long i, niters = *((long*)vargp);
    for(i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
```

Thread-Unsafe Functions

Class 2

- Functions that keep state across multiple invocations

```
unsigned next_seed = 1;
// after srand, rand returns the same sequence of
// pseudo-random numbers in a single thread, but it doesn't
// when called from multiple threads
unsigned rand(void) {
    next_seed = next_seed * 1103515245 + 12543;
    return (unsigned)(next_seed >> 16) % 32768;
}
void srand(unsigned new_seed) {
    next_seed = new_seed;
}
```

Thread-Unsafe Functions

Class 3

- Functions that return a pointer to a static variable

```
char *ftos(float f) {
    static char str[100];
    sprintf(str, "%f", f);
    return str;
}

void* thread(void *vargp) {
    float f = (float)vargp;
    printf("%s\n", ftos(f));
    return NULL;
}
```

Thread-Unsafe Functions

Class 4

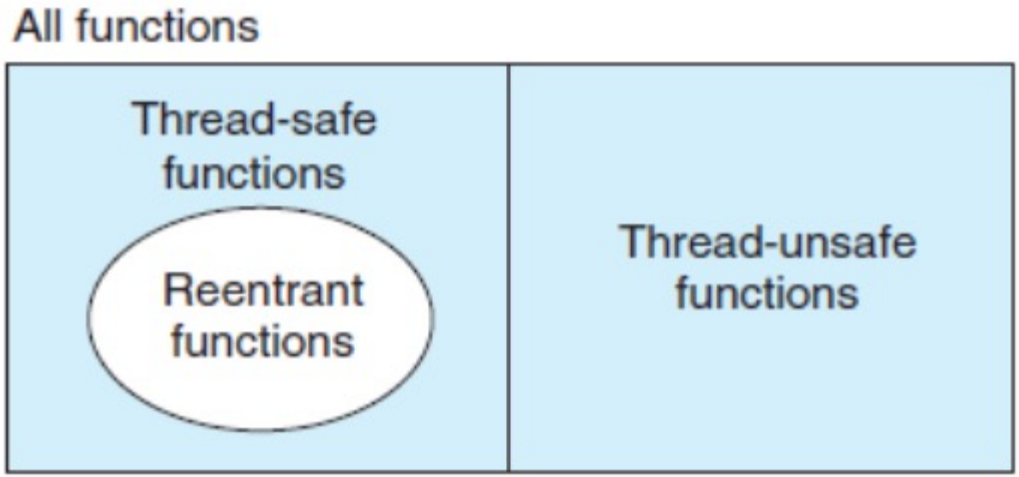
- Functions, say f, that call thread-unsafe functions, say g.
- If g is in class 2, no remedy but to rewrite the function
- If g is in case 1 or in class 3, f can be thread-safe by protecting the shared data with a mutex

```
sem_t mutex;
char *ftos_ts(float f, char *buf) {
    sem_wait(&mutex);
    strcpy(buf, ftos(f));
    sem_post(&mutex);
    return buf;
}
```

Reentrancy

Reentrant functions

- Functions that do not reference any shared data
- Sometimes, thread-safe and reentrant are (incorrectly) used as synonyms



Reentrant functions

Reentrant functions are more efficient than non-reentrant thread-safe functions (no synchronization)

Class 2 type thread-unsafe functions need to be rewritten to reentrant functions to be thread safe

Explicit reentrant function

- All function arguments are passed by values (no pointers) and all data references are local variables

Implicit reentrant functions

- Some parameters in otherwise explicit reentrant functions can be a reference (pointers)
- Be careful not to pass pointers to shared variables

Races

A race occurs when the correctness of a program depends on one thread reaching point x before another thread reaches point y

```
void *thread(void *vargp) {
    int myid = *((int*)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}

int main() {
    pthread_t tid[10];
    int i;

    for(i = 0; i < 10; i++)
        pthread_create(&tid[i], NULL, thread, &i);
    pthread_exit(0);
    return 0;
}
```

Races

```
void *thread(void *vargp) {
    int myid = *((int*)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}

int main() {
    pthread_t tid[10];
    int i, id[10];

    for(i = 0; i < 10; i++) {
        id[i] = i;
        pthread_create(&tid[i], NULL, thread, &id[i]);
    }
    pthread_exit(0);
    return 0;
}
```

Deadlocks

Deadlock

- A collection of threads is blocked, waiting for a condition that will never be true

Deadlock conditions

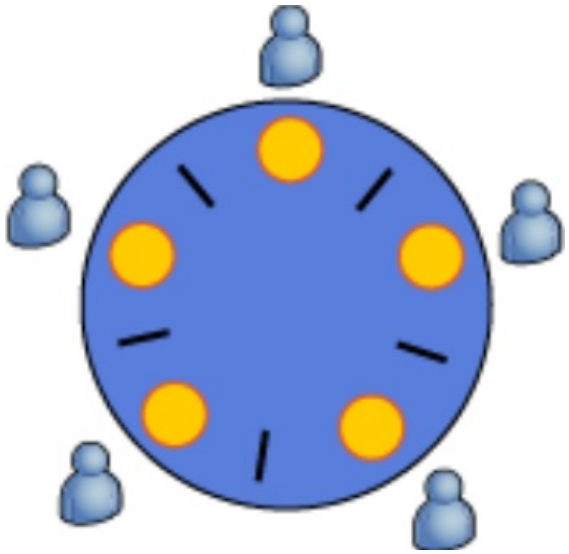
- Mutual exclusion: each resource is assigned to exactly one thread
- Hold and wait: threads currently holding resources can request new resources
- No preemption: resources previously granted cannot be forcefully taken away
- Circular wait: circular chain of two or more threads waiting for the resources held by others

Deadlocks

Dining philosophers problem

- Some philosophers are sitting on a table
- They are thinking and eating when they are hungry
- To eat, they need to pick two chopsticks one on the right and the other on the left

If all philosophers pick the chopstick on their right side, they will starve



Deadlock Example

```
// Dining Philosophers Problem
#define N 3
typedef struct {
    int id;
    sem_t *left;
    sem_t *right;
} Philosopher;
```

Deadlock Example

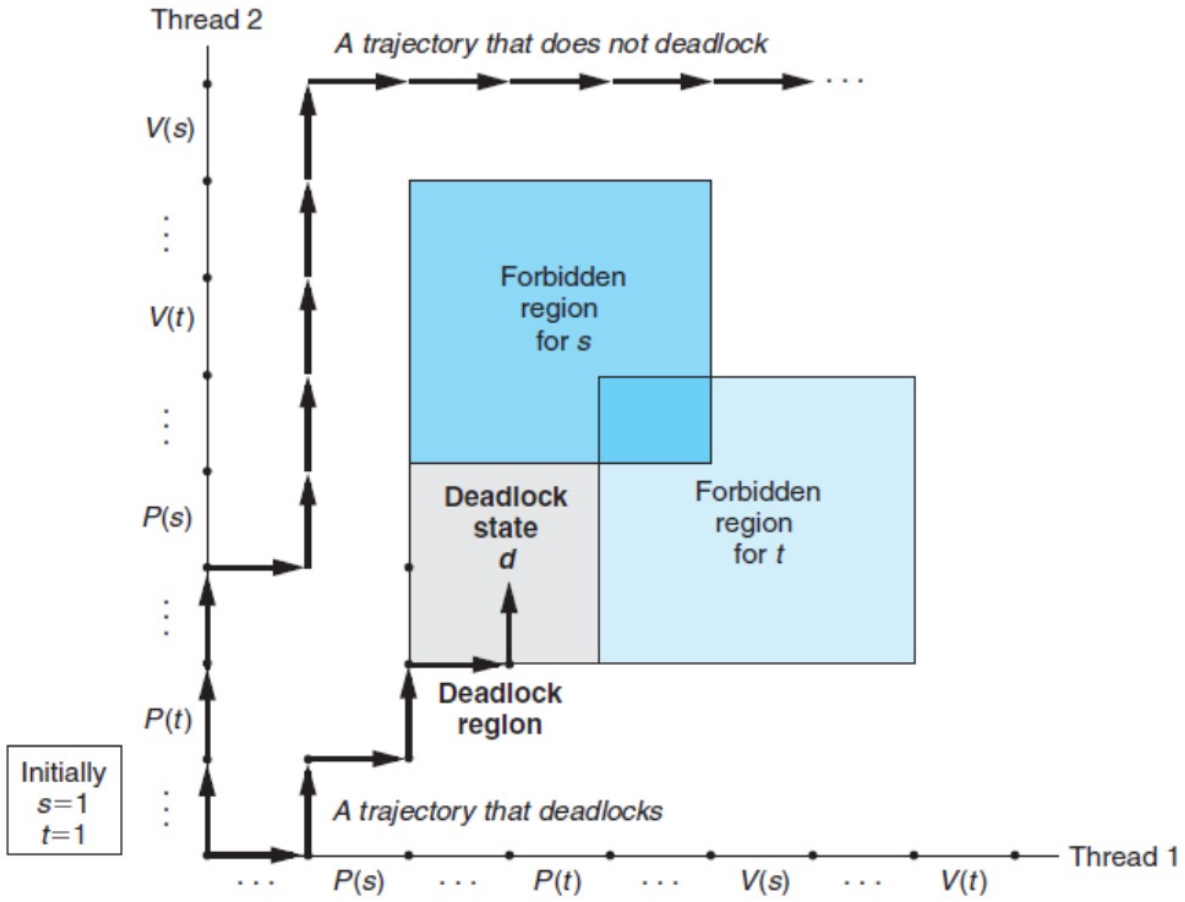
```
void *thread_func(void *vargp) {
    Philosopher *p = (Philosopher*)vargp;
    int i;
    for(i = 0; i < 100; i++) {
        fprintf(stderr, "%d: thinking\n", p->id);
        fprintf(stderr, "%d: getting left\n", p->id);
        sem_wait(p->left);
        fprintf(stderr, "%d: getting right\n", p->id);
        sem_wait(p->right);
        fprintf(stderr, "%d: eating\n", p->id);
        fprintf(stderr, "%d: putting left\n", p->id);
        sem_post(p->left);
        fprintf(stderr, "%d: putting right\n", p->id);
        sem_post(p->right);
    }
}
```

Deadlock Example

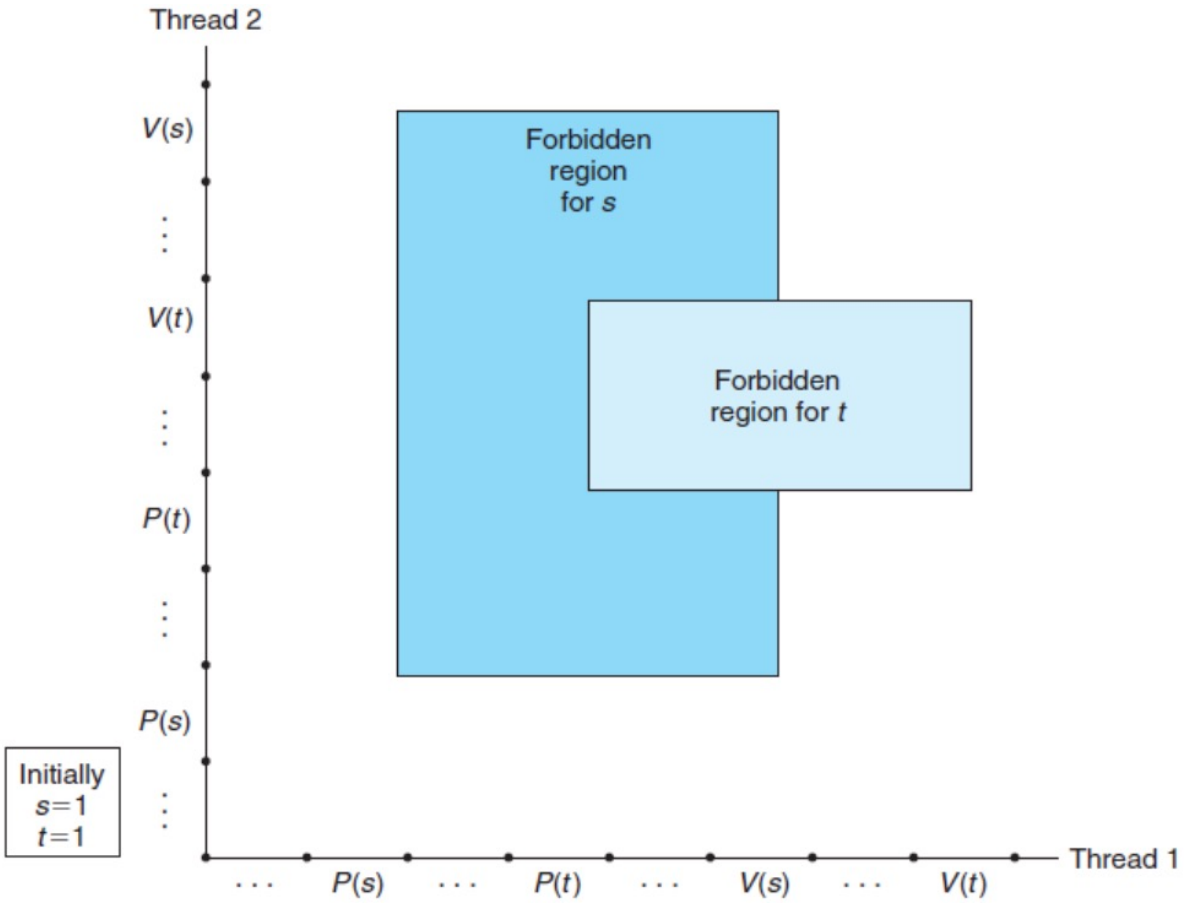
```
int main() {
    pthread_t tid[N];
    sem_t stick[N];
    Philosopher p[N];
    int i;
    for(i = 0; i < N; i++) {
        sem_init(stick+i, 0/*pshared*/, 1/*value*/);
        p[i].id = i;
        p[i].left  = &stick[i % N];
        p[i].right = &stick[(i+1) % N];
    }
    for(i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, thread_func, &p[i]);
    for(i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    for(i = 0; i < N; i++)
        sem_destroy (stick+i);
    return 0;
}

//in gdb, try info threads, thread #, bt
```

Deadlocks (progress graph)



Progress graph for a deadlock free program



Questions?