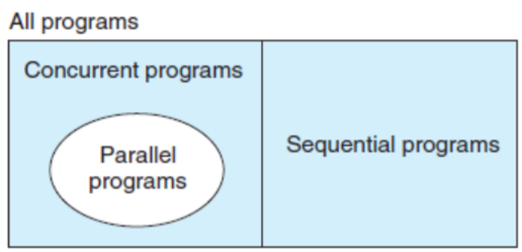
CSE320 System Fundamentals II
Issues in Synchronization

YoungMin Kwon

SUNY Korea
The State University of New York

Using Threads for Parallelism

- A parallel program is a concurrent program running on multiple processors
  - Multi-threading can utilize multiple processors



- Semaphores can be used to synchronize threads, but
  - Computational overheads can be large
    - Semaphores involve system calls

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);
...
}
```

SUNY Korea

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 (4 Classes)

- Class 1
  - Functions that do not protect shared variables

```
volatile long cnt = 0;

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

Thread-Unsafe Functions (4 Classes)

- Class 2
  - Functions that keep state across multiple invocations, even if mutual exclusion is ensured

```
sem_t mutex;
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) {
    sem_wait(&mutex);
    unsigned r = next_seed = next_seed * 1103515245 + 12543;
    sem_post(&mutex);
    return (unsigned)(r >> 16) % 32768;
}
void srand(unsigned new_seed) {
    sem_wait(&mutex);
    next_seed = new_seed;
    sem_post(&mutex);
}
```

Thread-Unsafe Functions (4 Classes)

- Class 3
  - Functions that return a pointer to a static variable

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

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

SUNY Korea

Thread-Unsafe Functions (4 Classes)

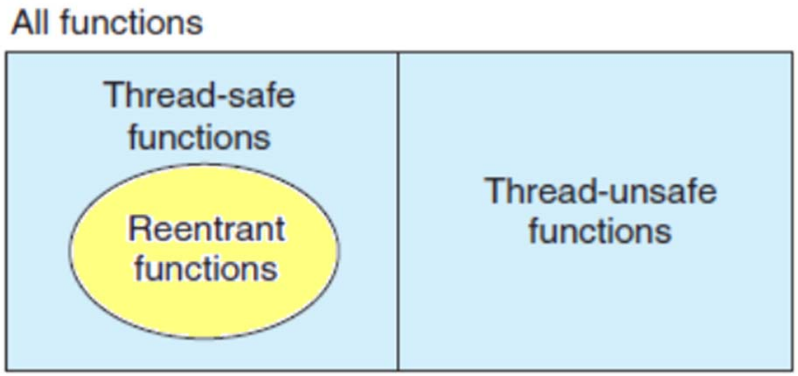
- Class 4
  - Functions, say f, that call thread-unsafe functions, say u.
- How to fix them?
  - If u is in case 1 or in case 3, f can be made thread-safe by protecting the shared data with a **mutex**

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

  - If u is in case 2, no remedy but to rewrite the function
    - E.g. use a stream for the random number generator

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
  - E.g. functional code is reentrant
- 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

- Race
  - the correctness of a program depends on one thread reaching point x before another thread reaches point y

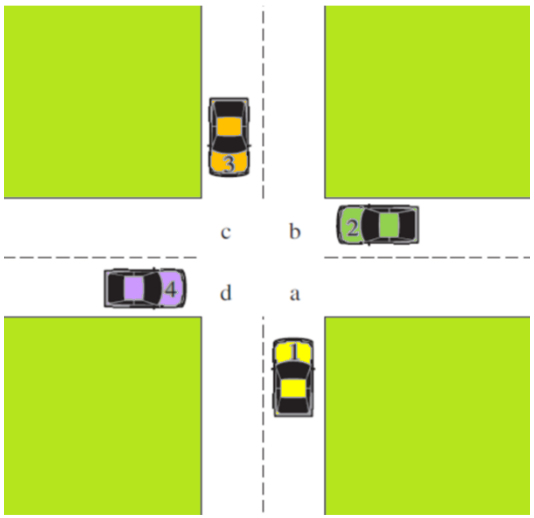
```
void *thread_fun(void *vargp) {
    int myid = *((int*)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}

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

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

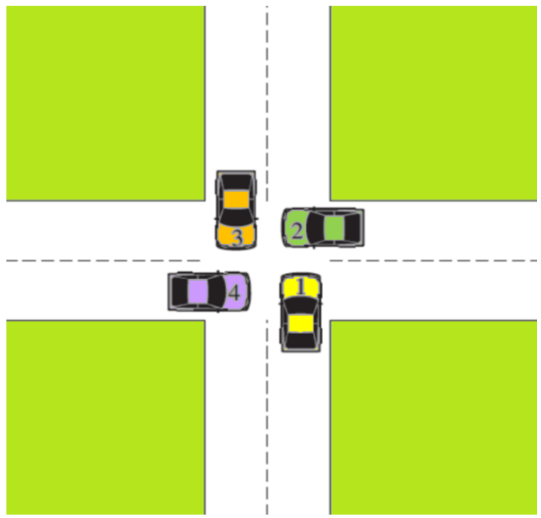
Races

```
void *thread_fun(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_fun, &id[i]);
    }
    pthread_exit(0);
    return 0;
}
```

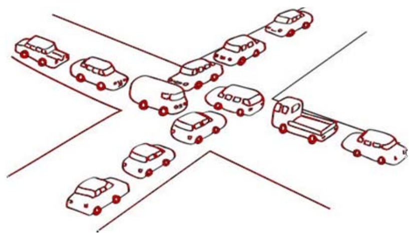
Deadlocks

- Deadlock
  - A collection of threads is blocked, waiting for a condition that will never be true
  - E.g.
    - a, b, c, d are resources and 1, 2, 3, 4 are processes
    - Each process needs two resources in front of it





Deadlocks

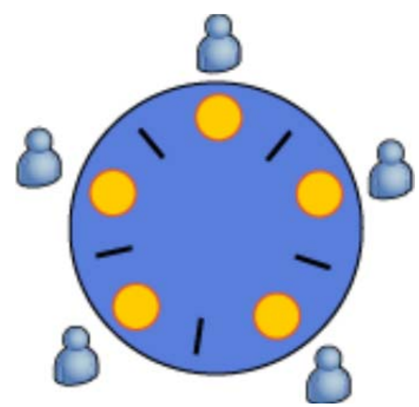


- Deadlock conditions
  - Mutual exclusion: each resource is assigned to exactly one thread
  - Hold and wait: threads currently holding resources can request new resources
  - No preemption: resources previously granted cannot be forcefully taken away
  - Circular wait: circular chain of two or more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

SUNY Korea
The State University of New York

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



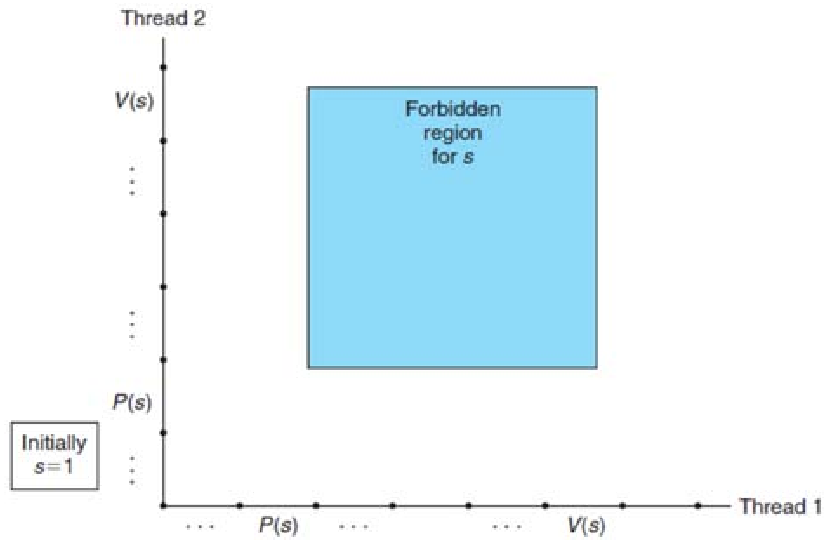
- If all philosophers pick the chopstick on their right side, they will starve

```c
#define N 3                    /*Dining Philosophers Problem*/
typedef struct {
    sem_t lock;
} chopstick_t;

typedef struct {
    int id;
    chopstick_t *left;
    chopstick_t *right;
} philosopher_t;

void *thread_func(void *vargp) {
    philosopher_t *p = (philosopher_t*)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->lock);
        fprintf(stderr, "%d: getting right\n", p->id);
        sem_wait(&p->right->lock);

        fprintf(stderr, "%d: eating\n",        p->id);

        fprintf(stderr, "%d: putting left\n",  p->id);
        sem_post(&p->left->lock);
        fprintf(stderr, "%d: putting right\n", p->id);
        sem_post(&p->right->lock);
    }
}
```

```c
int main() {
    pthread_t tid[N];
    chopstick_t stick[N];
    philosopher_t p[N];
    int i;
    for(i = 0; i < N; i++) {
        sem_init(&stick[i].lock, 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].lock);
    return 0;
}

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

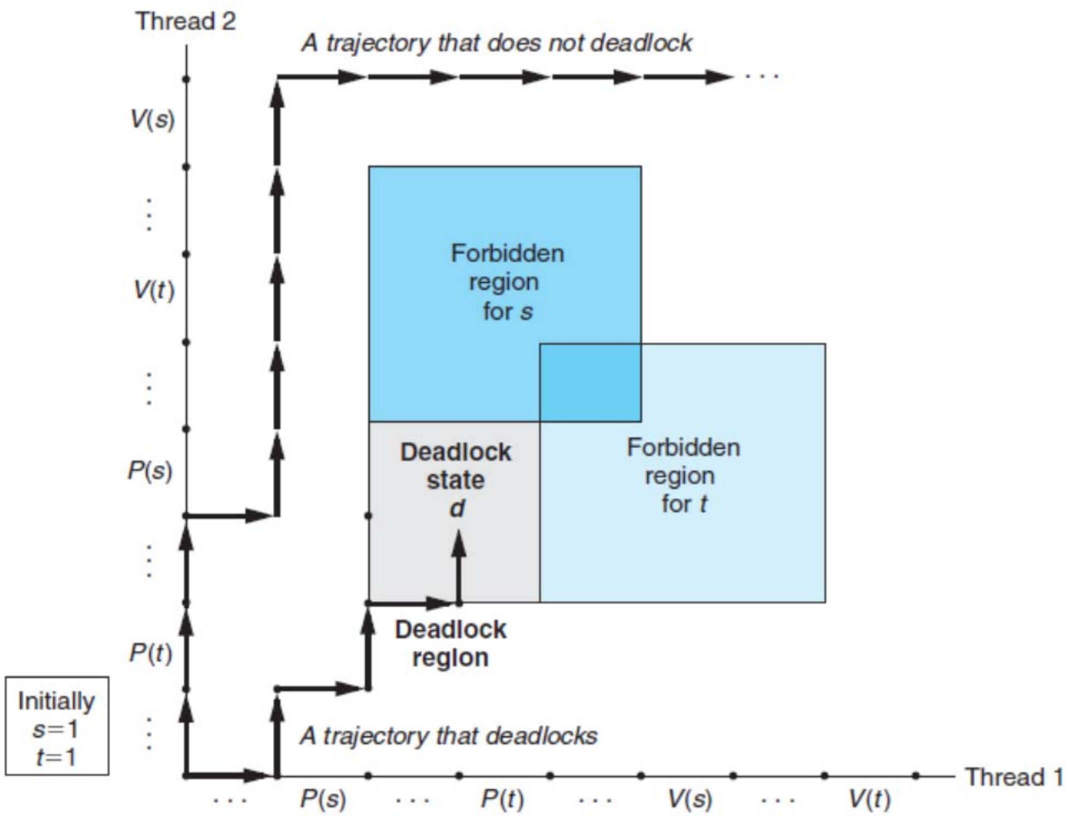
SUNY Korea

Deadlocks (Progress Graph)

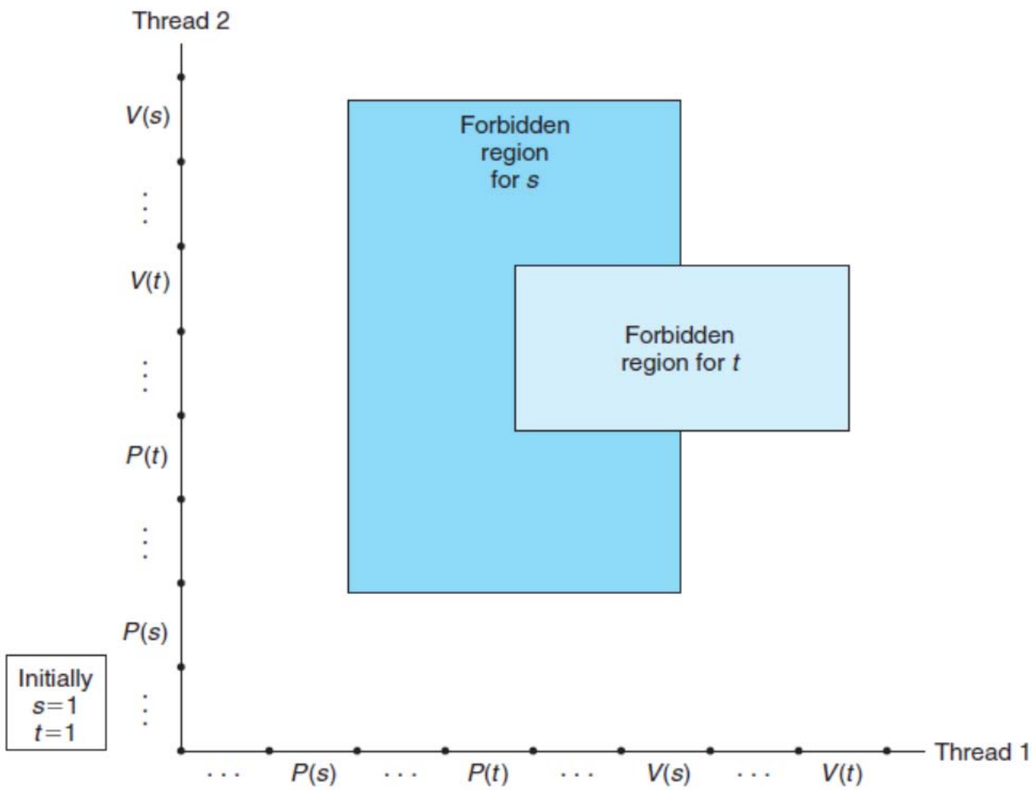


- Progress Graph
  - State: a pair of the states of thread1 and thread2
  - E.g. (0,0) → (2,0) → (2,1) → (3,1) → (3,2) ...

Deadlocks (Progress Graph)



Deadlocks (Progress Graph)



Using Locks in Interrupt Handlers

- Using semaphore in a signal handler (interrupt handler)
  - Signal handler runs before the control returns to the normal flow (signal handler has higher priority)
  - Deadlock: if a lock is acquired in the normal flow and then a signal handler tries to acquire the lock
    - Normal flow will not run (and release the lock) until the handler is finished
    - The handler cannot acquire the lock until the normal flow releases it
  - ⇒ disable the interrupt while accessing the shared resource

```c
#include <semaphore.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

sem_t mutex;
volatile long count = 0;
void handler(int sig) {
    sem_wait(&mutex);
    count = 0;
    sem_post(&mutex);
}
int main() {
    signal(SIGALRM, handler);

    sem_init(&mutex, 0, 1);
    while(1) {
        if(count == 0)
            alarm(1);
        sem_wait(&mutex);
        count++;
        printf("count = %ld\n", count);
        sem_post(&mutex);
    }
    sem_destroy(&mutex);

    return 0;
}

//in gdb, try bt
```

Programming Exercise

- Fix the dining philosophers problem by giving orders to the resources and make requests in that order
  - Implement the TODO block in the next slide

```c
// A solution for the dining philosophers problem
//

typedef struct {
    //TODO: add an id field to order the resource
    sem_t lock;
} chopstick_t;

void *thread_func(void *vargp) {
    philosopher_t *p = (philosopher_t*)vargp;
    int i;
    for(i = 0; i < 100; i++) {
        fprintf(stderr, "%d: thinking\n",      p->id);

        //TODO: break the circular wait by acquiring resources
        //in an order (e.g. resource id)

        fprintf(stderr, "%d: eating\n",        p->id);

        fprintf(stderr, "%d: putting left\n",  p->id);
        sem_post(&p->left->lock);
        fprintf(stderr, "%d: putting right\n", p->id);
        sem_post(&p->right->lock);
    }
}
```