Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (pthreads)
- 28 - Locks
- 29 - Concurrent Data Structures
- 30 Condition Variables
- 31 - Semaphores
- 32 - Common Problems (including deadlocks)
- 33 - Event-Based Concurrency

Semaphores

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

- wait()
  - decrement value by one
  - wait if resulting value negative
- post()
  - increment value by one
  - if one or more threads waiting: wake one

The value, when negative, is the number of waiting threads

Semantics

- mutex locks
  - "binary semaphore"
  - lock by calling wait()
  - unlock by calling post()
  - initial value of 1
- ordering primitive (like a condition variable)
  - "counting semaphore"
  - parent waiting for child, sharing a semaphore
    - parent calls wait()
    - child calls post()
    - initial value? 0

In general, how to determine the initial value?

- how many of your resources you are willing to give out?

Producer-Consumer *back to the basics*

```
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use];    // Line G1
    use = (use + 1) % MAX;    // Line G2
    return tmp;
}
```

Figure 31.9: **The Put And Get Routines**

Prod-Cons *semaphores*

Assume MAX = 1,
initially empty,
multiple consumers and producers

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);         // Line C1
        tmp = get();             // Line C2
        sem_post(&empty);        // Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

Figure 31.10: **Adding The Full And Empty Conditions**

all good!

Prod-Cons *semaphores, flawed*

Assume MAX = 10,
initially empty,
multiple consumers and producers

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);         // Line C1
        tmp = get();             // Line C2
        sem_post(&empty);        // Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

Figure 31.10: **Adding The Full And Empty Conditions**

Problem is we are not enforcing mutual exclusion over the put() and get().

Need to add mutual exclusion back in!

Prod-Cons *semaphores, fixed*

Deadlock!

empty buffer
consumer runs, blocks
producer runs, blocks

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line P0 (NEW LINE)
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
        sem_post(&mutex);        // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line C0 (NEW LINE)
        sem_wait(&full);         // Line C1
        int tmp = get();         // Line C2
        sem_post(&empty);        // Line C3
        sem_post(&mutex);        // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```


Prod-Cons *semaphores fixed again*

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        // Line P1
        sem_wait(&mutex);        // Line P1.5 (MUTEX HERE)
        put(i);                  // Line P2
        sem_post(&mutex);        // Line P2.5 (AND HERE)
        sem_post(&full);         // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);         // Line C1
        sem_wait(&mutex);        // Line C1.5 (MUTEX HERE)
        int tmp = get();         // Line C2
        sem_post(&mutex);        // Line C2.5 (AND HERE)
        sem_post(&empty);        // Line C3
        printf("%d\n", tmp);
    }
}
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

Reader-writer Locks

Either

- one or more readers, or
- a single writer

may be in the critical section at one time.

Reader-writer Locks *via semaphores*

Issues?
How to fix?

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}
```

readers

```
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```

writer

```
void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Figure 31.13: **A Simple Reader-Writer Lock**

Baboons and the River *by semaphore*

```
sem_t               rope_capacity;
sem_t               east_mutex, west_mutex;
pthread_mutex_t     mutex; // shared

int                 east_count = 0;
int                 west_count = 0;


int main() {
    // Initialize semaphores
    sem_init(&rope_capacity, 0, 3);
    sem_init(&east_mutex, 0, 1);
    sem_init(&west_mutex, 0, 1);
    pthread_mutex_init(&mutex, NULL);
        …
        pthread_create(&baboons[i], NULL, eastward_baboon, NULL);
        …
```


Baboons and the River *by semaphore*

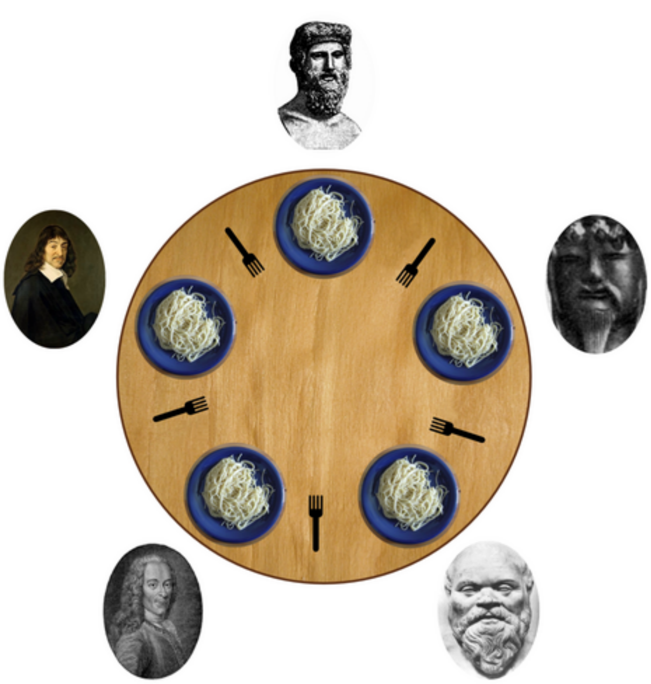
```
int main() {
    // Initialize semaphores
    // Semaphores
    sem_t east, west, dir, rope;
    int easts = 0, wests = 0;
    sem_init(&east, 0, 1);
    sem_init(&west, 0, 1);
    sem_init(&dir, 0, 1);
    sem_init(&rope, 0, 3);

    ...



void *east_baboon(void *arg) {
    sem_wait(&east);
    if (++easts == 1) {
        sem_wait(&dir);  // Block westward movement
    }
    sem_post(&east);

    sem_wait(&rope);  // Ensure at most 3 baboons on the rope
    printf("Baboon going EAST…\n”);
    sleep(1);  // Simulate crossing
    sem_post(&rope);

    sem_wait(&east);
    if (--easts == 0) {
        sem_post(&dir);  // Allow westward movement if no eastward baboons left
    }
    sem_post(&east);

    return NULL;
}

void *west_baboon(void *arg); // symmetric
```

Baboons and the River *by semaphore*

What about starvation?

- could use a synchronized queue
- all baboons, east and west, go onto queue in order of arrival
- baboon pops off queue if same polarity as those on rope, *and* if there are fewer than three currently on the rope

Dining Philosophers! *semaphores*



```
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
```

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

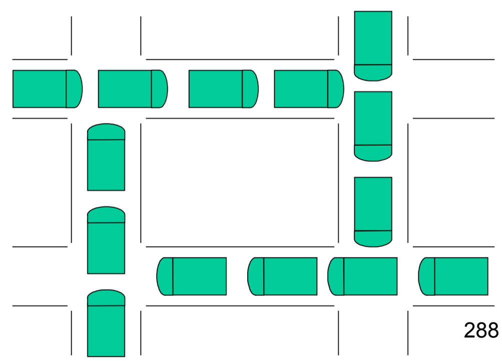
What could go wrong?

- deadlock
- cause: symmetry
- fix: asymmetry

pic from Wikipedia

Deadlocks *more generally*

- Necessary conditions for deadlock
  - **Mutual exclusion** - Threads claim exclusive control of resources (*binary semaphores*)
  - **Hold and wait** - Threads hold resources while waiting for additional resources (*semaphore waits*)
  - **No preemption** - Resources cannot be removed from threads that hold them (*semaphores cannot be taken by force*)
  - **Circular wait** - There exists a chain of threads such that each holds one or more resources that are requested by the next thread in the chain (*philosophers*)
- What to do?
  - prevent
  - avoid
  - deal with when they occur
  - pretend they never happen



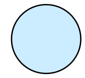
Resource Allocation Graph

A set of vertices V and a set of edges E:

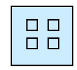
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set of all resource types in the system
- request edge: directed edge $P_i \rightarrow R_j$
- assignment edge: directed edge $R_j \rightarrow P_i$

Resource Allocation Graph *(cont.)*

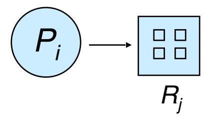
- Process



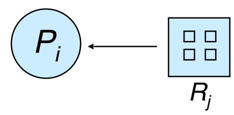
- Resource type with 4 instances



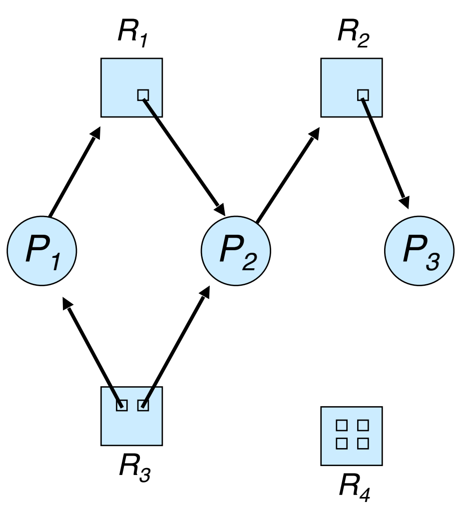
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Resource Allocation Graph *example*



- P_1 requesting instance of R_1
- one R_1 held by P_2
- P_2 requesting instance of R_2
- one R_2 held by P_3
- distinct R_3 instances held by P_1 and P_2