COMP 206 – Software Systems

Lecture 22 – November 28, 2018

Coordination of multi process programs

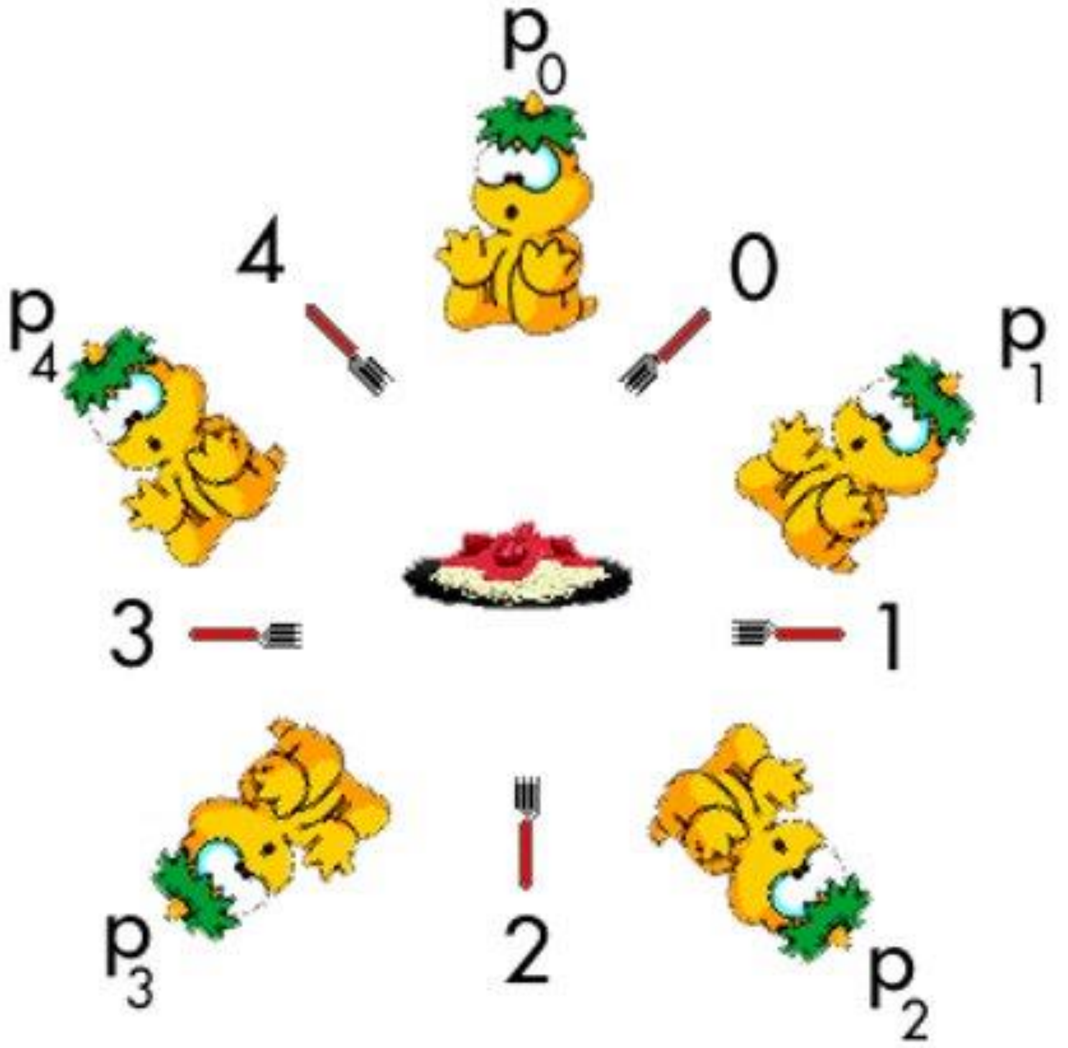
Last time

- Our processes' access to the CPU is through the scheduler

- We cannot guarantee what order jobs will run in, which can lead to bad outcomes when sharing a task:
  - "add_ten.bash" actually sets the file back to 1!
  - The Dining Philosophers can end up starving unless they have a good plan

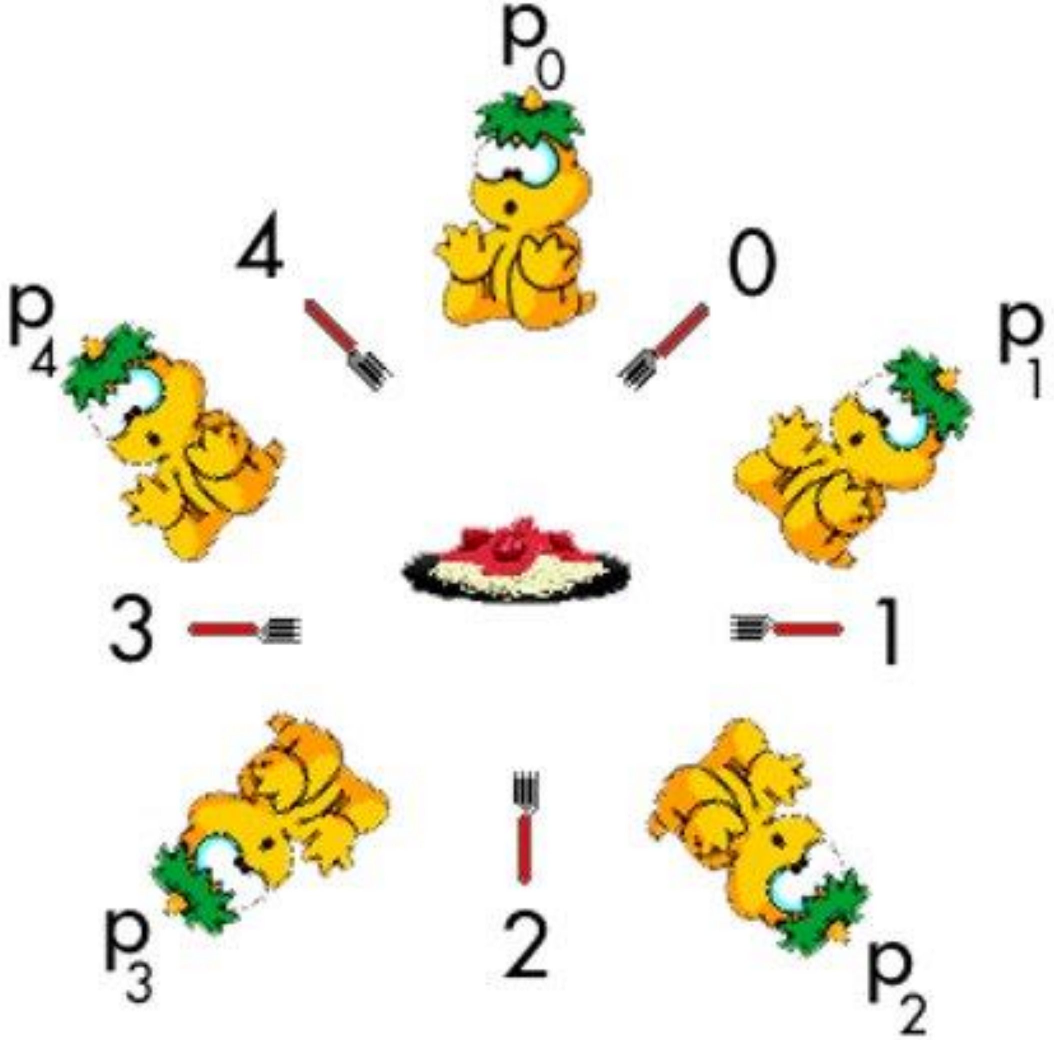
Dining Philosophers again

- Why did we play this goofy game?
  - Philosophers are processes
  - Sticks are resources (a file, a hardware component, etc)
  - Eating means being able to do our desire work: read from a camera and post our selfie online
- The solutions to Dining Philosophers have a relation to coordination mechanisms in software systems



Dining Philosophers solutions

- The key is to avoid the "one stick each" situation, and any equivalent state (such as alternating pick-up/put-down forever)
- We need a "tie-breaker". Lets you guarantee someone will get both sticks.
  - Recall: Numbering chopsticks works
  - Recall: Calling a "waiter" also works
  - Break the rules: communication would also work



Tie-breaking with semaphores

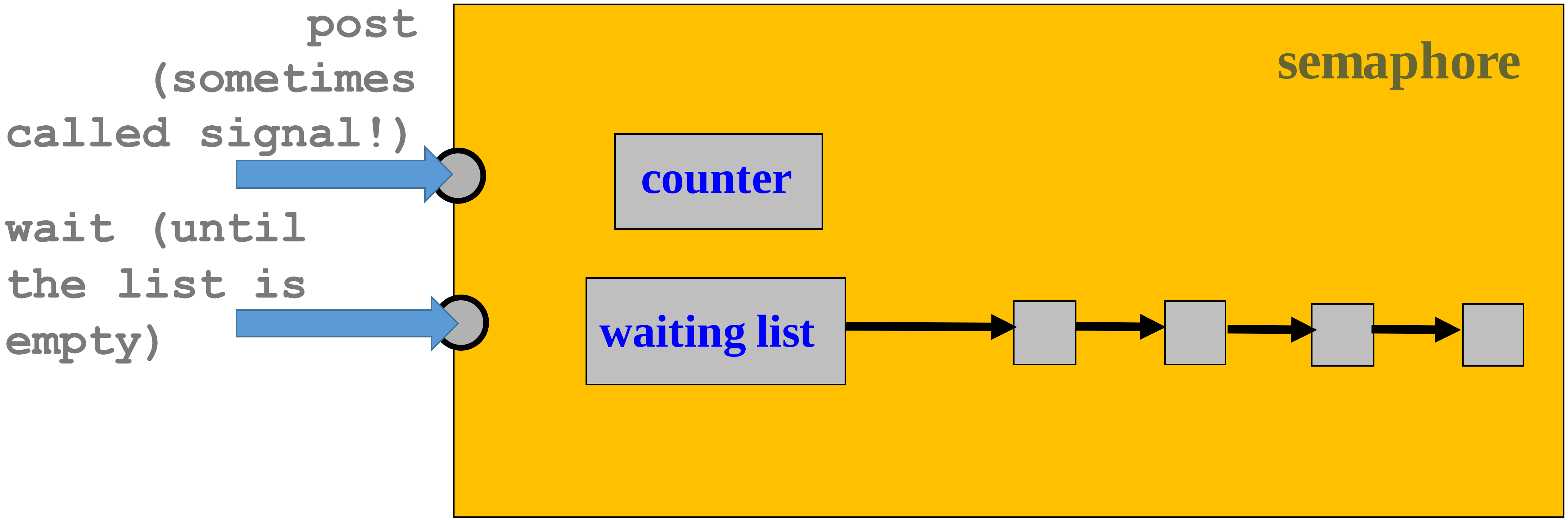
- Idea: have a central agent (the operating system) keep track of who asked first and only let one process execute at a time

- The process that asks first gets to proceed right away, and must then also say when it's finished

- All other processes are blocked by the central agent, and allowed to proceed one at a time.

Semaphores

- A *semaphore* is an object that consists of a **counter**, a **waiting list** of processes and two **methods** (*e.g.*, functions): `post` and `wait`.



Semaphore Method: wait

```
void wait(sem S){
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list ;
        block();
}}
```

- After decreasing the counter by 1, if the counter value becomes negative, then
  - add the caller to the waiting list, and then
  - block the caller.
  - if all processes call wait(), only "the initial value of count" can run at once!

Semaphore Method: post

```
void post(sem S){
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume( P);
}}
```

- After increasing the counter by 1, if the new counter value is not positive, then
  - remove a process **P** from the waiting list,
  - resume the execution of process **P**, and return
  - recall wait() let only "initial count" jobs start. Post means "I finished", the next in line can start.

Important Note: 1/4

wait

```
S.count--;
if (S.count<0) {
    add to list ;
    block();
}
```

post

```
S.count++;
if (S.count<=0) {
    remove P;
    resume(P);
}
```

- If `S.count < 0`, `abs(S.count)` is the number of waiting processes.
- This is because processes are added to (*resp*., removed from) the waiting list only if the counter value is `< 0` (*resp*., `<= 0`).

Important Note: 2/4

wait

```
S.count--;
if (S.count<0) {
     add to list ;
     block();
}
```

post

```
S.count++;
if (S.count<=0) {
     remove    P;
     resume(   P);
}
```

- The waiting list can be implemented with a queue if FIFO order is desired.
- However, the correctness of a program should not depend on a particular implementation of the waiting list.
- Your program should not make any assumption about the ordering of the waiting list.

Important Note: 3/4

wait

```
S.count--;
if (S.count<0) {
    add to list ;
    block();
}
```

post

```
S.count++;
if (S.count<=0) {
    remove P;
    resume(P);
}
```

- The caller may be blocked in the call to `wait()`.
- The caller never blocks in the call to `post()`. If `S.count > 0`, `post()` returns and the caller continues. Otherwise, a waiting process is released and the caller continues. In this case, *two* processes continue.

The Most Important Note: 4/4

wait

```
S.count--;
if (S.count<0) {
    add to list ;
    block();
}
```

post

```
S.count++;
if (S.count<=0) {
    remove P;
    resume(P);
}
```

- `wait()` **and** `post()` **must be executed** ***atomically*** **(*i.e.*, as one uninterruptible unit).**
- **Otherwise, *race conditions* may occur.**
- ***Exercise:*** **use execution sequences to show race conditions if** `wait()` **and/or** `post()` **is not executed atomically.**

Three Typical Uses of Semaphores

- There are three typical uses of semaphores:
  - **mutual exclusion**:

    Mutex (*i.e.*, *Mut*ual *Ex*clusion) locks
  - **count-down lock**:

    Keep in mind that semaphores have a counter.
  - **notification**:

    Indicate an event has occurred.

Use 1: Mutual Exclusion (Lock)

initialization is important

```
sem S = sem_init(1);
int       count = 0;
```

Process 1

```
while (1) {
    // do something
    wait(S);
        count++;
    post(S);
    // do something
}
```

Process 2

```
while (1) {
    // do something
    wait(S);
        count--;
    post(S);
    // do something
}
```

entry

critical sections

exit

- **Question: What if the initial value of `S` is zero?**
- `S` **is a *binary semaphore* (similar to a *lock*).**

Use 2: Count-Down Counter

```
sem S = sem_init(3);
```

Process 1	Process 2
`while (1) {`	`while (1) {`
`// do something`	`// do something`
`wait(S);`	`wait(S);`

at most 3 processes can be here!!!

`post(S);`	`post(S);`
`// do something`	`// do something`
`}`	`}`

- After **three** processes pass through `wait()`, this section is locked until a process calls `post()` .

Use 3: Notification

```
sem S1 = sem_init(1), S2 = sem_init(0);
```

process 1

```
while (1) {
    // do something
    wait(S1);
        printf("1");
    post(S2);
    // do something
}
```

notify (wait(S1)) · **notify** (post(S2))

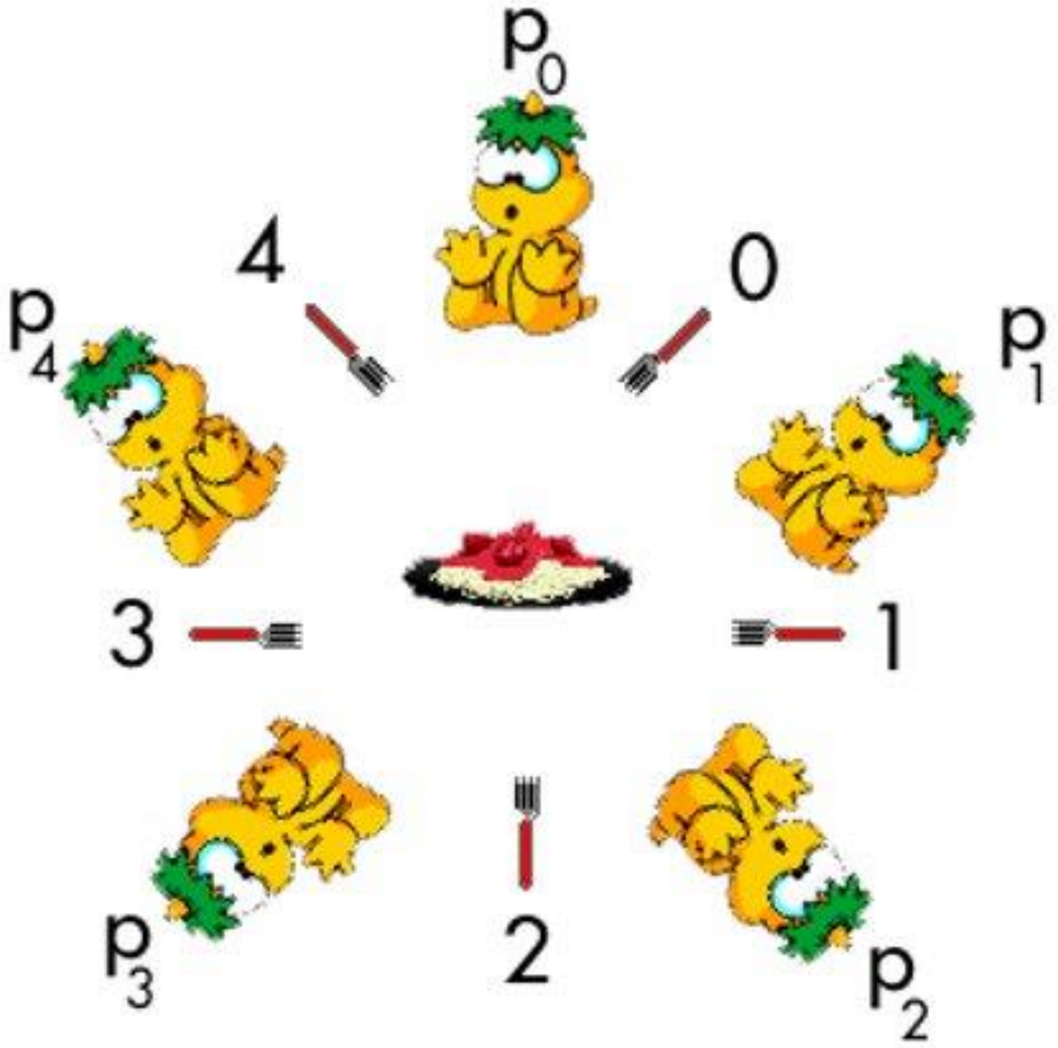
process 2

```
while (1) {
    // do something
    wait(S2);
        printf("2");
    post(S1);
    // do something
}
```

- **Process 1 uses `post(S1)` to notify process 2, indicating "I am done. Please go ahead."**
- **The output is `1 2 1 2 1 2` ......**
- **What if both `S1` and `S2` are both 0's or both 1's?**
- **What if `S1` = 0 and `S2` = 1?**

Dining Philosophers With Semaphors

- We have the tool now!
- Can you think of how to apply the wait() and signal() operations of a semaphor to ensure the Dining Philosophers can eat in peace?
  - How many semaphors to use?
  - What code to run on each Philosopher?



Dining Philosopher: Ideas

- Chopsticks are shared items (by two philosophers) and must be protected.
- Each chopstick has a semaphore with initial value 1.
- A philosopher calls `wait()` before they pick up a chopstick and calls `post()` to release it.

```
sem C[5];
// init all to 1;
```

outer critical section

inner critical section

```
wait(C[i].);
wait(C[(i+1)%5]);
```

has 2 chops and eats

```
post(C[(i+1)%5]);
post(C[i]);
```

Dining Philosophers: Proposal

```
sem C[5]; // init all to 1;
```

For all philosophers i *(in parallel)*

```
while (1) {
    // thinking
    wait(C[i]);
    wait(C[(i+1)%5]);
    // eating
    post (C[(i+1)%5]);
    post (C[i]);
    // finishes eating
}
```

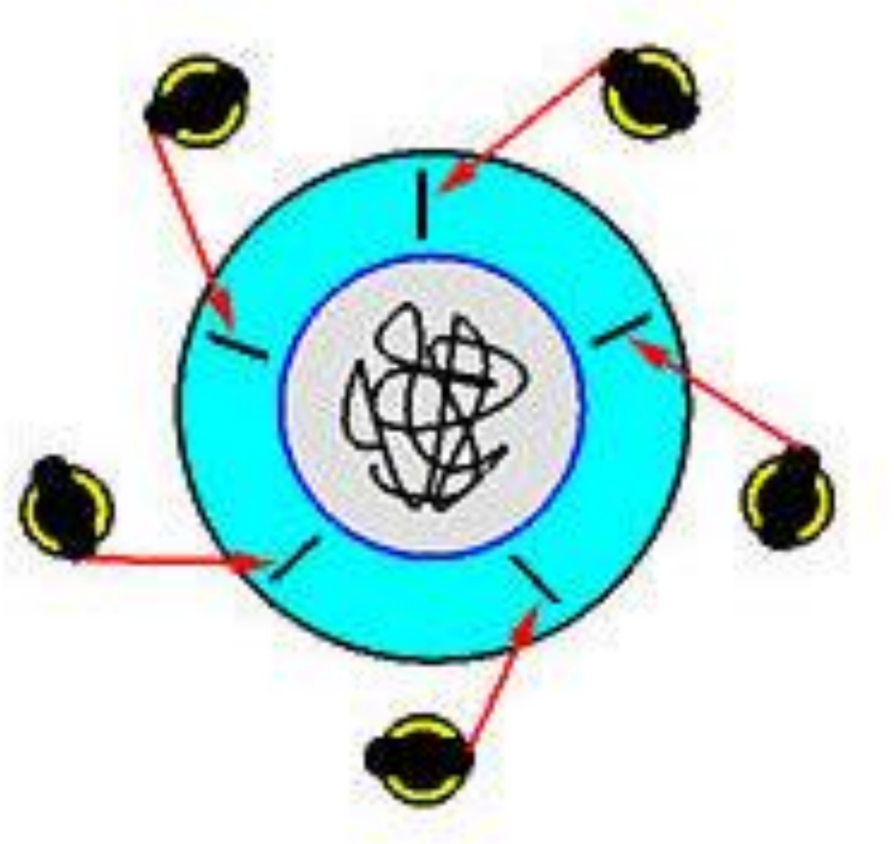
wait for my left chop
wait for my right chop

release my right chop
release my left chop

Does this solution work?

Dining Philosophers: Deadlock!

- No! This was the first solution we saw that didn't work.
- The semaphors here ensure the consistency of the game: no chopstick shared
- But, the coordination ***protocol*** ends in ***deadlock***



Dining Philosophers: A Better Idea

```
sem C[5]; // init all to 1;
```

philosopher *i* (0, 1, 2, 3)

```
while (1) {
    // thinking
    wait(C[i]);
    wait(C[(i+1)%5]);
    // eating
    post(C[(i+1)%5]);
    post(C[i]);
    // finishes eating;
}
```

lock left chop first

Philosopher 4: the weirdo

```
while (1) {
    // thinking
    wait(C[(i+1)%5]);
    wait(C[i]);
    // eating
    post(C[i]);
    post(C[(i+1)%5]);
    // finishes eating
}
```

lock right chop first

Dining Philosophers: Questions

- **The following are some important questions to think over.**
  - **We choose philosopher 4 to be the weirdo. Does this choice matter?**
  - **Show that this solution does not cause *circular waiting*.**
  - **Show that this solution will not have *circular waiting* if we have more than 1 and less than 5 weirdoes.**

Semaphores in C/Linux

- Functionality included in:
  - #include <semaphore.h>
  - #include <fcntl.h>
- Sits on top of the system call ***semctl()***
  - We will not look at that directly, only the easier C interface
- Allows the use of wait() and post() between processes and Linux. These tools and some experience give us full control over process ordering!
- More info: http://man7.org/linux/man-pages/man7/sem_overview.7.html

Synchronized counting with semaphores

- Recall our fork() example from last class that counted to 200. The two processes did not stay in synch. Can you think of how to synchronize them so we see: "1, 1, 2, 2, 3, 3, …. , 199, 199, 200, 200" each time?

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#define MAX_COUNT 200
#define BUF_SIZE 100
void main(void)
{
    pid_t pid;
    int i;
    char buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf,
                "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

```c
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>

#include <semaphore.h>
#include <fcntl.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

sem_t *s1;
sem_t *s2;

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     s1 = sem_open( "first", O_CREAT, 0666, 1 );
     s2 = sem_open( "second", O_CREAT, 0666, 0 );

     int x = fork();
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {

          if( x == 0 ){
            sem_wait( s1);
          }
          else{
            sem_wait( s2 );
          }

          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));

          if( x == 0 ){
            sem_post( s2);
          }
          else{
            sem_post( s1 );
          }
     }
}
```

Solution:

This line is from pid 22918, value = 1

This line is from pid 22917, value = 1

This line is from pid 22918, value = 2

This line is from pid 22917, value = 2

…

This line is from pid 22918, value = 199

This line is from pid 22917, value = 199

This line is from pid 22918, value = 200

This line is from pid 22917, value = 200

Solution:

- This is the "notification" pattern:
  - Each side has to wait until the other has posted to advance
  - We start the semaphores at 0 and 1 to specify explicitly who gets to run first
  - Afterwards, the order is not guaranteed, but the fact that they stay within 1 number is

```c
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>

#include <semaphore.h>
#include <fcntl.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

sem_t *s1;
sem_t *s2;

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     s1 = sem_open( "first", O_CREAT, 0666, 1 );
     s2 = sem_open( "second", O_CREAT, 0666, 0 );

     int x = fork();
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {

          if( x == 0 ){
            sem_wait( s1);
          }
          else{
            sem_wait( s2 );
          }

          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));

          if( x == 0 ){
            sem_post( s2);
          }
          else{
            sem_post( s1 );
          }
     }
}
```

Further thoughts

- Programming languages implement many additional features to allow mutual exclusion, coordination etc.
- We will sometimes see Linux semaphores directly in your C code, but more often they will be hidden within other features :
  - Shared memory (exclusion tied directly to memory access)
  - Mutex (implements only the binary lock for critical sections)
  - Scoped lock (no need to call init or signal, the variable's lifetime on the stack controls the exclusion)
- However, the concepts of all these are the same and now we've seen under the hood a bit. Let's think back on the key elements.

Coordination Summary Points

- Processes must ***actively synchronize*** due to unpredictable scheduling
- ***Critical sections*** are portions of shared/parallel code where it is essential only a single process should be operating at a time:
  - File reading/writing, updating memory locations, eating spaghetti
- A ***race condition*** is a situation where performance depends on the ordering of processes. We must avoid these!
- Semaphores are one coordination construct: wait(), post(), a counter and a list of waiting processes
- The key to implementing semaphores (and all other coordination) is that all calls are ***atomic operations*. **This means behavior is guaranteed even if 2 processes call the function at exactly the same time (or close). Atomic operations ***break ties!***