# Operating Systems

Inter-process Communication (IPC)

Deadlock



## Coordination and Deadlock

- Process coordination can lead to *deadlock*
  - With blocking/synchronous application interfaces, processes may wait forever
- Obviously Process A can't wait on a message from B, while B waits on a message from A, or they will be waiting forever
  - This can occur with mutexes/semaphores as well

```
thread 1                    thread 2
--------                    --------
  . . .                       . . .
pthread_mutex_lock(&a)      pthread_mutex_lock(&b)
pthread_mutex_lock(&b)      pthread_mutex_lock(&a)
  . . .                       . . .
```



## Deadlock Definitions

- A set of processes is deadlocked when every process in the set is waiting for an event that can only be generated by some process in the set
- Livelock is similar but the states of the processes are changing
  - Still no process is actually progressing
- Contrast with starvation: a process waits indefinitely because some other process is using a resource
  - Deadlock and livelock are instances of starvation, but starvation is more general



## Standard Problem – Dining Philosophers

- Acquire left and right forks/chopsticks to eat pasta/rice
- Get utensils, eat, drop utensils, think, repeat

# Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
While (true) {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

              // eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

              // think

}
```

5

# Dining Philosophers (3)

```
#define N            5              /* number of philosophers */
#define LEFT         (i+N–1)%N      /* number of i's left neighbor */
#define RIGHT        (i+1)%N        /* number of i's right neighbor */
#define THINKING     0              /* philosopher is thinking */
#define HUNGRY       1              /* philosopher is trying to get forks */
#define EATING       2              /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}
```

Solution to dining philosophers problem (part 1) - Tannenbaum

6

# Dining Philosophers (4)

```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
    test(i);                        /* try to acquire 2 forks */
    up(&mutex);                     /* exit critical region */
    down(&s[i]);                    /* block if forks were not acquired */
}
void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = THINKING;            /* philosopher has finished eating */
    test(LEFT);                     /* see if left neighbor can now eat */
    test(RIGHT);                    /* see if right neighbor can now eat */
    up(&mutex);                     /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Solution to dining philosophers problem (part 2)

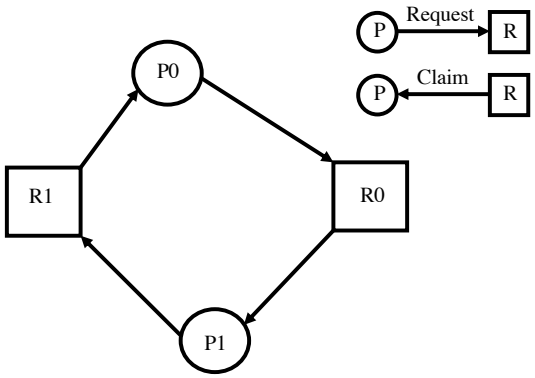
7

# Standard Problem – Dining Philosophers

- Acquire left and right forks/chopsticks to eat pasta/rice
  - Eat, drop utensils, think, repeat
- Solutions
  - Resource hierarchy – number "forks" and acquire in order
  - Arbitrator – acquire both "forks" at once or none at all (talking to a "waiter", or using a mutex)
  - Communicate – indicate intent to acquire with a request or by setting state to "hungry"

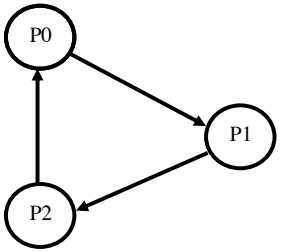
8

## Resource Allocation Graph




9

## Wait-for Graph




10

## Solutions to Deadlock

- The OS can't really do much
  - Application environments like databases can, however
- Some algorithms attempt to stay in a safe state by having processes declare their resource needs, and ensuring that at least one process can acquire all resources needed to complete
  - That there is a possible path to completion
  - Unrealistic to have processes declare their resource needs
- Can periodically test for cycles in the waiting graph
  - Kill all processes?
  - Unblock processes?
- A programmer can use non-blocking messaging interfaces, but care must be taken for synchronization

11

## end

12