Semaphores and Deadlock

CMPU 224 – Computer Organization
Jason Waterman

Semaphore

- Created by Dijkstra to be a single primitive for synchronization
  - Can be used as both **locks** and **condition variables**
- An object with an integer value associated with it
- We can manipulate with two routines
  - **`sem_wait()`**
  - **`sem_post()`**
- Must initialize before use

```
1   #include <semaphore.h>
2   sem_t s;
3   sem_init(&s, 0, 1); // initialize s to the value 1
```

  - Declare a semaphore `s` and initialize it to the value 1
  - The second argument, 0, indicates that the semaphore is shared between threads in the same process

Semaphore operations

- `sem_wait(sem_t *s)`
  - **Decrements** the integer value of the semaphore by 1
  - If the value is **negative** the semaphore will wait
    - It will cause the caller to suspend execution waiting for a subsequent post
    - Similar to a `cond_wait()`
  - If the value of the semaphore (after the decrement) is positive or zero, return immediately
- `sem_post(sem_t *s)`
  - **Increments** the value of the semaphore by 1
  - If there are any threads waiting on the semaphore, **wake** one of them up
- When negative, the value of the semaphore is the number of threads waiting on the semaphore
- **Both operations happens atomically**

Using a Semaphore as a Lock

- Semaphores can be used to provide mutual exclusion

```
1    sem_t m;
2    sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4    sem_wait(&m);
5    //critical section here
6    sem_post(&m);
```

  - What should the semaphore above be initialized to?
    - The semaphore should be initialized to 1
- This is known as a binary semaphore
  - Works the same as a lock

Value of Semaphore	Thread 0
1	
1	call sem_wait()
0	sem_wait() returns
0	(crit sect)
0	call sem_post()
1	sem_post() returns

Thread Trace: Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	`call sem_wait()`	Running		Ready
0	`sem_wait() returns`	Running		Ready
0	`(crit set: begin)`	Running		Ready
0	*`Interrupt; Switch → T1`*	Ready		Running
0		Ready	`call sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem < 0)→sleep`	sleeping
-1		Running	*`Switch → T0`*	sleeping
-1	`(crit sect: end)`	Running		sleeping
-1	`call sem_post()`	Running		sleeping
0	`increment sem`	Running		sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post() returns`	Running		Ready
0	*`Interrupt; Switch → T1`*	Ready		Running
0		Ready	`sem_wait() returns`	Running
0		Ready	`(crit sect)`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`sem_post() returns`	Running

Semaphores as Condition Variables

```
1   sem_t s;
2
3   void* child(void *arg) {
4       printf("child\n");
5       sem_post(&s); // signal here: child is done
6       return NULL;
7   }
8
9   int main(int argc, char *argv[]) {
10      sem_init(&s, 0, X); // what should X be?
11      printf("parent: begin\n");
12      pthread_t c;
13      Pthread_create(c, NULL, child, NULL);
14      sem_wait(&s); // wait here for child
15      printf("parent: end\n");
16      return 0;
17  }
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- What should **X** be?
  - The value of semaphore should be set to is **0**

The Producer/Consumer (Bounded-Buffer) Problem

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

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

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

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```
sem_t empty; // count of empty slots
sem_t full;  // count of full slots
sem_t mutex; // lock

int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);
    // …
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        int tmp = get();
        sem_post(&mutex);
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}
```

Reader-Writer Locks

- Imagine several concurrent list operations, including **inserts** and simple **lookups**
  - **Insert**
    - Change the state of the list
    - A traditional **critical section** makes sense
  - **Lookup**
    - Simply read the data structure
    - If we can guarantee that no insert is on-going, we can allow many lookups to proceed **concurrently**

This special type of lock is known as a reader-writer lock

A Reader-Writer Locks

- Only **a single writer** can acquire the lock
- Once a reader has acquired a read lock
  - **More readers** will be allowed to acquire the read lock too
  - A writer will have to wait until all readers are finished
- What about **fairness**?
  - It would be relatively easy for reader to starve writer
  - A more sophisticated scheme could prevent this

```
1    typedef struct _rwlock_t {
2         sem_t lock;          // binary semaphore (basic lock)
3         sem_t writelock;     // allow ONE writer or MANY readers
4         int readers;         // count of readers reading in critical section
5    } rwlock_t;
6    void rwlock_init(rwlock_t *rw) {
7         rw->readers = 0;
8         sem_init(&rw->lock, 0, 1);
9         sem_init(&rw->writelock, 0, 1);
10   }
11   void rwlock_acquire_readlock(rwlock_t *rw) {
12        sem_wait(&rw->lock);
15        rw->readers++;
16        if (rw->readers == 1)
17            sem_wait(&rw->writelock); // first reader acquires writelock
18        sem_post(&rw->lock);
19   }
20   void rwlock_release_readlock(rwlock_t *rw) {
21        sem_wait(&rw->lock);
22        rw->readers--;
23        if (rw->readers == 0)
24            sem_post(&rw->writelock); // last reader releases writelock
25        sem_post(&rw->lock);
26   }
27   void rwlock_acquire_writelock(rwlock_t *rw) {
28        sem_wait(&rw->writelock);
29   }
30   void rwlock_release_writelock(rwlock_t *rw) {
31        sem_post(&rw->writelock);
32   }
```

Thread throttling

- Used to prevent “too many” threads from doing something all at once
- Limit the number of concurrent threads with a threshold semaphore
  - **Throttling**, a form of **admission control**
- Example:
  - Hundreds of threads solving a parallel problem
  - One area of the code is memory-intensive
  - If all threads are allowed into this area, machine will start swapping and thrashing
- Solution:
  - Add a semaphore initialized to the maximum number of threads allowed in the memory-intensive area
  - Put a `sem_wait()` and `sem_post()` around the memory-intensive area

How To Implement Semaphores

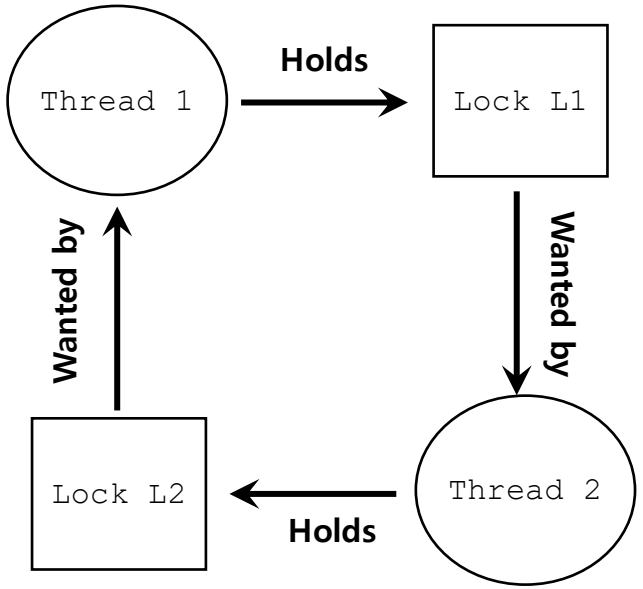
- Build our own version of semaphores called **Zemaphores**
- Doesn't maintain the invariant that a negative value is a count of threads waiting on the semaphore
  - The value is never lower than zero
  - This behavior is easier to implement and matches the current Linux implementation

```
1   typedef struct __Zem_t {
2       int value;
3       pthread_cond_t cond;
4       pthread_mutex_t lock;
5   } Zem_t;
6
7   // only one thread can call this
8   void Zem_init(Zem_t *s, int value) {
9       s->value = value;
10      Cond_init(&s->cond);
11      Mutex_init(&s->lock);
12  }
13  void Zem_wait(Zem_t *s) {
14      Mutex_lock(&s->lock);
15      while (s->value <= 0)
16          Cond_wait(&s->cond, &s->lock);
17      s->value--;
18      Mutex_unlock(&s->lock);
19  }
22  void Zem_post(Zem_t *s) {
23      Mutex_lock(&s->lock);
24      s->value++;
25      Cond_signal(&s->cond);
26      Mutex_unlock(&s->lock);
27  }
```

Deadlock

- The presence of a **cycle** in a resource-allocation graph
  - `Thread1` is holding a lock `L1` and waiting for another one, `L2`
  - `Thread2` that holds lock `L2` is waiting for `L1` to be release

```
Thread 1:          Thread 2:
lock(L1);          lock(L2);
lock(L2);          lock(L1);
```



Why Do Deadlocks Occur?

- Reason 1:
  - In large code bases, complex dependencies arise between components
- Reason 2:
  - Due to the nature of encapsulation
    - Hide details of implementations and make software easier to build in a modular way
    - Such modularity does not mesh well with locking

Why Do Deadlocks Occur? (Cont.)

- **Example**: Java Vector class and the method `AddAll()`

```
1    Vector v1,v2;
2    v1.AddAll(v2);
```

  - **Locks** for both the vector being added to (`v1`) and the parameter (`v2`) need to be acquired
    - The routine acquires locks in some order (e.g., v1 then v2)
    - If some other thread `calls v2.AddAll(v1)` at nearly the same time → We have the potential for deadlock

Conditions for Deadlock

- **Four conditions** need to hold for a deadlock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

- If any of these four conditions are not met, **deadlock cannot occur**

Deadlock Prevention

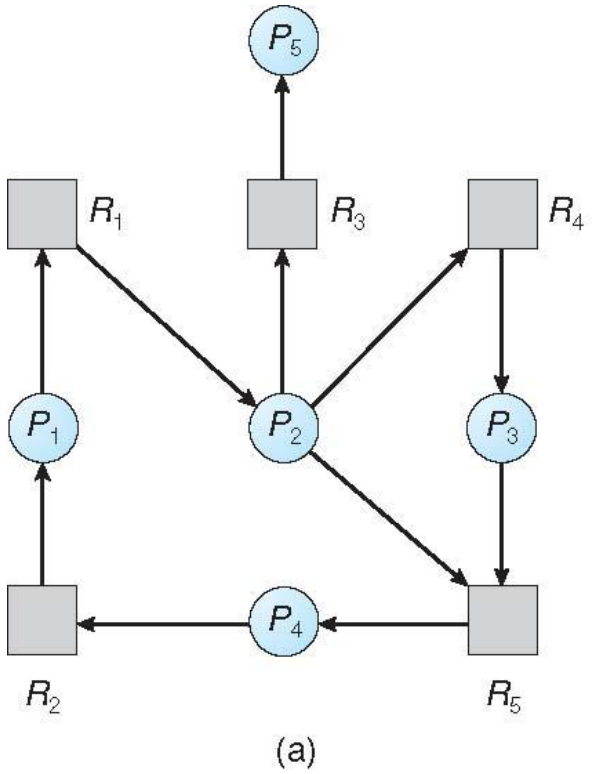
- Restrain the ways requests can be made to make at least one of the four deadlock conditions does not hold
  - **Mutual Exclusion**
    - Not required for sharable resources (e.g., read-only files)
    - Must hold only for non-sharable resources
  - **Hold and Wait**
    - Require process to request and be allocated all its resources before it begins execution
    - Low resource utilization; starvation possible
  - **No Preemption**
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
  - **Circular Wait**
    - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Detection

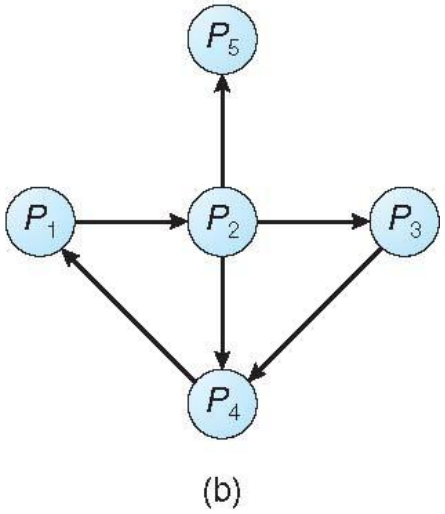
- Allow system to enter deadlock state
- Detect that deadlock has occurred
  - Detection algorithm
- Recover from deadlock
  - Recovery scheme

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph
  - If there is a cycle, deadlock exists
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph



Resource-Allocation Graph



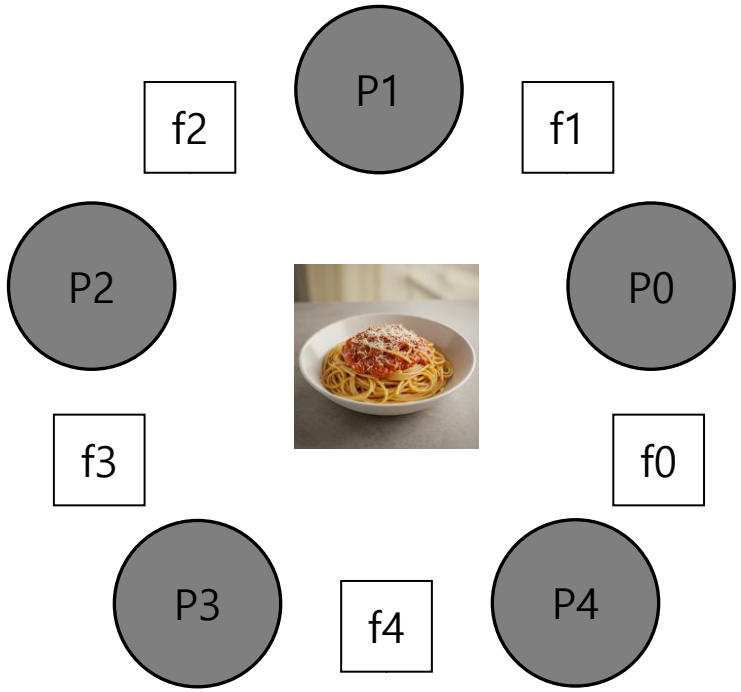
Corresponding wait-for graph

Recovery from Deadlock: Process Termination

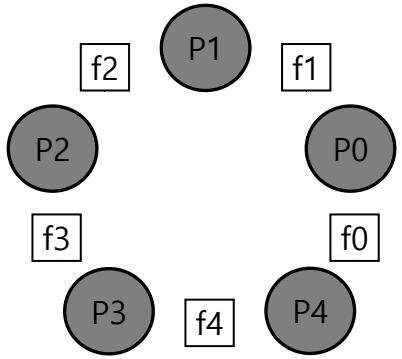
- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

The Dining Philosophers

- Assume there are five “philosophers” sitting around a table
  - Between each pair of philosophers is a single fork (five total)
  - The philosophers each have times when they think, and don’t need any forks, and times when they eat
  - To eat, a philosopher needs two forks, both the one on their left and the one on their right



The Dining Philosophers (Cont.)



- Key challenges
  - There is **no deadlock**
  - **No** philosopher **starves** and never gets to eat
  - **Concurrency** is high

```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

int right(int p) {
        return (p + 1) % 5;
}
```

Helper functions

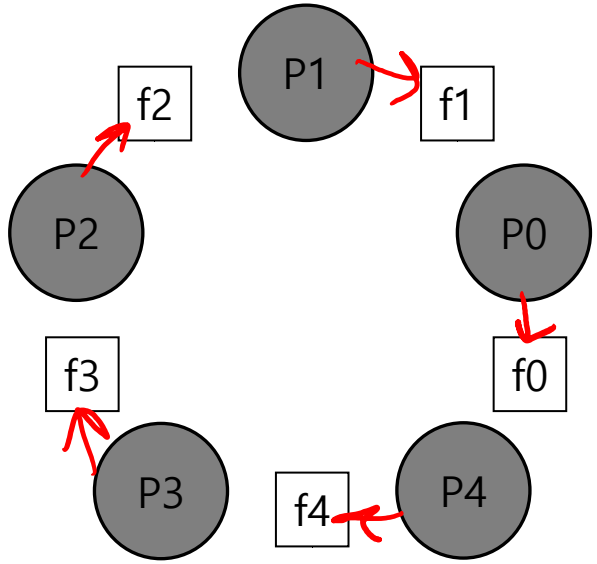
- Philosopher `p` wishes to refer to the fork on their left → call `left(p)`
- Philosopher `p` wishes to refer to the fork on their right → call `right(p)`

The Dining Philosophers (Cont.)

- We need some **semaphores**, one for each fork: `sem_t forks[5]`

```
1    void getforks() {
2        sem_wait(forks[left(p)]);
3        sem_wait(forks[right(p)]);
4    }
5
6    void putforks() {
7        sem_post(forks[left(p)]);
8        sem_post(forks[right(p)]);
9    }
```

The `getforks()` and `putforks()` Routines (Broken Solution)

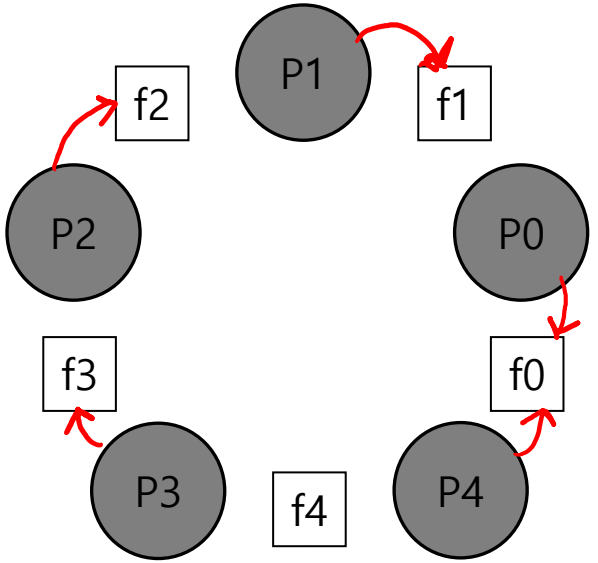


- Deadlock occurs
  - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right
  - Each will be stuck *holding one fork* and waiting for another, *forever*

A Solution: Breaking The Dependency

- Change how forks are acquired
  - Let's assume that philosopher 4 acquires the forks in a *different order*

```
1    void getforks() {
2         if (p == 4) {
3                   sem_wait(forks[right(p)]);
4                   sem_wait(forks[left(p)]);
5         } else {
6                   sem_wait(forks[left(p)]);
7                   sem_wait(forks[right(p)]);
8         }
9    }
```



  - There is no situation where each philosopher grabs one fork and is stuck waiting for another
  - **The cycle of waiting is broken**

Summary

- We need to synchronize for correctness
  - Unsynchronized code can cause incorrect behavior
  - But too much synchronization means threads spend a lot of time waiting, not performing useful work
- Getting synchronization right is hard
  - Testing isn't enough
  - Need to assume worst case: all interleavings are possible
- How to choose between locks, semaphores and condition variables?
  - Locks are very simple and suitable for many cases
    - Issues: Maybe not the most efficient solution
    - E.g., can't allow multiple readers but one writer inside a standard lock
  - Condition variables allow threads to sleep until an event occurs
    - Just remember the state of the world might have changed since the signal was called
  - Semaphores provide general functionality
    - But can be tricky to get correct