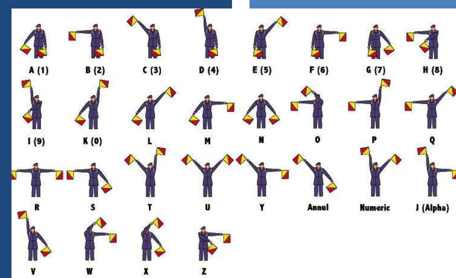
TCSS 422: OPERATING SYSTEMS



Semaphores

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

OBJECTIVES

- Semaphores - API
- Uses
- Reader/Writer Locks
- Dining Philosophers

ANOTHER APPROACH TO CONCURRENCY

- We've looked at **Locks** (ch. 28) and **Conditions** (ch. 30) to provide atomicity in critical sections for concurrency
- Now we'll look at "**semaphores**"
- Provide same functionality
- With different "packaging"

THE SEMAPHORE

- Semaphores (struct in Linux):
- Contains:
- Lock
- Integer: (essentially a counter)
- List: (thread wait list)

SEMAPHORE API

- sem_init():

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

- Initializes new semaphore:

- First param- address of a semaphore
  Second param: 0- single process, 1- multiprocess
  "1" can be used with fork() to synchronize processes
  Third param: initial value of counter

Spring 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	B.5

SEMAPHORE API - 2

- sem_wait():
  - Decrements the value of the semaphore counter, and returns
  - Adds thread to wait queue if counter <= 0 and blocks it

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}
```

  - The negative value corresponds to the number of queued, waiting threads

Spring 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	B.6

SEMAPHORE API - 3

- sem_post():
  - Increments the semaphore counter by 1.
  - Awakens a thread on the wait queue (if any)
    - *(when counter < 0)*

```
int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```


SEMAPHORE AS A LOCK

- What should the value of X be below?
  - Consider two threads entering this code, one immediately after the other
  - What should the first thread do?
  - The second thread do?

```
sem_t m;
sem_init(&m, 0, X);  // initialize semaphore to X

sem_wait(&m);         // similar to lock
                      // critical section goes here
sem_post(&m);         // similar to unlock
```


TWO THREADS AND A SEMAPHORE

- Semaphore as a lock:

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	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() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

SEMAPHORE AS A CONDITION VARIABLE

- Semaphores can be thought of as “mutants”
  - They can be used as locks, or condition variables

- Consider an example

SEMAPHORE AS A CONDITION VARIABLE -2

- What should be the value of X ?

```c
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```


ORDERING OF EXECUTION – 1 OF 2

- Parent calls sem_wait() before child calls sem_post()

Value	Parent	State	Child	State
0	Create(Child)	Running	*(Child exists; is runnable)*	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	*Switch→Child*	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	sem_wait() retruns	Running		Ready

ORDERING OF EXECUTION – 2 OF 2

- **Child runs, calls sem_post() before parent calls sem_wait()**

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

PRODUCER/CONSUMER W/ SEMAPHORES

- **Producer: put()**
- **Consumer: get()**
- **With MAX=1, 1 consumer thread, 1 producer thread:**

```
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;
}
```

PRODUCER/CONSUMER W/ SEMAPHORES - 2

```c
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 i, tmp = 0;
    while (tmp != -1) {
            sem_wait(&full);          // line C1
            tmp = get();              // line C2
            sem_post(&empty);         // line C3
            printf("%d\n", tmp);
    }
}
…
```

Spring 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	B.15

PRODUCER/CONSUMER W/ SEMAPHORES - 3

```c
int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX);      // MAX buffers are empty to begin with…
    sem_init(&full, 0, 0);         // … and 0 are full
    // …
}
```

- This code is sufficient for any size buffer with 1 producer, 1 consumer
- Try it out
- But what happens if we add multiple producers and consumers?
- Try it out
- Must consider critical sections

Spring 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	B.16

MULTI THREAD P/C SEMAPHORES W/ MUTUAL EXCLUSION

- Which part of the code is the critical section?

```
sem_t empty;
sem_t full;
sem_t mutex;

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)
     }
}

(Cont.)
```


MULTI THREAD P/C SEMAPHORES W/ MUTUAL EXCLUSION - 2

```
(Cont.)
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);
     }
}
```

EXECUTION FLOW

- With one producer, one consumer
  - Consumer acquires mutex (the lock)
  - Consumer calls sem_wait() to wait for data
  - No data available, consumer blocks are yields the CPU
    - Still has mutex (the lock)
  - Producer tries to acquire mutex (the lock)
  - Producer becomes stuck in deadlock
    - Consumer is waiting for data, and will never release the mutex

MULTITHREAD P/C W/ SEMAPHORES

- Lock should only protect put(), get()

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
     int i;
     for (i = 0; i < loops; i++) {
              sem_wait(&empty);          // line p1
              sem_wait(&mutex);          // line p1.5 (MOVED MUTEX HERE…)
              put(i);                    // line p2
              sem_post(&mutex);          // line p2.5 (… AND HERE)
              sem_post(&full);           // line p3
     }
 }

(Cont.)
```

MULTITHREAD P/C W/ SEMAPHORES - 2

- Try it out…

```
(Cont.)
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&full);          // line c1
            sem_wait(&mutex);         // line c1.5 (MOVED MUTEX HERE…)
            int tmp = get();          // line c2
            sem_post(&mutex);         // line c2.5 (… AND HERE)
            sem_post(&empty);         // line c3
            printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with …
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // …
}
```


CONCURRENT DATA STRUCTURES

- Concurrent data structures ideally will:
  - Ensure atomicity of writes
  - Enable multiple synchronous reads
    - As long as elements being read are not concurrently changed
- Concurrent linked list, use a **Reader-Writer Lock**
  - Insert
    - Has traditional critical section which **<u>must not</u>** be multiply entered
  - Read
    - Should support concurrent reads if not being changed
    - Semaphores: *good for tracking concurrent reads*

CONCURRENT LIST WITH SEMAPHORES

- Multiple readers can acquire a lock
  - Writer must wait until all readers finish

```
typedef struct _rwlock_t {
     sem_t lock;       // binary semaphore (basic lock)
     sem_t writelock;  // used to allow ONE writer or MANY readers
     int readers;      // count of readers reading 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);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
     sem_wait(&rw->lock);
     …
```


CONCURRENT LIST WITH SEMAPHORES - 2

```
     rw->readers++;
     if (rw->readers == 1)
             sem_wait(&rw->writelock); // first reader acquires writelock
     sem_post(&rw->lock);
}

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

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

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

READER-WRITER LOCK

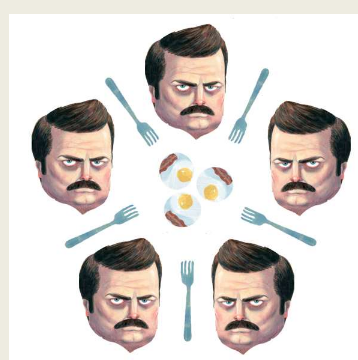
- Fairness problem
  - With many readers, it becomes difficult for a writer to obtain the lock
  - One improvement is to prevent more readers from reading once a writer is waiting for the lock
  - **How could we implement this improvement?**

Spring 2021 | TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma | B.25

DINING PHILOSOPHERS PROBLEM

- Classic computer science problem
- Possible job interview question

- Philosopher's
1. Think
2. Pick up forks (wait if not available)
3. Eat
4. Put down forks

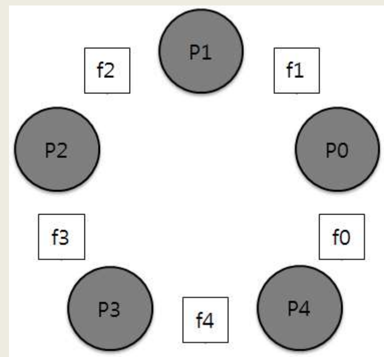


Spring 2021 | TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma | B.26

DINING PHILOSOPHERS - 2

- P- Philosopher
- f- fork (eating utensil)

- Key challenges
  - There is no deadlock
  - No philosopher starves
  - Concurrency is high
    - Forks get used as much as possible
    - Philosophers have plenty of eating opportunities



DINING PHILOSOPHERS - 3

- Philosophers:

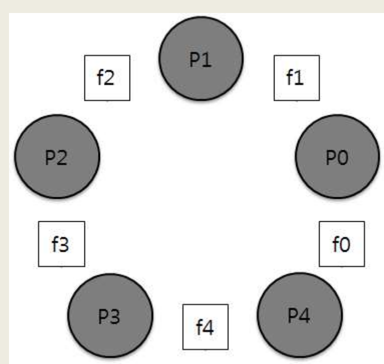
```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

- Fork helper functions

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

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

- Fork on left: left(P1) = f1
- Fork on right: right(P1) = f2



DINING PHILOSOPHERS - 4

- If we just protect the forks with semaphores:

```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

void putforks() {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

- Try this:

DINING PHILOSOPHERS - 5

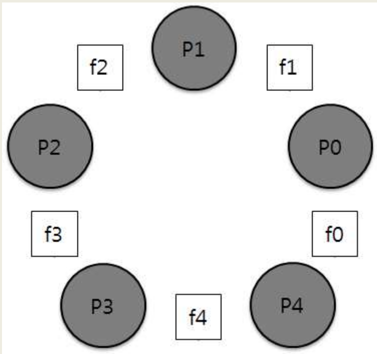
```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

void putforks() {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

- Complete the table

Philosopher	LEFT	RIGHT
P0		
P1		
P2		
P3		
P4		



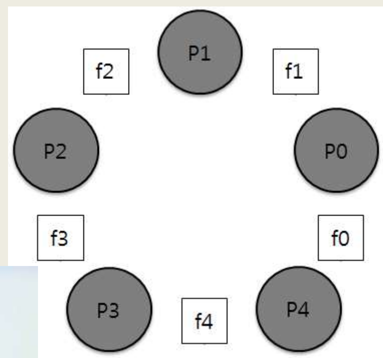
DINING PHILOSOPHERS - 5

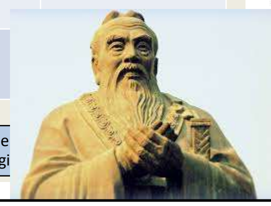
```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

void putforks() {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

- **DEADLOCK: All Philosophers Starve!**

Philosopher	LEFT	RIGHT
P0	acquires f0	waits for f1
P1	acquires f1	waits for f2
P2	acquires f2	waits for f3
P3	acquires f3	waits for f4
P4	acquires f4	





ALTERNATE PHILOSOPHER

- We need another approach to acquiring forks
- Consider which fork philosophers grab first
- What if we have a ***alternate-handed philosopher?***

```
void getforks() {
    if (p == 4) {
        sem_wait(forks[right(p)]);
        sem_wait(forks[left(p)]);
    } else {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
    }
}
```

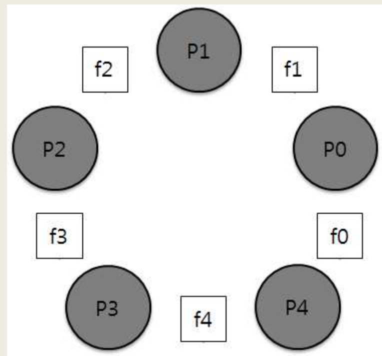
- **Solves the Dining Philosopher's problem !!!**
- Remember that one philosopher grabs a different fork

ALTERNATE PHILOSOPHER - 2

- P3 eats! Solves deadlock

```
void getforks() {
     if (p == 4) {
          sem_wait(forks[right(p)]);
          sem_wait(forks[left(p)]);
     } else {
          sem_wait(forks[left(p)]);
          sem_wait(forks[right(p)]);
     }
}
```

Philosopher	LEFT	RIGHT
P0	acquires f0	waits for f1
P1	acquires f1	waits for f2
P2	acquires f2	waits for f3
P3	acquires f3	acquires f4, eats…
P4		Waits for f0



SEMAPHORE IMPLEMENTATION

- Semaphores can be built using locks and conditions
  - pthread_mutex_t
  - pthread_cond_t
- Linux implementation
  - Does not track negative counter values
  - Easier to implement
- Zemaphore

SEMAPHORE IMPLEMENTATION - 2

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
    Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}
```

```
void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```


SEMAPHORES SUMMARY

- Provide one construct for both concurrency features
  - Binary semaphore: provides basic mutex lock
    - Ensures mutual exclusion in critical sections
  - Condition semaphore: Synchronize one or more threads which need to wait for something to happen
  - Allows fewer concurrency related variables in your code
  - Potentially makes code more ambiguous

- After seeing Locks, Conditions, and Semaphores, Which do you like better?

QUESTIONS