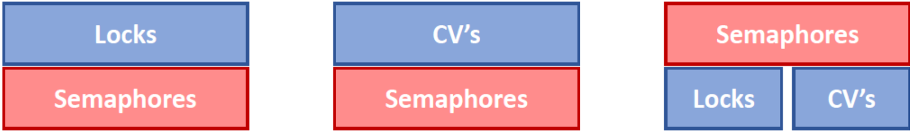
IE411: Operating Systems

Semaphores

Synchronization constructs

- Previously we looked at locks and condition variables
- We can combine these two concepts to create a new type of synchronization primitive called semaphore

Equivalence



Semaphore data structure

```
typedef struct {
  int count;
  queue_t waiting;
} semaphore_t;
```

- an integer value
- a queue of threads waiting on the semaphore

Semaphore operations - sem_wait()

- sem_wait() decrements the semaphore's value
  - also called P() (Dijkstra - Dutch for prolaag)
- if the resulting value is negative, the calling thread is blocked
  - a blocked thread can be woken up when the semaphore value is incremented

```
void sem_wait(semaphore_t *s)
{
    s->count--;
    if (s->count < 0) {
        add calling thread to s->queue;
        put calling thread to sleep;
    }
}
```

Semaphore operations - sem_post()

- sem_post() increments the semaphore's value
  - also called V() (Dijkstra - Dutch for verhoog)
- incrementing the value when it is negative causes one of the waiter threads to become runnable

```
void sem_post(semaphore_t *s)
{
    s->count++;
    if (s->count <= 0) {
        remove one thread from s->queue;
        wake up this thread;
    }
}
```

Semaphore nuance

- `sem_wait()` and `sem_post()` are made atomic
- how negative the semaphore's value is represents the number of threads waiting

Application - Mutual Exclusion

- like a lock
- integer value represents a 1/0 value to lock and unlock a critical section

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

Application - Ordering

- Like a condition variable
- If we choose the correct starting value, we can ensure some simple ordering scenarios
- $X = ?$

```
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;
}
```

Figure 31.6: **A Parent Waiting For Its Child**

Application - Ordering

```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;
}
```

Figure 31.6: **A Parent Waiting For Its Child**

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)`→`sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Application - Ordering

```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;
}
```

Figure 31.6: **A Parent Waiting For Its Child**

Val	Parent	State	Child	State
0	create(Child)	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	child runs	Run
0		Ready	call sem_post()	Run
1		Ready	inc sem	Run
1		Ready	wake(nobody)	Run
1		Ready	sem_post() returns	Run
1	parent runs	Run	*Interrupt→Parent*	Ready
1	call sem_wait()	Run		Ready
0	decrement sem	Run		Ready
0	(sem≥0)→awake	Run		Ready
0	sem_wait() returns	Run		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

Producer-Consumer Problem

- The bounded-buffer producer-consumer problem assumes there is a buffer of size N
- The producer puts items to the buffer area
- The consumer consumes items from the buffer
- The producer and consumer execute concurrently

Producer-Consumer Model

- Shared data

```
sem_t full, empty;
```

- Initially

```
full = 0;        /* The number of full buffers */
empty = MAX;     /* The number of empty buffers */
```

First Attempt: MAX = 1

```
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);
    }
}

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
    // ...
}
```

```
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;
}
```

Put and Get routines

First Attempt: MAX = 10?

```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);
    }
}

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
    // ...
}
```

```c
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;
}
```

Put and Get routines

First Attempt: MAX = 10?

fill = 0

empty = 10

Producer 0: Running

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

Producer 1: Runnable

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

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: Running

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

Producer 1: Runnable

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

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

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: Running

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

Producer 1: Runnable

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

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

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: Sleeping

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

Producer 1: Runnable

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

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

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: Runnable

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

Producer 1: Running

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

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

First Attempt: MAX = 10?

fill = 0

Overwrite!

empty = 8

Producer 0: Runnable

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

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

Producer 1: Running

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

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

Producer-Consumer Model

- Shared data

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

- Initially

```
full  = 0;      /* The number of full buffers */
empty = MAX;    /* The number of empty buffers */
mutex = 1;      /* Semaphore controlling the access
                   to the buffer pool */
```

Add Mutual Exclusion

```
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)
    }
}

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);
    }
}

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 (NEW LINE)
    // ...
}
```

Add Mutual Exclusion

```c
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)
    }
}

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);
    }
}

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 (NEW LINE)
    // ...
}
```

What if consumer gets to run first??

Add Mutual Exclusion

mutex = 1

full = 0

empty = 10

Producer 0: Runnable

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

Consumer 0: Running

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

Add Mutual Exclusion

mutex = 0

full = 0

empty = 10

Producer 0: Runnable

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

Consumer 0: Running

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

Consumer 0 is waiting for full to be greater than or equal to 0

Add Mutual Exclusion

mutex = -1
full = -1
empty = 10

Producer 0: Running

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

Consumer 0: Runnable

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

Consumer 0 is **waiting** for full to be greater than or equal to 0

Add Mutual Exclusion

Deadlock!!

mutex = -1

full = -1

empty = 10

Producer 0: Running

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

Producer 0 **gets stuck** at acquiring `mutex` which has been locked by Consumer 0!

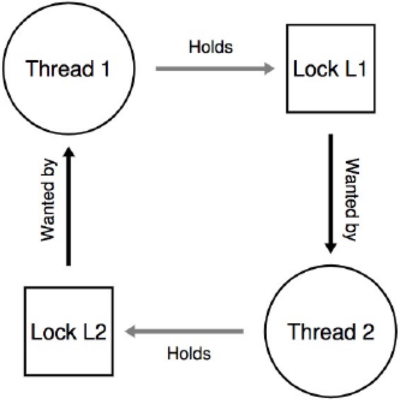
Consumer 0: Runnable

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

Consumer 0 is **waiting** for full to be greater than or equal to 0

Deadlocks

When every thread in a set of threads is waiting for an event that can be caused only by another thread in the set



A typical deadlock dependency graph

Correct Mutual Exclusion

```
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
    }
}

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
    // ...
}
```

Mutex wraps just around critical section!

Mutex wraps just around critical section!

Reader/Writers

- Single shared object
- Want to allow any number of threads to read simultaneously
- But, only one thread should be able to write to the object at a time
  - And, not interfere with any readers ...

Readers/Writers

- readers share:
  - semaphore wrt; // initialized to 1
  - int readcount; // initialized to 0
- writers also share semaphore wrt

Writer:

```
while (1) {
   P(wrt);
    // writing
   V(wrt);
}
```

Reader:

```
while (1) {
   readcount++;
   if (readcount == 1) {
       P(wrt);
   }
   // reading
   readcount--;
   if (readcount == 0) {
       V(wrt);
   }
}
```

- Seems simple, but this code is broken . . .

Readers/Writers

Writer:

```
while (1) {
   P(wrt);
    // writing
   V(wrt);
}
```

What can happen if we context switch here?

Reader:

```
while (1) {
   readcount++;
   if (readcount == 1) {
       P(wrt);
   }
   // reading
   readcount--;
   if (readcount == 0) {
       V(wrt);
   }
}
```

Readers/Writers

Another Reader() could start and "readcount==1" never happens!

Writer:

```
while (1) {
   P(wrt);
    // writing
   V(wrt);
}
```

Reader:

```
while (1) {
   readcount++;
   if (readcount == 1) {
       P(wrt);
   }
   // reading
   readcount--;
   if (readcount == 0) {
       V(wrt);
   }
}
```

Readers/Writers

What can happen if we context switch here?

Writer:

```
while (1) {
   P(wrt);
    // writing
   V(wrt);
}
```

Reader:

```
while (1) {
   readcount++;
   if (readcount == 1) {
       P(wrt);
   }
   // reading
   readcount--;
   if (readcount == 0) {
       V(wrt);
   }
}
```

Readers/Writers

A Writer() could start, P the semaphore first, then subsequent Reader() threads would be able to get past the semaphore (since "readcount != 1")

Writer:

```
while (1) {
   P(wrt);
    // writing
   V(wrt);
}
```

Reader:

```
while (1) {
   readcount++;
   if (readcount == 1) {
      P(wrt);
   }
   // reading
   readcount--;
   if (readcount == 0) {
      V(wrt);
   }
}
```

Readers/Writers fixed

- Problem: Multiple Readers are accessing `readcount`
- Solution: Make "increment, test, P" and "decrement, test, V" both atomic - using a mutex

Writer:

```
while (1) {
    P(wrt);
    // writing
    V(wrt);
}
```

Reader:

```
while (1) {
    P(mutex);
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    V(mutex);
    // reading
    P(mutex);
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
    V(mutex);
}
```

Readers/Writers fixed

Writer:

```
while (1) {
    P(wrt);
    // writing
    V(wrt);
}
```

Reader:

```
while (1) {
    P(mutex);
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    V(mutex);
    // reading
    P(mutex);
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
    V(mutex);
}
```

What if a Writer() is active, the first Reader() stalls on P(wrt), and additional Readers() try to enter?

Readers/Writers fixed

Writer:

```
while (1) {
   P(wrt);
   // writing
   V(wrt);
}
```

Reader:

```
while (1) {
   P(mutex);
   readcount++;
   if (readcount == 1) {
       P(wrt);
   }
   V(mutex);
   // reading
   P(mutex);
   readcount--;
   if (readcount == 0) {
       V(wrt);
   }
   V(mutex);
}
```

The subsequent Reader() threads stall on P(mutex) !