CS330 - Operating Systems

Locks and CVs

15-09-2025

Implementing Synchronization

- Interrupt-controls
- Peterson's solution
  - Classic solution (works on early architectures) for two threads
  - Uses `load` and `store` (which were atomic in early arch.)

```
boolean flag[2];
int turn;

void init() {
     flag[0] = flag[1] = false;


}
```

Can lead to starvation on one or both threads

```
void lock() {
     flag[i] = true;

     while (flag[j])
     ;// wait until it's i's turn
}

void unlock() {
   flag[i] = false;
}
```

Implementing Synchronization

- Interrupt-controls
- Peterson's solution
  - Classic solution (works on early architectures) for two threads
  - Uses `load` and `store` (which were atomic in early arch.)

```
boolean flag[2];
int turn;

void init() {
     flag[0] = flag[1] = false;
     // whose turn is it? (thread 0 or 1)
     turn = 0;
}
```

```
void lock() {
     flag[i] = true;
     turn = j; // make it other thread's turn
     while (flag[j] && (turn == j));
     // wait until it's i's turn
}

void unlock() {
   flag[i] = false;
}
```

Implementing Synchronization

- Interrupt-controls
- Peterson's solution
- Test and Set

```
int TestAndSet(int *old_ptr, int new)
    // fetch old value at old_ptr
     int old = *old_ptr;
    // store 'new' into old_ptr
    *old_ptr = new;
    // return the old value
    return old;
}
```

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Implementing Synchronization

- Interrupt-controls
- Peterson's solution
- Test and Set
- Compare and Swap

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

Implementing Synchronization

- Cons of spin-wait locks
  - Performance
    - Interrupt thread inside critical section
  - Priority Inversion

T1	**T2**
Ready	Running
Ready	Blocked
Running (Lock)	Ready
Ready (Lock)	Running (Wait)
	(Waits forever)

Too Much Spinning

- Context switch in critical section
  - Performance issues
  - Hung execution (Not fair)

What Really Happened on Mars?" by Glenn E. Reeves.
research.microsoft.com/en-us/um/people/mbj/Mars Pathfinder/Authoritative Account.html

Too Much Spinning

- Yield
  - Assume `yield` function
  - Instead of spinning, yield the CPU

```
while (TestAndSet(&flag, 1) == 1) { yield(); }
```

  - What happens with multiple threads?
- Keep track of threads that are waiting in a queue

Example: Bounded buffer

- Producer(s) put things into a shared buffer
- Consumer(s) take them out
- Need synchronization to coordinate producers/consumers

- Multi-threaded web server
- Pipes

```
grep foo file.txt | wc -l
```

Example: Bounded buffer

- Producer(s) put things into a shared buffer
- Consumer(s) take them out
- Need synchronization to coordinate producers/consumers

```
int buffer;
int count = 0; // initially, empty
```

```
void put (int value) {
        count = 1;
        buffer = value;
}
```

```
int get() {
    count = 0;
    return buffer;
}
```

Example: Bounded buffer

- Producer(s) put things into a shared buffer
- Consumer(s) take them out
- Need synchronization to coordinate producers/consumers

```
int buffer;
int count = 0; // initially, empty
```

```
void put (int value) {
        count = 1;
        buffer = value;
}
```

```
int get() {
    count = 0;
    return buffer;
}
```

```
void *producer(int arg) {
    int i;
    int loops = arg;
    for (i = 0; i < loops; i++) {
        put(i);
}}
```

```
void *consumer() {
   while (1) {
      int tmp = get();
      printf("%d\n", tmp);
   }
}
```

Synchronization between Parent and Child

```
1   void *child(void *arg) {
2       printf("child\n");
3       // XXX how to indicate we are done?
4       return NULL;
5   }
6
7   int main(int argc, char *argv[]) {
8       printf("parent: begin\n");
9       pthread_t c;
10      Pthread_create(&c, NULL, child, NULL); // child
11      // XXX how to implement the wait for child?
12      printf("parent: end\n");
13      return 0;
14  }
```

```
1   void done() {
2
3       cond_signal(&c);
4
5   }
6
7   void join() {
8
9       cond_wait(&c);
10
11  }
```

Synchronization between Parent and Child

```
1   void *child(void *arg) {
2       printf("child\n");
3       // XXX how to indicate we are done?
4       return NULL;
5   }
6
7   int main(int argc, char *argv[]) {
8       printf("parent: begin\n");
9       pthread_t c;
10      Pthread_create(&c, NULL, child, NULL); // child
11      // XXX how to implement the wait for child?
12      printf("parent: end\n");
13      return 0;
14  }
```

```
1   void done() {
2       lock(&m);
3       cond_signal(&c);
4       unlock(&m);
5   }
6
7   void join() {
8       lock(&m);
9       cond_wait(&c);
10      unlock(&m);
11  }
```

Synchronization between Parent and Child

```
1   void *child(void *arg) {
2           printf("child\n");
3           // XXX how to indicate we are done?
4           return NULL;
5   }
6
7   int main(int argc, char *argv[]) {
8           printf("parent: begin\n");
9           pthread_t c;
10          Pthread_create(&c, NULL, child, NULL); // child
11          // XXX how to implement the wait for child?
12          printf("parent: end\n");
13          return 0;
14  }
```

```
1   void done() {
2           lock(&m);
3           done = 1;
4           cond_signal(&c);
5           unlock(&m);
6   }
7
8   void join() {
9           lock(&m);
10          if (done == 0)
11                  cond_wait(&c, &m);
12          unlock(&m);
13  }
```

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
- Wait: atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Producer-consumer with CV

- Single producer and consumer

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        if (count == 1)
            Pthread_cond_wait(&cond, &mutex);
        put(i);
        Pthread_cond_signal(&cond);
        Pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        if (count == 0)
            Pthread_cond_wait(&cond, &mutex);
        int tmp = get();
        Pthread_cond_signal(&cond);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Multiple Producers and Consumers

- Mesa Semantics
  - Assume we have two consumers Tc1 and Tc2 and producer Tp
  - Consumer Tc1 runs and *waits* because buffer is empty
  - Producer runs and fills the buffer
    - Signals the buffer is filled
    - Moves Tc1 from waiting/sleeping state to ready state
  - Tp sleeps as buffer is full
  - Tc2 starts running and eats up the buffer; goes to sleep
  - Tc1 is already in the ready queue
    - Runs next and finds that the buffer is empty!!!

Producer-consumer with CV

- Modified code

  (All code is available at: https://github.com/remzi-arpacidusseau/ostep-code/)

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&cond, &mutex);
        put(i);
        Pthread_cond_signal(&cond);
        Pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&cond, &mutex);
        int tmp = get();
        Pthread_cond_signal(&cond);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Producer-consumer Problem

- Use two condition variables

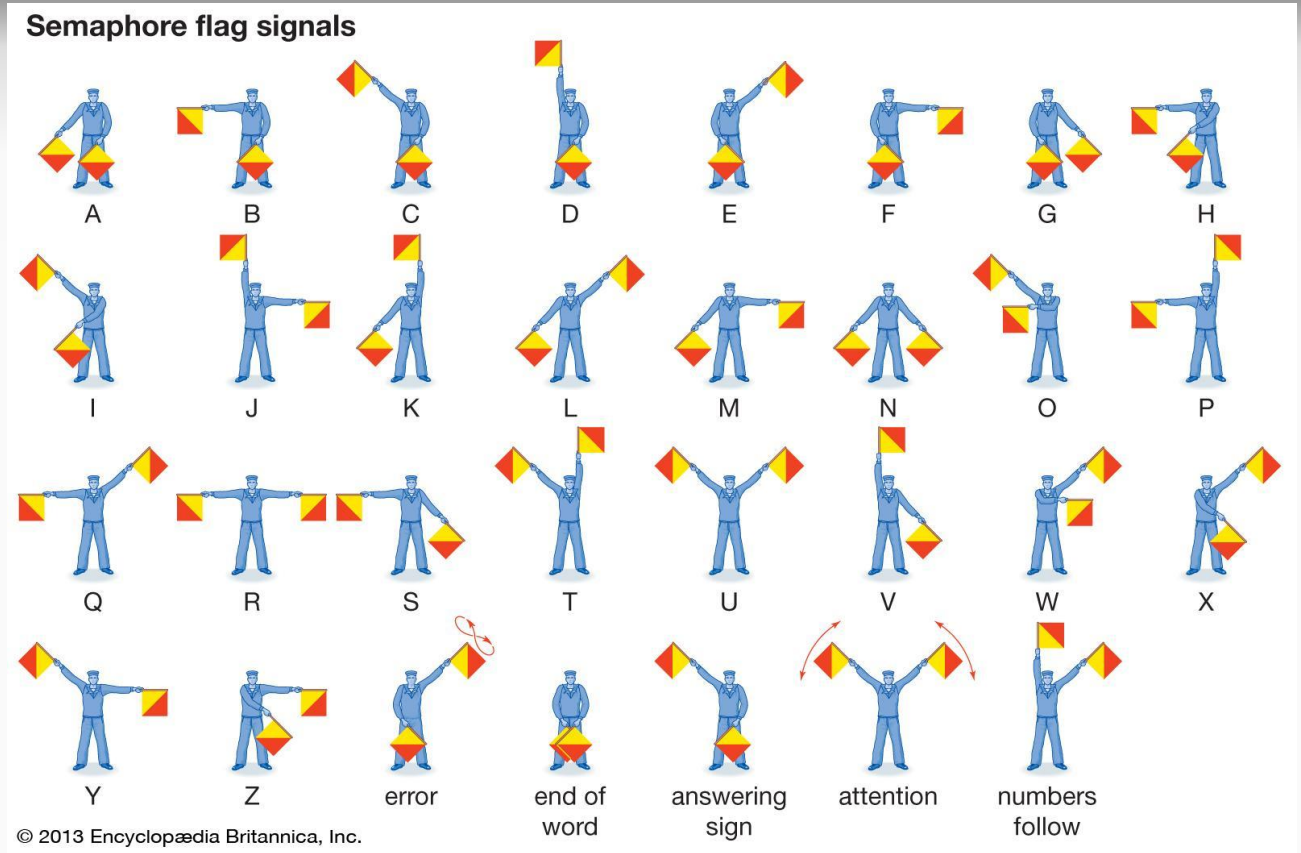
```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Condition Variables

- Hold lock when calling wait() and signal()
  - For shared state; always hold lock when accessing shared state
- Condition variables are memoryless
  - If signal happens when no one is waiting, NO-OP
  - If not wait before signal wakes up the waiter
- Wait atomically releases lock
- Woken thread may not run immediately
  - signal puts thread on ready list; anyone can acquire it after release
- Wait must happen in “loop”

Semaphores


Semaphore flag signals
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
error
end of word
answering sign
attention
numbers follow
© 2013 Encyclopædia Britannica, Inc.

Semaphores

- Single primitive for all synchronization (Dijkstra)
- Positive Integer
  - Initialize its value to any positive integer, but after that the only operations you are allowed to perform are increment and decrement
  - Cannot read the current value
  - When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore
  - When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked

Semaphores

- Signal (V)
  - increment_and_wake_a_waiting_process_if_any
  - sema_up
- Wait (P)
  - decrement_and_block_if_the_result_is_negative
  - sema_down

Semaphores

- Signal (V)
  - increment_and_wake_a_waiting_process_if_any
- Wait (P)
  - decrement_and_block_if_the_result_is_negative

- Impose deliberate constraints to avoid errors.
- Often clean and organized
- Implemented efficiently on many systems

Rendezvous

Given two threads `T1, T2`:

```
T1 : a1  a2
T2 : b1  b2
```

Guarantee that `a1` happens before `b2` and `b1` before `a2`

Semaphore Usage

- Binary semaphore (or locks (or mutexes))
  - Enforce mutual exclusion

    Thread

```
mutex.wait()
// critical section
count = count + 1
mutex.signal()
```

- Multiplex
  - Generalize the previous solution so that it allows multiple threads to run in the critical section at the same time but no more than `n` threads can run in the critical section at the same time

Semaphores Usage

- Counting Semaphores
  - Similar in use to condition variables
- Finite buffer producer-consumer problem

```
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(buffer.size())
```

```
items.wait()
mutex.wait()
      buffer.get()
mutex.signal()
spaces.signal()
```

```
spaces.wait()
mutex.wait()
     buffer.put(i)
mutex.signal()
items.signal()
```

Producer-consumer Problem with Semaphores

- Finite buffer producer-consumer problem

```
int buffer[SIZE]
items = Semaphore(0)
spaces = Semaphore(buffer.size())
```

```
items.wait()
      buffer.get()
spaces.signal()
```

```
spaces.wait()
     buffer.put(i)
items.signal()
```

Producer-consumer Problem with Semaphores

```
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;
}
```

```
items = Semaphore(0)
spaces = Semaphore(buffer.size())
```

```
items.wait()
      buffer.get()
spaces.signal()
```

```
spaces.wait()
     buffer.put(i)
items.signal()
```

Producer-consumer Problem with Semaphores

```
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(buffer.size())
```

```
mutex.wait()
items.wait()
      buffer.get()
spaces.signal()
mutex.signal()
```

Consumer

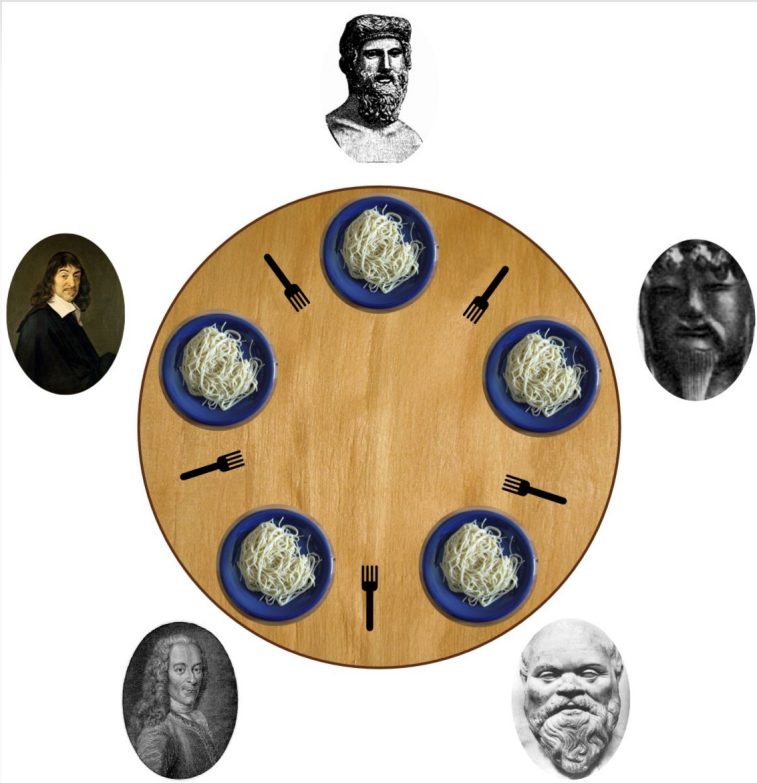
```
mutex.wait()
spaces.wait()
     buffer.put(i)
items.signal()
mutex.signal()
```

Producer

Dining Philosophers Problem

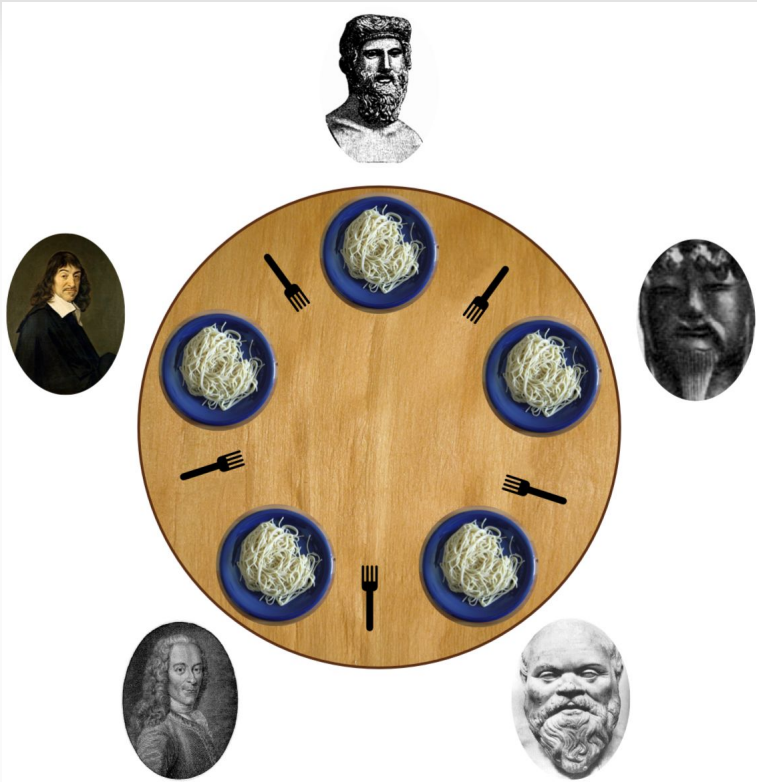
- Five philosophers
- Each pair shares a fork
- Each need two to eat
- Think when do not eat

```
while (true) {
    think ();
    get_forks (p);
    eat ();
    put_forks (p);
}
```



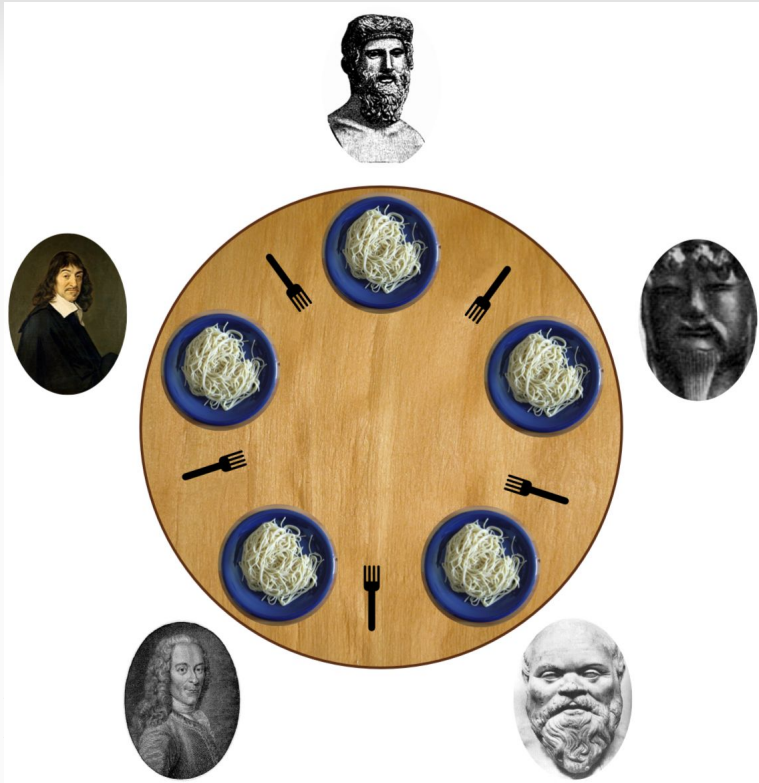
Dining Philosophers Problem

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

   }
void get_forks(int p) {

   }
```



Dining Philosophers Problem

```
int left(int p) {
   return p; }
int right(int p) {
   return (p + 1) % 5; }
void put_forks(int p) {
   sema_up(&forks[left(p)]);
   sema_up(&forks[right(p)]);}
void get_forks(int p) {
   sema_down(&forks[left(p)]);
   sema_down(&forks[right(p)]);
```



Deadlocks

Deadlocks

Thread 1:

```
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Thread 2:

```
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

- Conditions for deadlock
  - Mutual exclusion
    - exclusive access of locks
  - Hold-and-wait
    - hold locks when waiting
  - No preemption
    - locks cannot be removed from threads
  - Circular wait
    - circular chain of threads holding each others' locks

Real-world Concurrency Threats - Example

- Race in the Therac-25 radiation therapy machine caused massive overdose
  - resulted in patient deaths and serious injuries
  - Data Entry and Keyboard Handler routines share a variable, which recorded whether the technician had completed entering commands.
  - A race condition bug of this shared variable cause the UI to display the wrong mode to operators