CS 537: Intro to Operating Systems (Fall 2017)

Worksheet 7 - Thread and Concurrent Data Structures

This worksheet is only for practice and will NOT be graded.

1. Threads vs Processes!

Part I: Assume that the code snippet below compiles successfully, all the APIs like **pthread_create()** do not fail, and the values in the **malloc**'ed memory are all **initialized to 0**.

```
void worker(int *balance) {
    int *counter = malloc(sizeof(int));
    for (int i = 0; i < 1000000; i++) {
        ( *balance )++;
        ( *counter )++;
    }
    printf("balance : val %d, addr %p\n", *balance, balance);
    printf("counter : val %d, addr %p\n", *counter, counter);
}

int main() {
    int *balance;
    balance = malloc(sizeof(int));
    pthread_t t[2];
    for (int i = 0; i < 2; i++) // Creating new threads
        pthread_create(&t[i], NULL, worker, balance);
    for (int i = 0; i < 2; i++)
        pthread_join(t[i], NULL);
}
```

a. What are the **values** of the 2 variables (balance and counter) after the 2 **threads** (t1 and t2) finish execution? Value here means the contents printed using the following print statements. If the value of a variable may be different in different runs of the program, you should write N/A.

`printf("%d\n", *balance);`   `printf("%d\n", *counter);`

Value	t1	t2
balance	N/A	N/A
counter	1M	1M

b. Consider the **Virtual Addresses (VA)** printed using the following print statements in the 2 **threads** (t1 and t2). **PA** stands for **Physical Address.**

```
printf("%p\n", balance);    printf("%p\n", counter);
```

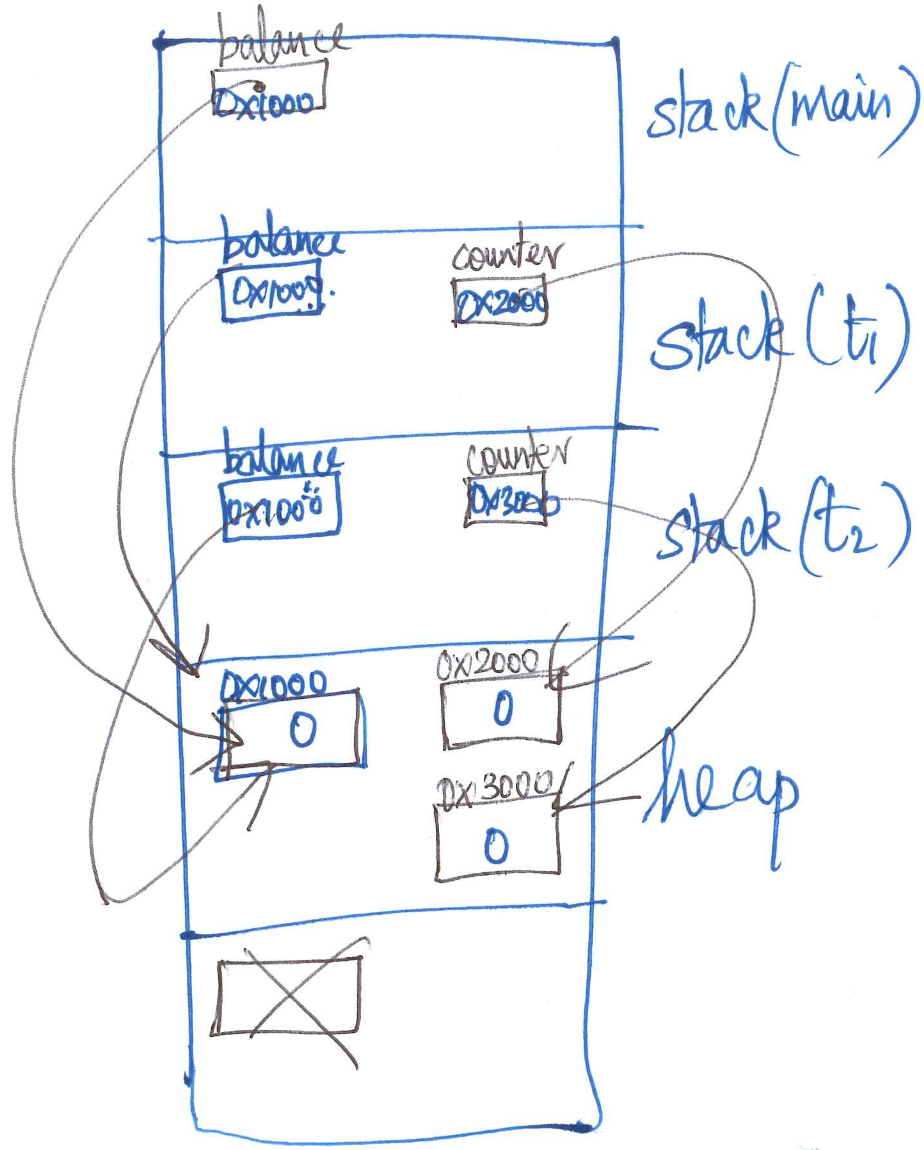
i. VA of balance in t1 == VA of balance in t2? (TRUE / FALSE)
ii. VA of counter in t1 == VA of counter in t2? (TRUE / FALSE)
iii. PA of balance in t1 == PA of balance in t2? (TRUE / FALSE)
iv. PA of counter in t1 == PA of counter in t2? (TRUE / FALSE)

Part II: Now assume that the code given below compiles successfully, all the APIs like **fork**() do not fail, and the values in the **malloc**'ed memory are all **initialized to 0**.

```
void worker(int *balance) {
    int *counter = malloc(sizeof(int));
    for (int i = 0; i < 1000000; i++) {
        ( *balance )++;
        ( *counter )++;
    }
    printf("balance : val %d, addr %p\n", *balance, balance);
    printf("counter : val %d, addr %p\n", *counter, counter);
}

int main() {
    int *balance;
    balance = malloc(sizeof(int));
    for (int i = 0; i < 2; i++) { // Creating new processes
        if (fork() == 0) {
            worker(balance);
            exit(0);
        }
    }
    for (int i = 0; i < 2; i++)
        wait(NULL);
}
```

c. What are the **values** of the 2 variables (`balance` and `counter`) after the 2 **processes** (p1 and p2) created using **fork**() finish execution? Value here means the contents printed using the following print statements. If the value of a variable may be different in different runs of the program, you should write N/A.

```
printf("%d\n", *balance);    printf("%d\n", *counter);
```


balance
0x1000
stack(main)
balance
0x1000
counter
0x2000
stack(t1)
balance
0x1000
counter
0x3000
stack(t2)
0x1000
0
0x2000
0
0x3000
0
heap

Value	p1	p2
balance	1m	1m
counter	1m	1m

d. Consider the **Virtual Addresses (VA)** printed using the following print statements in the 2 **processes** (p1 and p2). **PA** stands for **Physical Address**.

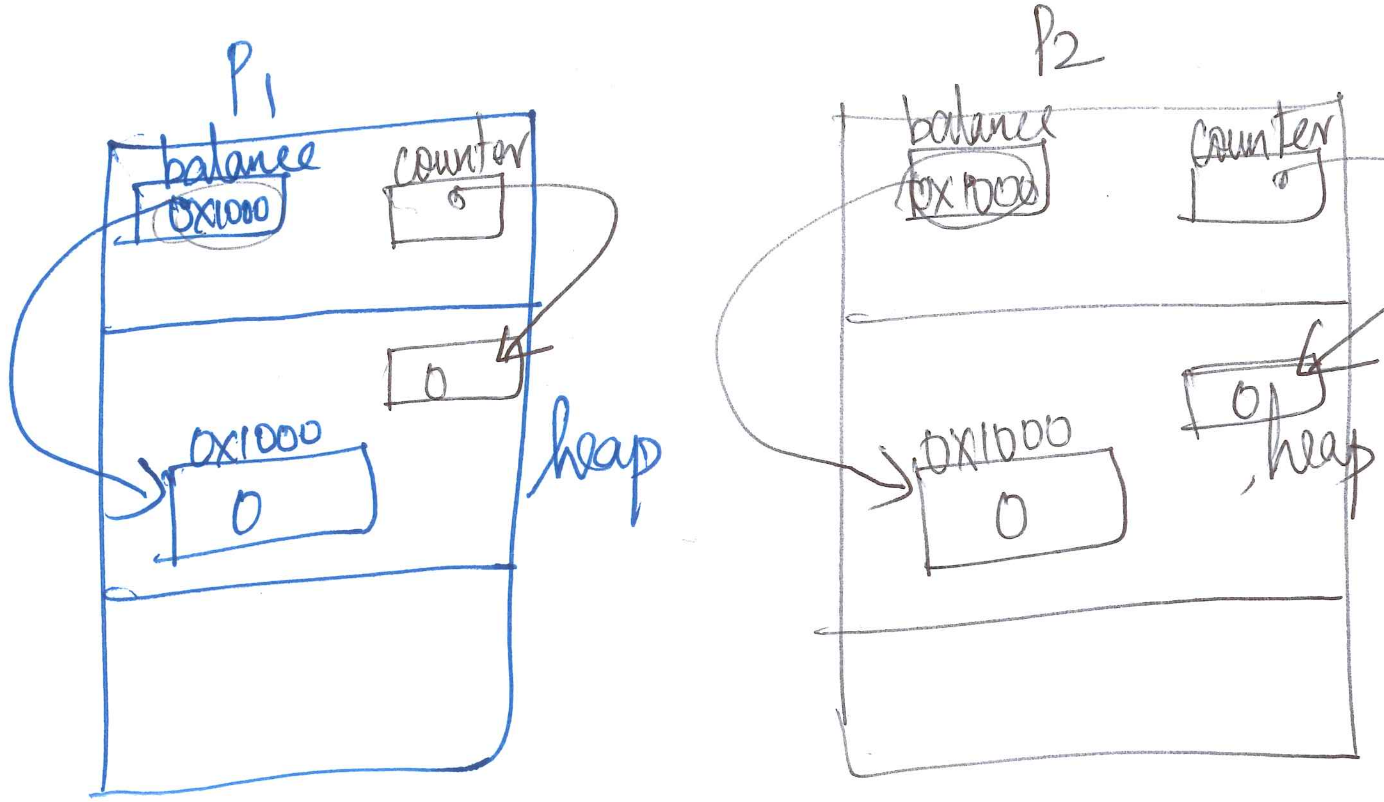
```
printf("%p\n", balance);    printf("%p\n", counter);
```

i. VA of balance in p1 == VA of balance in p2? (TRUE / FALSE)

→ ii. VA of counter in p1 == VA of counter in p2? (TRUE / FALSE)

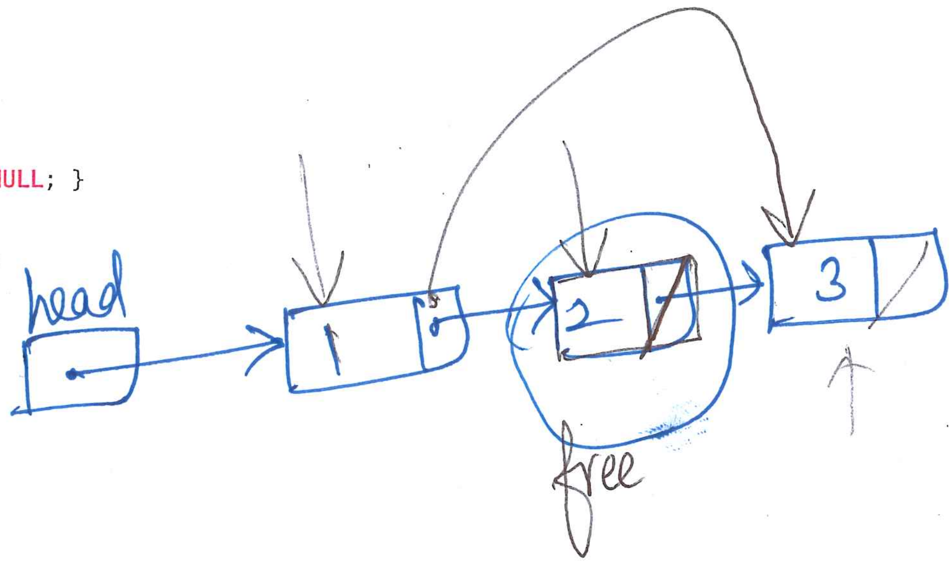
iii. PA of balance in p1 == PA of balance in p2? (TRUE / FALSE)

iv. PA of counter in p1 == PA of counter in p2? (TRUE / FALSE)



P1
balance
0x1000
counter
0
0
0x1000
0
heap
P2
balance
0x1000
counter
0
0
0x1000
0
heap

```c
// Make this Linked List implementation to be thread-safe!

#include <stdio.h>
#include <stdlib.h>

typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

typedef struct __list_t {
  node_t *head;
} list_t;

void List_Init(list_t *L) { L->head = NULL; }

void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return;
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
}

int List_Lookup(list_t *L, int key) {
  node_t *tmp = L->head;
  while (tmp) {
    if (tmp->key == key) return 1;
    tmp = tmp->next;
  }
  return 0;
}

void List_Print(list_t *L) {
  node_t *tmp = L->head;
  while (tmp) {
    printf("%d ", tmp->key);
    tmp = tmp->next;
  }
  printf("\n");
}

int main(int argc, char *argv[]) {
  list_t mylist;
  List_Init(&mylist);
  List_Insert(&mylist, 10);
  List_Insert(&mylist, 30);
  List_Insert(&mylist, 5);
  List_Print(&mylist);
  printf("In List: 10? %d 20? %d\n", List_Lookup(&mylist, 10),
         List_Lookup(&mylist, 20));
  return 0;
}
```



```
void lock(int *lock) {
    while(1) {
        while(LoadLinked(lock) == 1)
            ;
        if(StoreConditional(lock, 1) == 1)   // success
            return;
    }
}
void unlock(int *lock) {
    *lock = 0;
}
```

	LL & SC
Correctness	✓
Fairness	X
Perf	X

TEMPLATE: FILL THIS IN TO MAKE YOUR OWN LOCK

```
typedef struct __lock_t {
   // whatever data structs you need goes here
} lock_t;

void init(lock_t *lock) {
   // init code goes here
}

void acquire(lock_t *lock) {
   // lock acquire code goes here
}

void release(lock_t *lock) {
   // lock release code goes here
}
```

FIRST PRIMITIVE: TEST-AND-SET (or ATOMIC EXCHANGE)

```
// given ptr, sets *ptr to new value; returns the old value at *ptr
int Exchange(int *lock, int new) {
   int old = *lock;
   *lock = new;
   return old;
}
```

SECOND PRIMITIVE: COMPARE-AND-SWAP

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

THIRD PRIMITIVE(S): LOAD-LINKED, STORE-CONDITIONAL

```
int LoadLinked(int *ptr) {
   return *ptr;
}

int StoreConditional(int *ptr, int value) {
   if (no one has updated *ptr since LoadLinked to this address) {
         *ptr = value;
         return 1; // success
   } else {
         return 0; // fail (does not do the store)
   }
}
```

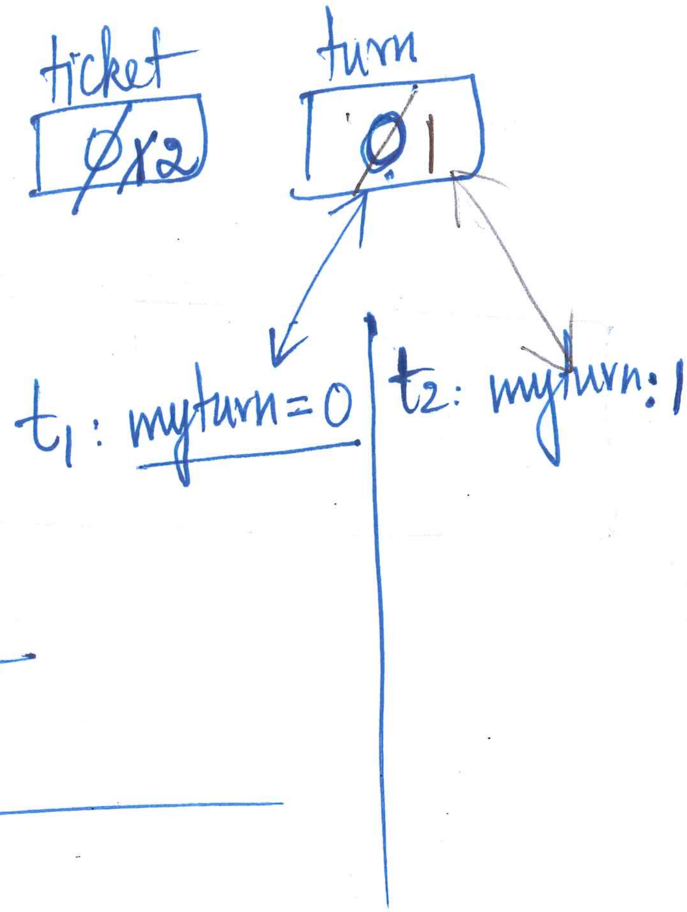
FOURTH PRIMITIVE: FETCH-AND-ADD

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

EXAMPLE USING FETCH-AND-ADD: TICKET LOCKS

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void acquire(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void release(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

yield();



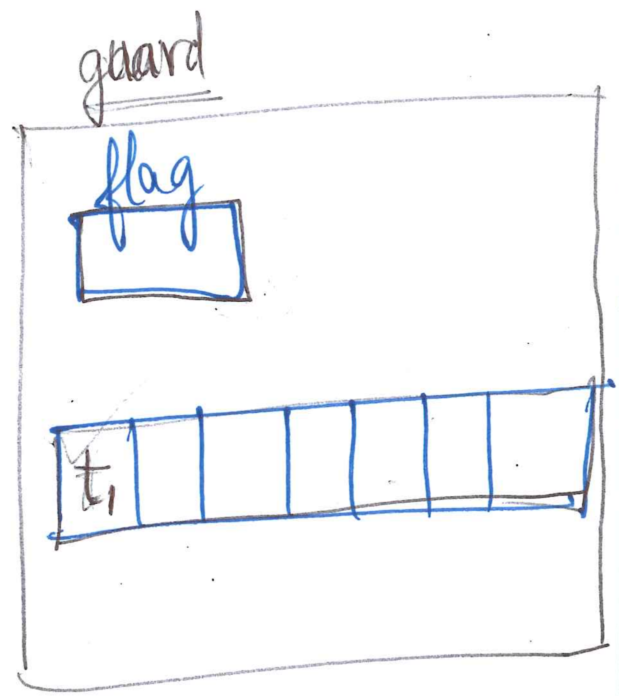
	Ticket Lock	
C	✓	
F	✓	
P	X	

Solaris Lock

park ( ) → thread is put to sleep

unpark (threadId) → wakeup the thread with threadId.

```c
// Lock with Queues, Test-And-Set, Yield, and Wakeup

typedef struct __lock_t {
    int         flag;   // state of lock: 1=held, 0=free
    int         guard;  // use to protect flag, queue
    queue_t     *q;     // explicit queue of waiters
};


void lock_init(lock_t *lock) {
    lock->flag = lock->guard = 0;
    lock->q    = queue_init();
}


void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
        ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
        lock->flag = 1;
        lock->guard = 0;
    } else {               // lock not free: sleep
        queue_push(lock->q, gettid());
        lock->guard = 0;
        park();            // put self to sleep
    }
}


void unlock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
        ; // spin
    if (queue_empty(lock->q))
        lock->flag = 0;
    else
        unpark(queue_pop(lock->q));
    lock->guard = 0;
}
```



CVs

wait (cv , m)

signal (cv)

pthread_cond_t cv;

pthread_mutex_t m;

parent: begin

child

parent: end

Condition Variables

```
mutex_t    lock;  // declare a lock
cond_t     cv;    // declare a condition variable
```

a **condition variable** (CV) is:
- queue of waiting threads

a single **lock** is associated with each CV
- see below for usage

There are two main operations that are important for CVs:

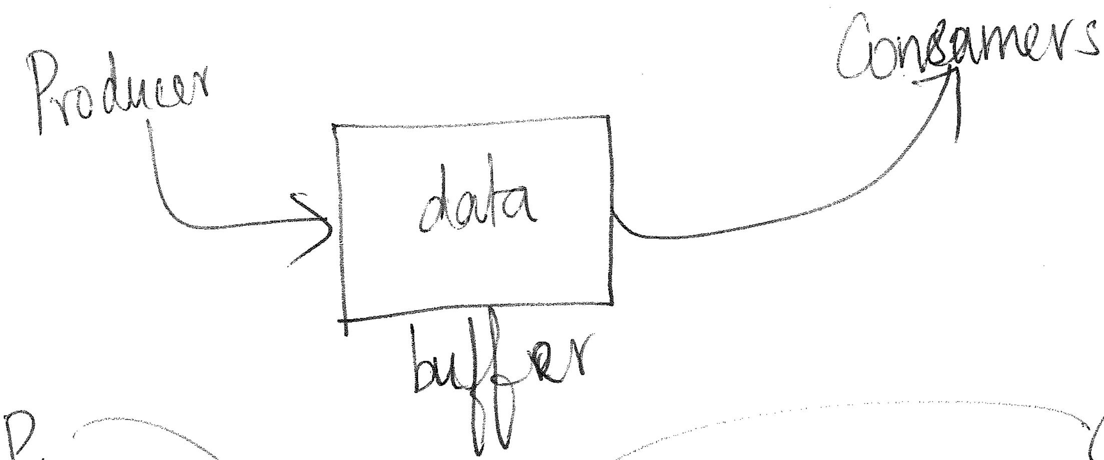
wait (cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

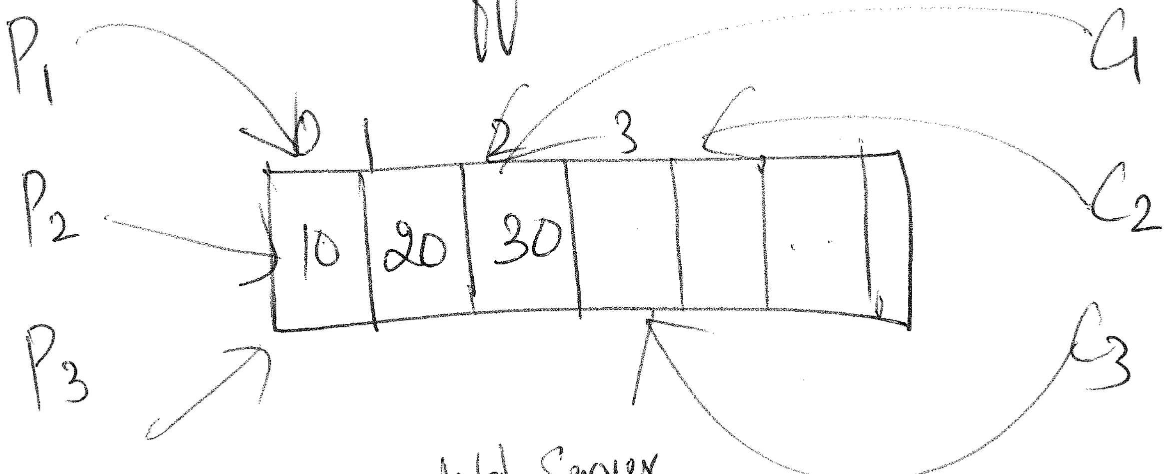
signal (cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

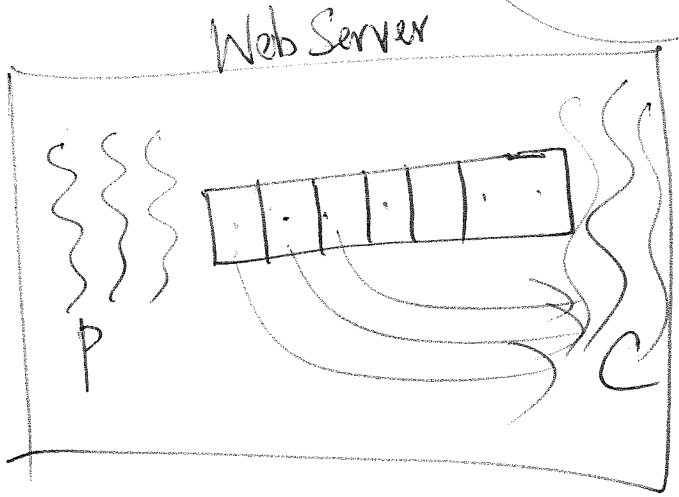
A CV is usually **PAIRED** with some kind **state variable**
- e.g., integer (which indicates the state of the system
                 that we're interested in)

```
int        state; // related "state" variable (could be an int)
```

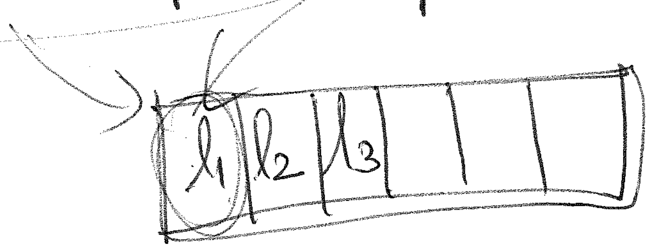
Producer-Consumer Problem







> grep CS537 file.text | wc -l



max=1.

MAIN PROGRAM

```
int main(int argc, char *argv[]) {
    max   = atoi(argv[1]);
    loops = atoi(argv[2]);
    consumers = atoi(argv[3]);
    buffer = (int *) Malloc(max * sizeof(int));
    pthread_t pid, cid[CMAX];
    Pthread_create(&pid, NULL, producer, NULL);
    for (int i = 0; i < consumers; i++)
        Pthread_create(&cid[i], NULL, consumer, NULL);
    Pthread_join(pid, NULL);
    for (i = 0; i < consumers; i++)
        Pthread_join(cid[i], NULL);
}
```

QUEUE GET/PUT

```
void do_fill(int value) {
    buffer[fillptr] = value;
    fillptr = (fillptr + 1) % max;
    numfull++;
}

int do_get() {
    int tmp = buffer[useptr];
    useptr = (useptr + 1) % max;
    numfull--;
    return tmp;
}
```

Solution v1 (Single CV)

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Mutex_lock(&m);             // p1
    while (numfull == max)      // p2
      Cond_wait(&cond, &m);     // p3
    do_fill(i);                 // p4
    Cond_signal(&cond);         // p5
    Mutex_unlock(&m);           // p6
  }
}
void *consumer(void *arg) {
  while (1) {
    Mutex_lock(&m);             // c1
    while (numfull == 0)        // c2
      Cond_wait(&cond, &m);     // c3
    int tmp = do_get();         // c4
    Cond_signal(&cond);         // c5
    Mutex_unlock(&m);           // c6
    printf("%d\n", tmp);
  }
}
```

Solution v2 (2 CVs, "if")

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Mutex_lock(&m);             // p1
    if (numfull == max)         // p2
      Cond_wait(&empty, &m);    // p3
    do_fill(i);                 // p4
    Cond_signal(&fill);         // p5
    Mutex_unlock(&m);           // p6
  }
}
void *consumer(void *arg) {
  while (1) {
    Mutex_lock(&m);             // c1
    if (numfull == 0)           // c2
      Cond_wait(&fill, &m);     // c3
    int tmp = do_get();         // c4
    Cond_signal(&empty);        // c5
    Mutex_unlock(&m);           // c6
    printf("%d\n", tmp);
  }
}
```

Solution v3 (2 CVs, "while")

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Mutex_lock(&m);             // p1
    while (numfull == max)      // p2
      Cond_wait(&empty, &m);    // p3
    do_fill(i);                 // p4
    Cond_signal(&fill);         // p5
    Mutex_unlock(&m);           // p6
  }
}
void *consumer(void *arg) {
  while (1) {
    Mutex_lock(&m);             // c1
    while (numfull == 0)        // c2
      Cond_wait(&fill, &m);     // c3
    int tmp = do_get();         // c4
    Cond_signal(&empty);        // c5
    Mutex_unlock(&m);           // c6
    printf("%d\n", tmp);
  }
}
```

Solution v4 (2 CVs, "while", unlock)

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Mutex_lock(&m);             // p1
    while (numfull == max)      // p2
      Cond_wait(&empty, &m);    // p3
    Mutex_unlock(&m);           // p3a
    do_fill(i);                 // p4
    Mutex_lock(&m);             // p4a
    Cond_signal(&fill);         // p5
    Mutex_unlock(&m);           // p6
  }
}
void *consumer(void *arg) {
  while (1) {
    Mutex_lock(&m);             // c1
    while (numfull == 0)        // c2
      Cond_wait(&fill, &m);     // c3
    Mutex_unlock(&m);           // c3a
    int tmp = do_get();         // c4
    Mutex_lock(&m);             // c4a
    Cond_signal(&empty);        // c5
    Mutex_unlock(&m);           // c6
  }
}
```

S1

P: P1 P2 P4 P5 P6 P1 P2 P3 (wait) P2 P4 . . .

C: C_1 C_2 C_4 C_5 (signal) (now awake) C_6 C_1 C_2 C_3 (wait)

P: P_1 P_2 P_4 P_5 (signal) → (Ca awake) P_6 P_1 P_2 P_3 (wait)

Ca: C_1 C_2 C_3 wait → (Ca awake) C_2 C_4 C_5 (signal) C_6 C_1 C_2 C_3 (wait)

Cb: C_1 C_2 C_3 (wait) ... C_2 C_3 (wait)

Soln #2

C_a: C_1 C_2 C_3 (wait) (awake) ← C_4 CRASH — get on an empty buffer.

P: P_1 P_2 P_4 P_5 (signal) P_6 P_1 P_2 P_3 (wait) (awake)

C_b: C_1 C_2 C_4 C_5 (signal) C_6 C_1 C_2 C_3 (wait)

Mesa semantics (widely used)

Hoare "