SEMAPHORE: DEFINITION

```
sem_init(sem_t *s, int value) {
    s->value = value;
}
sem_wait(sem_t *s) {
    while (s->value <= 0)
        put_self_to_sleep();
    s->value--;
}
sem_post(sem_t *s) {
    s->value++;
    wake_one_waiting_thread();
}
// Each routine executes ATOMICALLY
```

#1: Mutual Exclusion

```
sem_t lock;
void *worker(void *arg) {
    int i;
    // What goes here?
    for (i = 0; i < 1e6; i++)
        counter++;
    // What goes here?
    return NULL;
}
int main(int argc, char *argv[]) {
  int num = atoi(argv[1]);
  pthread_t pid[PMAX];
  sem_init(&lock, /* What goes here? */);
  for (int i = 0; i < num; i++)
    Pthread_create(&pid[i], 0, worker, 0);
  for (int i = 0; i < num; i++)
    Pthread_join(pid[i], NULL);
  printf("counter: %d\n", counter);
  return 0;
}
```

#2: Fork/Join

```
sem_t s;
void *child(void *arg) {
  printf("child\n");
  // What goes here?
  return NULL;
}
int main(int argc, char *argv[]) {
  pthread_t p;
  printf("parent: begin\n");
  sem_init(&s, /* What goes here? */);
  Pthread_create(&p, 0, child, 0);
  // What goes here?
  printf("parent: end\n");
  return 0;
}
```

#3: Reader/Writer Locks

```
typedef struct _rwlock_t {
  sem_t write_lock;
  sem_t lock;
  int readers;
} rwlock_t;

void rw_init(rwlock_t *L) {
  L->readers = 0;
  sem_init(&L->lock, 1);
  sem_init(&L->write_lock, 1);
}
void acquire_readlock(rwlock_t *L){
  sem_wait(&L->lock);          // ra1
  L->readers++;                // ra2
  if (L->readers == 1)         // ra3
    sem_wait(&L->write_lock); // ra4
  sem_post(&L->lock);          // ra5
}
void release_readlock(rwlock_t *L){
  sem_wait(&L->lock);          // rr1
  L->readers--;                // rr2
  if (L->readers == 0)         // rr3
    sem_post(&L->write_lock); // rr4
  sem_post(&L->lock);          // rr5
}
void acquire_writelock(rwlock_t *L){
  sem_wait(&L->write_lock);
}
void release_writelock(rwlock_t *L){
  sem_post(&L->write_lock);
}
```

#4: Zemaphores

```
typedef struct __Zem_t {
    int             value;
    pthread_cond_t  cond;
    pthread_mutex_t lock;
} Zem_t;
// init() only called by one thread
void Zem_init(Zem_t *z, int value) {
    z->value = value;
    // What to write here?
}
void Zem_wait(Zem_t *z) {
    // Code? Use sem def as guide
}
void Zem_post(Zem_t *z) {
    // Code? Use sem def as guide
}
```