EE516 : Homework 2

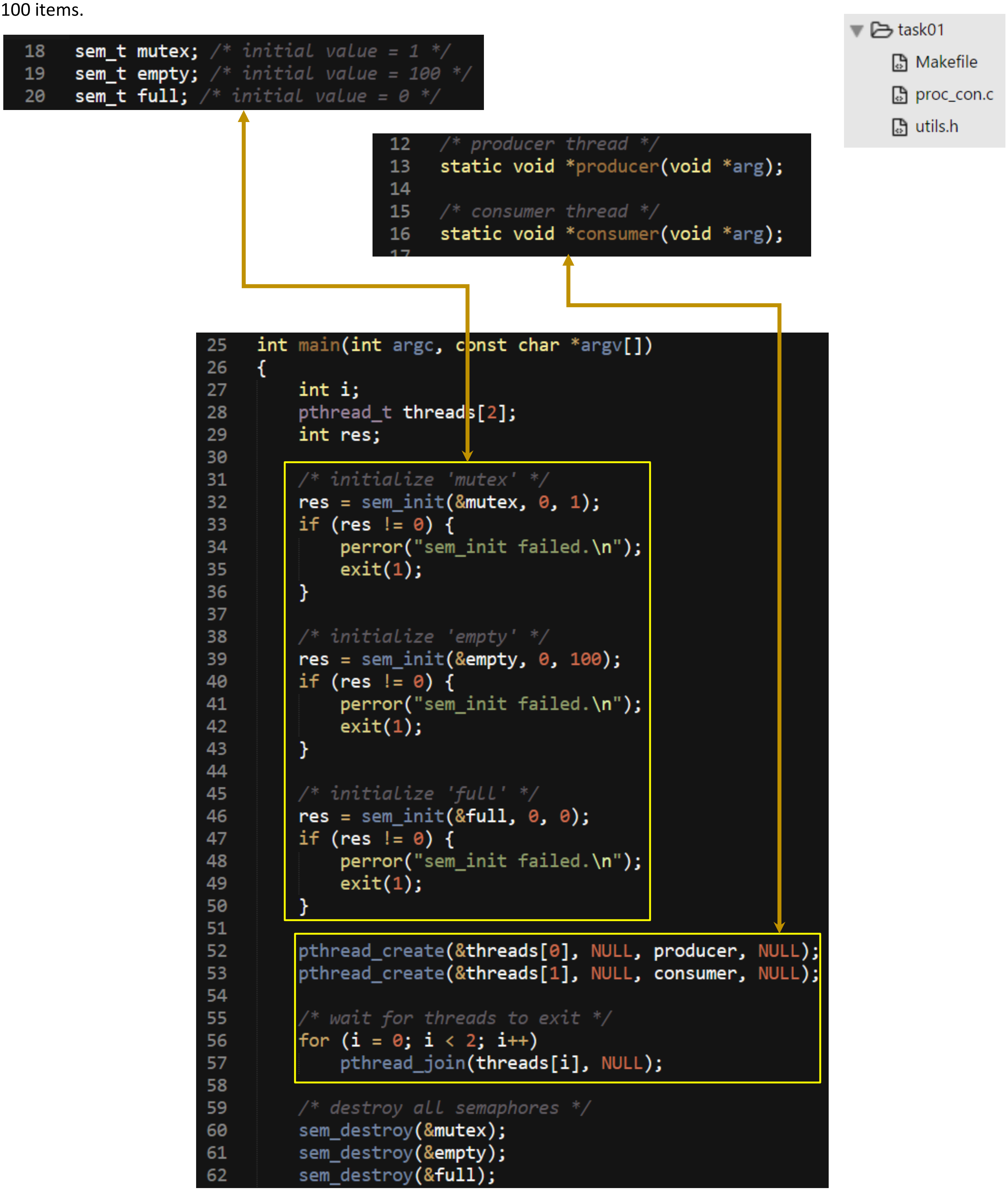
Gaurav Kalra

(Student ID: 2016 45 93)

gvkalra@kaist.ac.kr

Problem 1: Write code for producer & consumer problem

Assume that there is one producer and one consumer, where each producer produces 100 items and each consumer consumes 100 items.

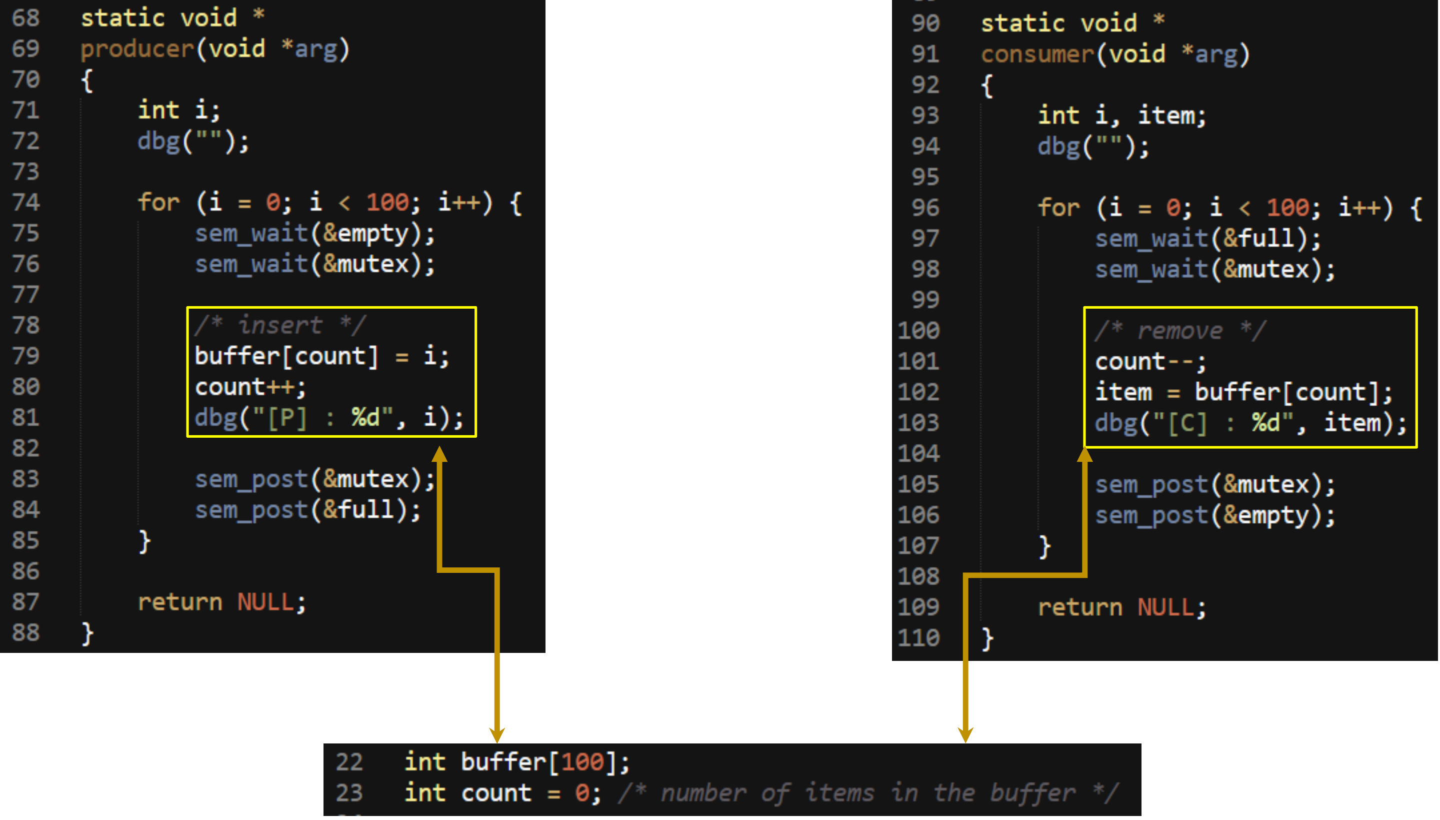


```c
sem_t mutex; /* initial value = 1 */
sem_t empty; /* initial value = 100 */
sem_t full; /* initial value = 0 */
```

```c
/* producer thread */
static void *producer(void *arg);

/* consumer thread */
static void *consumer(void *arg);
```

```c
int main(int argc, const char *argv[])
{
    int i;
    pthread_t threads[2];
    int res;

    /* initialize 'mutex' */
    res = sem_init(&mutex, 0, 1);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    /* initialize 'empty' */
    res = sem_init(&empty, 0, 100);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    /* initialize 'full' */
    res = sem_init(&full, 0, 0);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    pthread_create(&threads[0], NULL, producer, NULL);
    pthread_create(&threads[1], NULL, consumer, NULL);

    /* wait for threads to exit */
    for (i = 0; i < 2; i++)
        pthread_join(threads[i], NULL);

    /* destroy all semaphores */
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
```

Figure 1: Spawning producer-consumer threads, setting up semaphores



```c
static void *
producer(void *arg)
{
    int i;
    dbg("");

    for (i = 0; i < 100; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);

        /* insert */
        buffer[count] = i;
        count++;
        dbg("[P] : %d", i);

        sem_post(&mutex);
        sem_post(&full);
    }

    return NULL;
}
```

```c
static void *
consumer(void *arg)
{
    int i, item;
    dbg("");

    for (i = 0; i < 100; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        /* remove */
        count--;
        item = buffer[count];
        dbg("[C] : %d", item);

        sem_post(&mutex);
        sem_post(&empty);
    }

    return NULL;
}
```

```c
int buffer[100];
int count = 0; /* number of items in the buffer */
```

Figure 2: Producer & Consumer synchronizing access for shared data

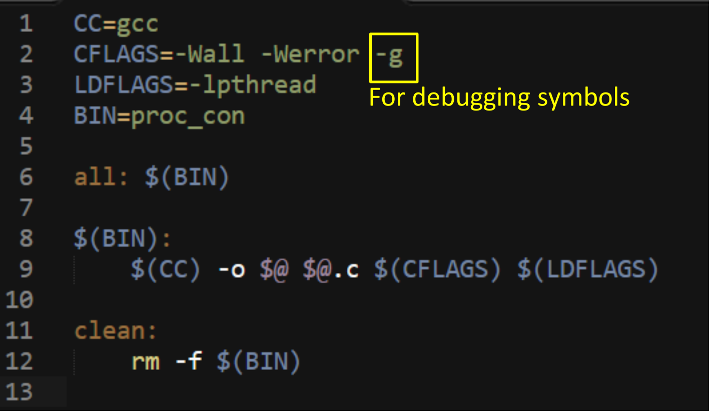
sem_wait(&variable) will atomically decrement (lock) ‘variable’

if ‘variable’ = 0, it will wait (block) until the value becomes > 0

else, it will simply decrement ‘variable’ atomically

sem_post(&variable) will atomically increment (unlock) ‘variable’

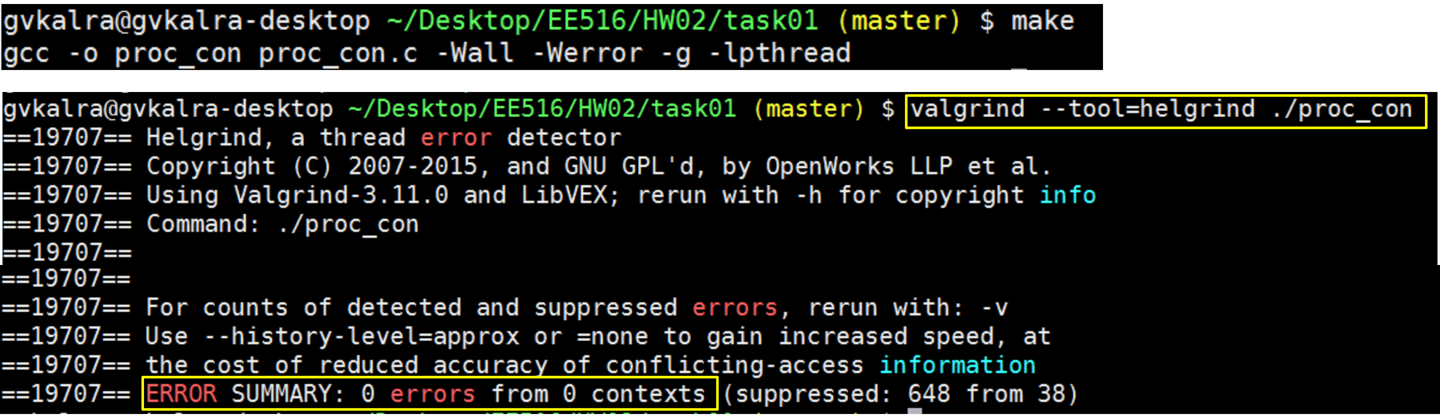
if ‘variable’ consequently becomes > 0, then another task blocked in sem_wait() will be woken up & proceed to decrement (lock) ‘variable’



```
CC=gcc
CFLAGS=-Wall -Werror -g
LDFLAGS=-lpthread
BIN=proc_con

all: $(BIN)

$(BIN):
	$(CC) -o $@ $@.c $(CFLAGS) $(LDFLAGS)

clean:
	rm -f $(BIN)
```

For debugging symbols

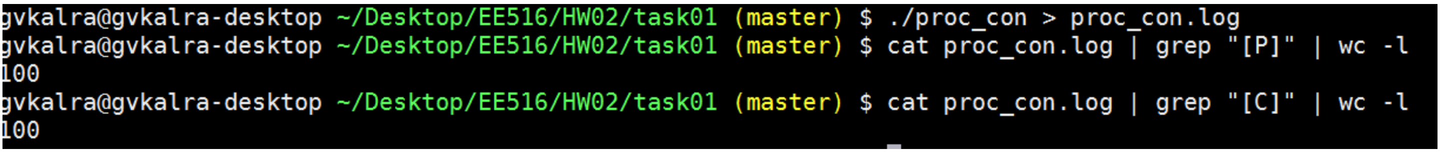
Figure 3: Makefile



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task01 (master) $ make
gcc -o proc_con proc_con.c -Wall -Werror -g -lpthread
```

```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task01 (master) $ valgrind --tool=helgrind ./proc_con
==19707== Helgrind, a thread error detector
==19707== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==19707== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==19707== Command: ./proc_con
==19707==
==19707==
==19707== For counts of detected and suppressed errors, rerun with: -v
==19707== Use --history-level=approx or =none to gain increased speed, at
==19707== the cost of reduced accuracy of conflicting-access information
==19707== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 648 from 38)
```

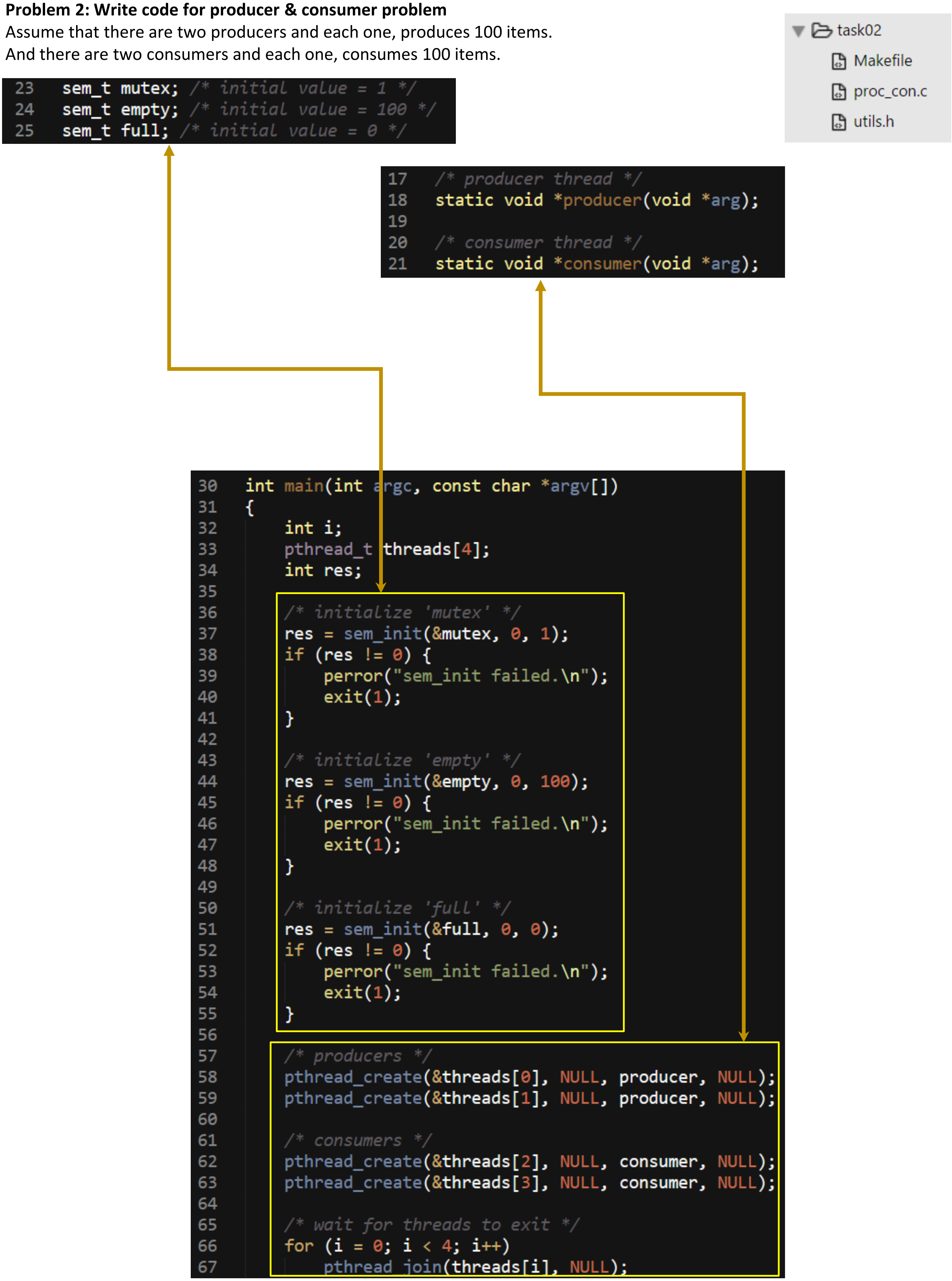
Figure 4: Verification using “helgrind”



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task01 (master) $ ./proc_con > proc_con.log
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task01 (master) $ cat proc_con.log | grep "[P]" | wc -l
100
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task01 (master) $ cat proc_con.log | grep "[C]" | wc -l
100
```

Figure 5: Verification using logs (P & C must be 100 each)

Problem 2: Write code for producer & consumer problem
Assume that there are two producers and each one, produces 100 items.
And there are two consumers and each one, consumes 100 items.

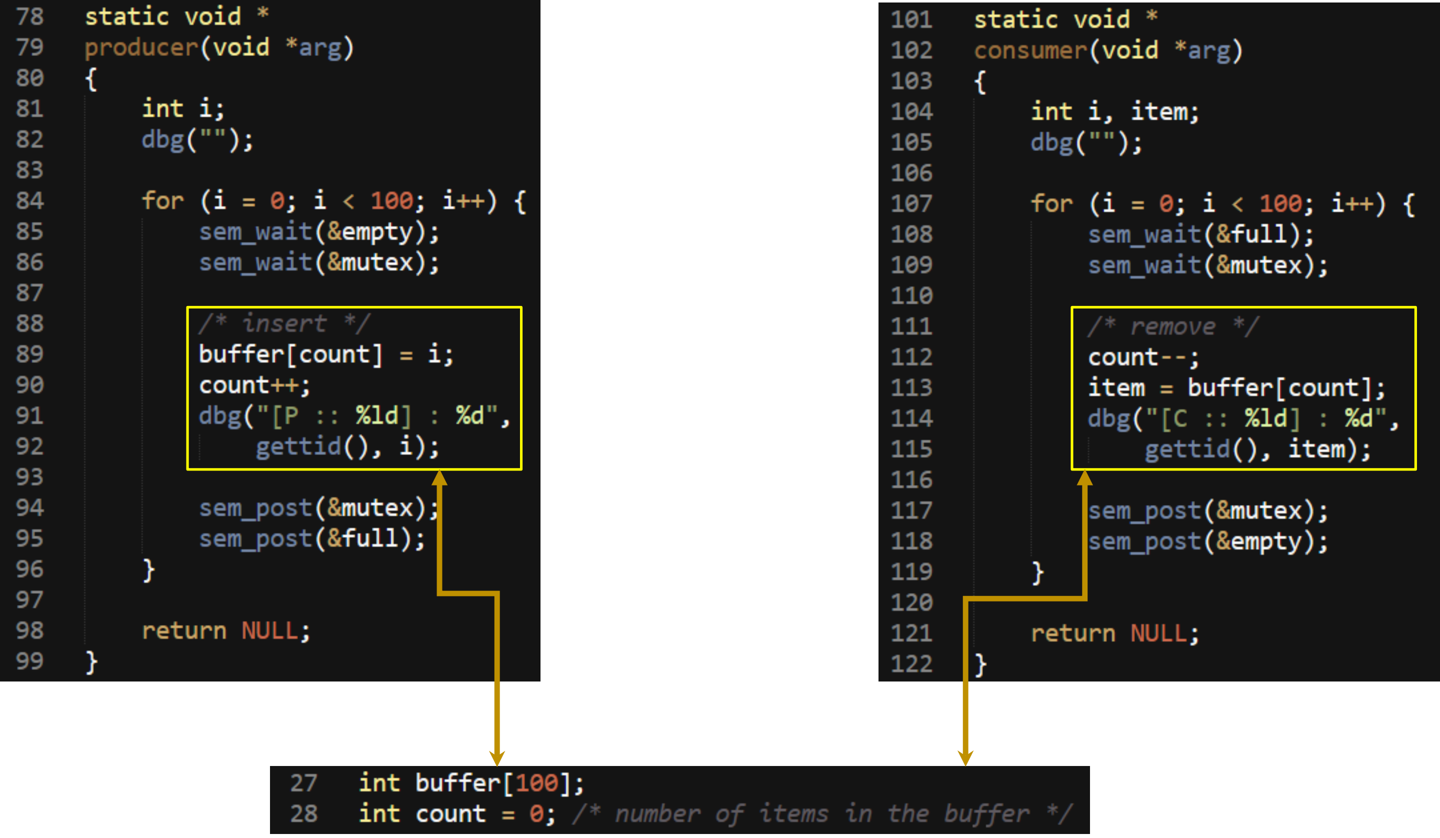


```
sem_t mutex; /* initial value = 1 */
sem_t empty; /* initial value = 100 */
sem_t full; /* initial value = 0 */
```

```
/* producer thread */
static void *producer(void *arg);

/* consumer thread */
static void *consumer(void *arg);
```

```
int main(int argc, const char *argv[])
{
    int i;
    pthread_t threads[4];
    int res;

    /* initialize 'mutex' */
    res = sem_init(&mutex, 0, 1);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    /* initialize 'empty' */
    res = sem_init(&empty, 0, 100);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    /* initialize 'full' */
    res = sem_init(&full, 0, 0);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    /* producers */
    pthread_create(&threads[0], NULL, producer, NULL);
    pthread_create(&threads[1], NULL, producer, NULL);

    /* consumers */
    pthread_create(&threads[2], NULL, consumer, NULL);
    pthread_create(&threads[3], NULL, consumer, NULL);

    /* wait for threads to exit */
    for (i = 0; i < 4; i++)
        pthread_join(threads[i], NULL);
```

Figure 6: Spawning producer-consumer threads, setting up semaphores

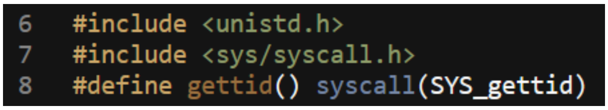


```c
static void *
producer(void *arg)
{
    int i;
    dbg("");

    for (i = 0; i < 100; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);

        /* insert */
        buffer[count] = i;
        count++;
        dbg("[P :: %ld] : %d",
            gettid(), i);

        sem_post(&mutex);
        sem_post(&full);
    }

    return NULL;
}
```

```c
static void *
consumer(void *arg)
{
    int i, item;
    dbg("");

    for (i = 0; i < 100; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        /* remove */
        count--;
        item = buffer[count];
        dbg("[C :: %ld] : %d",
            gettid(), item);

        sem_post(&mutex);
        sem_post(&empty);
    }

    return NULL;
}
```

```c
int buffer[100];
int count = 0; /* number of items in the buffer */
```

Figure 7: Producer & Consumer synchronizing access for shared data

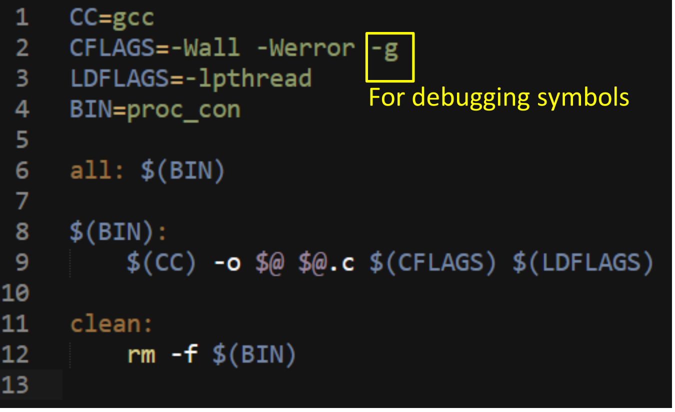


```c
#include <unistd.h>
#include <sys/syscall.h>
#define gettid() syscall(SYS_gettid)
```

Figure 8: gettid() for fetching thread-ID

sem_wait(&variable) will atomically decrement (lock) ‘variable’
  if ‘variable’ = 0, it will wait (block) until the value becomes > 0
  else, it will simply decrement ‘variable’ atomically

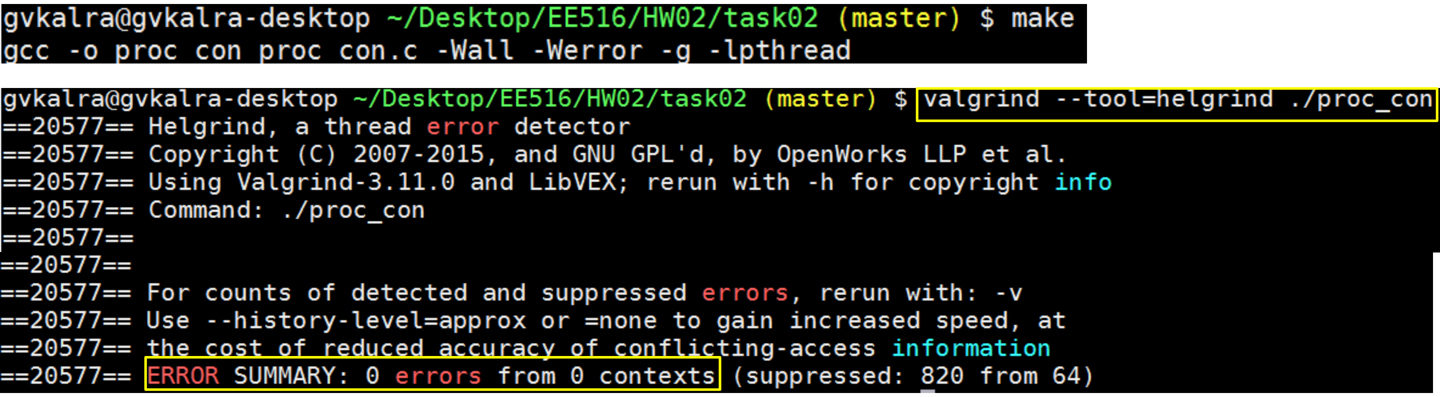
sem_post(&variable) will atomically increment (unlock) ‘variable’
  if ‘variable’ consequently becomes > 0, then another task blocked in sem_wait() will be woken up & proceed to decrement (lock) ‘variable’



```
CC=gcc
CFLAGS=-Wall -Werror -g
LDFLAGS=-lpthread
BIN=proc_con

all: $(BIN)

$(BIN):
	$(CC) -o $@ $@.c $(CFLAGS) $(LDFLAGS)

clean:
	rm -f $(BIN)
```

For debugging symbols

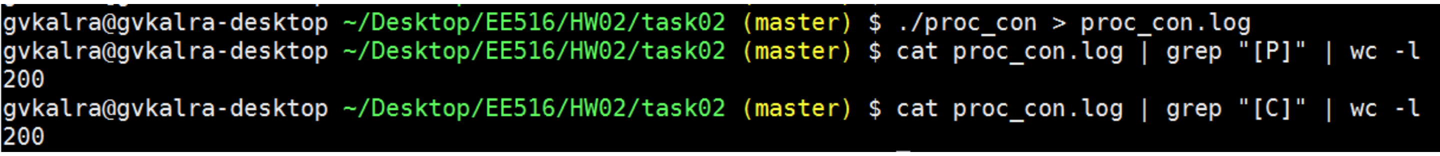
Figure 9: Makefile



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task02 (master) $ make
gcc -o proc_con proc_con.c -Wall -Werror -g -lpthread
```

```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task02 (master) $ valgrind --tool=helgrind ./proc_con
==20577== Helgrind, a thread error detector
==20577== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==20577== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20577== Command: ./proc_con
==20577==
==20577==
==20577== For counts of detected and suppressed errors, rerun with: -v
==20577== Use --history-level=approx or =none to gain increased speed, at
==20577== the cost of reduced accuracy of conflicting-access information
==20577== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 820 from 64)
```

Figure 10: Verification using “helgrind”

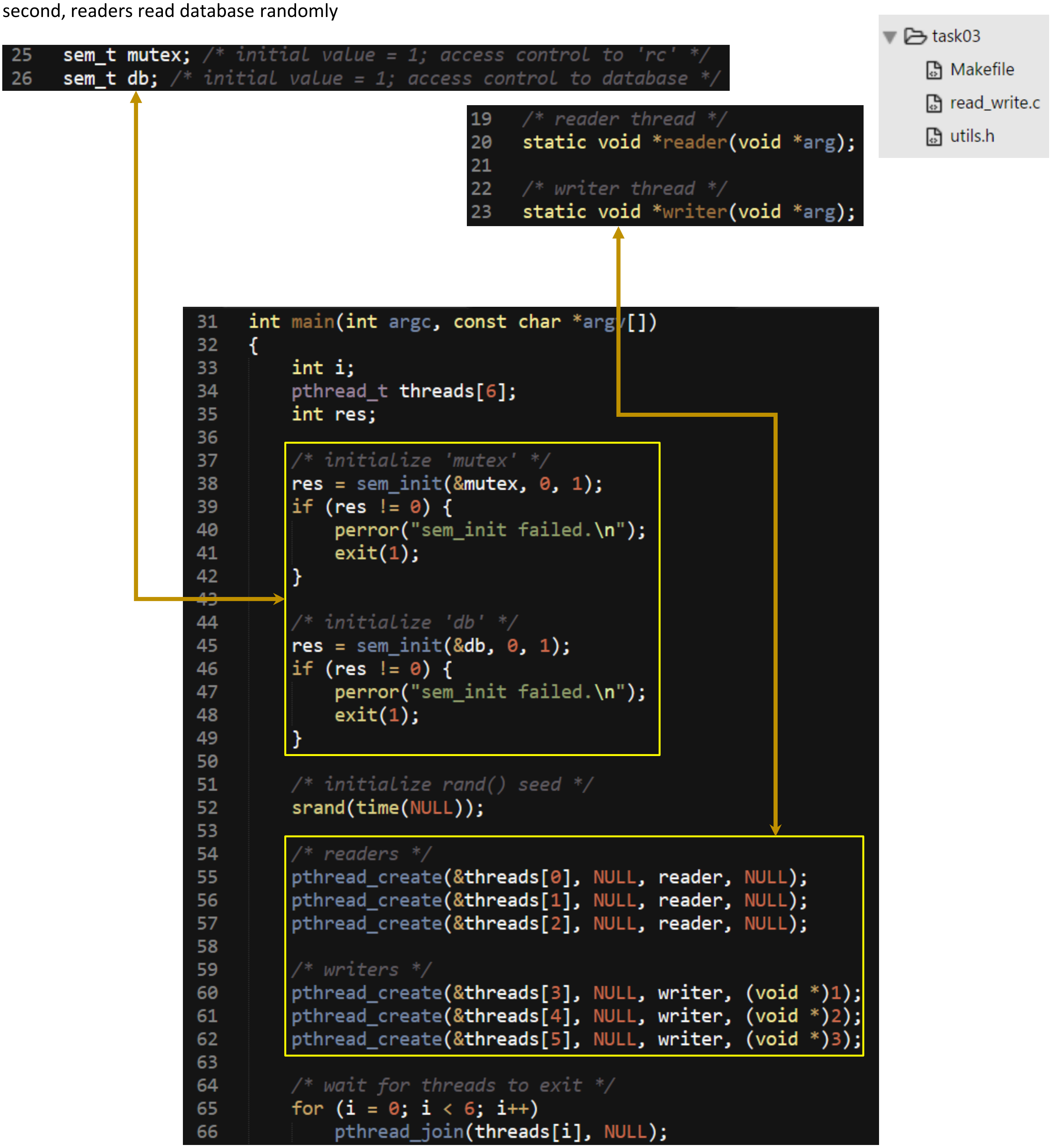


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task02 (master) $ ./proc_con > proc_con.log
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task02 (master) $ cat proc_con.log | grep "[P]" | wc -l
200
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task02 (master) $ cat proc_con.log | grep "[C]" | wc -l
200
```

Figure 11: Verification using logs (P & C must be 200 each)

Problem 3: Write code for reader & writer problem

Assume that there are 3 readers and 3 writers, the size of database is only 1 byte, writer-i writes i to the database every 1 second, readers read database randomly



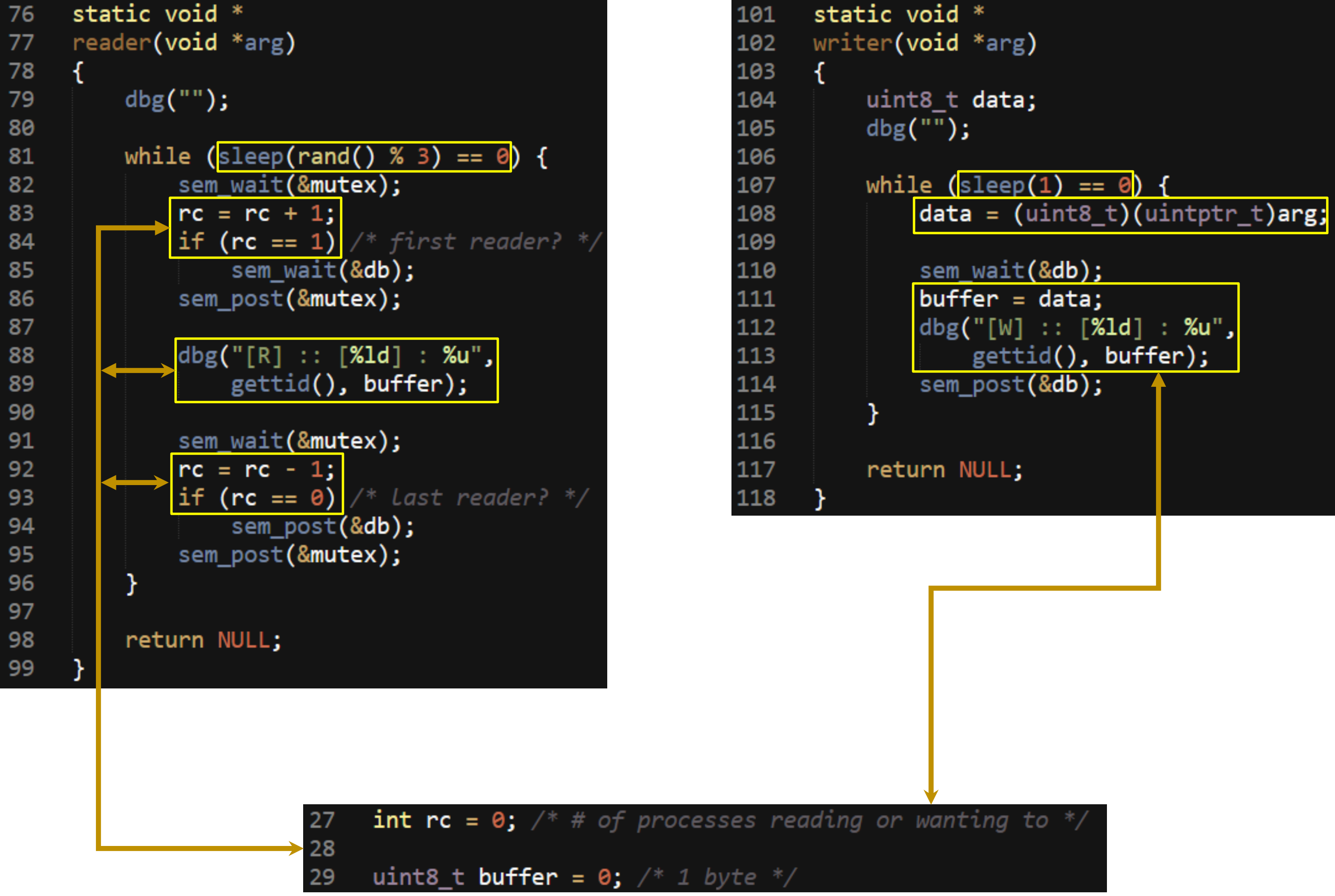
task03
- Makefile
- read_write.c
- utils.h

```c
sem_t mutex; /* initial value = 1; access control to 'rc' */
sem_t db; /* initial value = 1; access control to database */
```

```c
/* reader thread */
static void *reader(void *arg);

/* writer thread */
static void *writer(void *arg);
```

```c
int main(int argc, const char *argv[])
{
    int i;
    pthread_t threads[6];
    int res;

    /* initialize 'mutex' */
    res = sem_init(&mutex, 0, 1);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    /* initialize 'db' */
    res = sem_init(&db, 0, 1);
    if (res != 0) {
        perror("sem_init failed.\n");
        exit(1);
    }

    /* initialize rand() seed */
    srand(time(NULL));

    /* readers */
    pthread_create(&threads[0], NULL, reader, NULL);
    pthread_create(&threads[1], NULL, reader, NULL);
    pthread_create(&threads[2], NULL, reader, NULL);

    /* writers */
    pthread_create(&threads[3], NULL, writer, (void *)1);
    pthread_create(&threads[4], NULL, writer, (void *)2);
    pthread_create(&threads[5], NULL, writer, (void *)3);

    /* wait for threads to exit */
    for (i = 0; i < 6; i++)
        pthread_join(threads[i], NULL);
```

Figure 12: Spawning reader-writer threads, setting up semaphores



```
static void *
reader(void *arg)
{
    dbg("");

    while (sleep(rand() % 3) == 0) {
        sem_wait(&mutex);
        rc = rc + 1;
        if (rc == 1) /* first reader? */
            sem_wait(&db);
        sem_post(&mutex);

        dbg("[R] :: [%ld] : %u",
            gettid(), buffer);

        sem_wait(&mutex);
        rc = rc - 1;
        if (rc == 0) /* last reader? */
            sem_post(&db);
        sem_post(&mutex);
    }

    return NULL;
}
```

```
static void *
writer(void *arg)
{
    uint8_t data;
    dbg("");

    while (sleep(1) == 0) {
        data = (uint8_t)(uintptr_t)arg;

        sem_wait(&db);
        buffer = data;
        dbg("[W] :: [%ld] : %u",
            gettid(), buffer);
        sem_post(&db);
    }

    return NULL;
}
```

```
int rc = 0; /* # of processes reading or wanting to */

uint8_t buffer = 0; /* 1 byte */
```

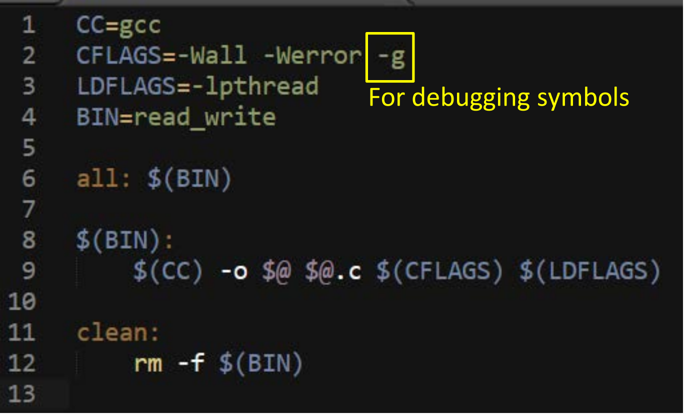
Figure 13: Reader & Writer synchronizing access for shared data

sem_wait(&variable) will atomically decrement (lock) ‘variable’
if ‘variable’ = 0, it will wait (block) until the value becomes > 0
else, it will simply decrement ‘variable’ atomically

sem_post(&variable) will atomically increment (unlock) ‘variable’
if ‘variable’ consequently becomes > 0, then another task blocked in sem_wait() will be woken up & proceed to decrement (lock) ‘variable’

rand() is used to make the reader **sleep**() for random seconds (between 0 to 2).

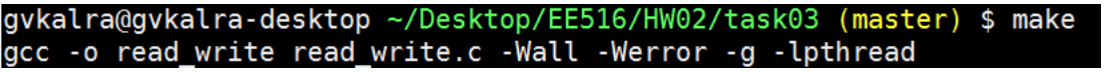
At any given time, there can be many ‘readers’ reading the database. However, during such times, database access is not given to any ‘writer’. Consequently, if database is locked for writing, no reader is allowed access to the database.



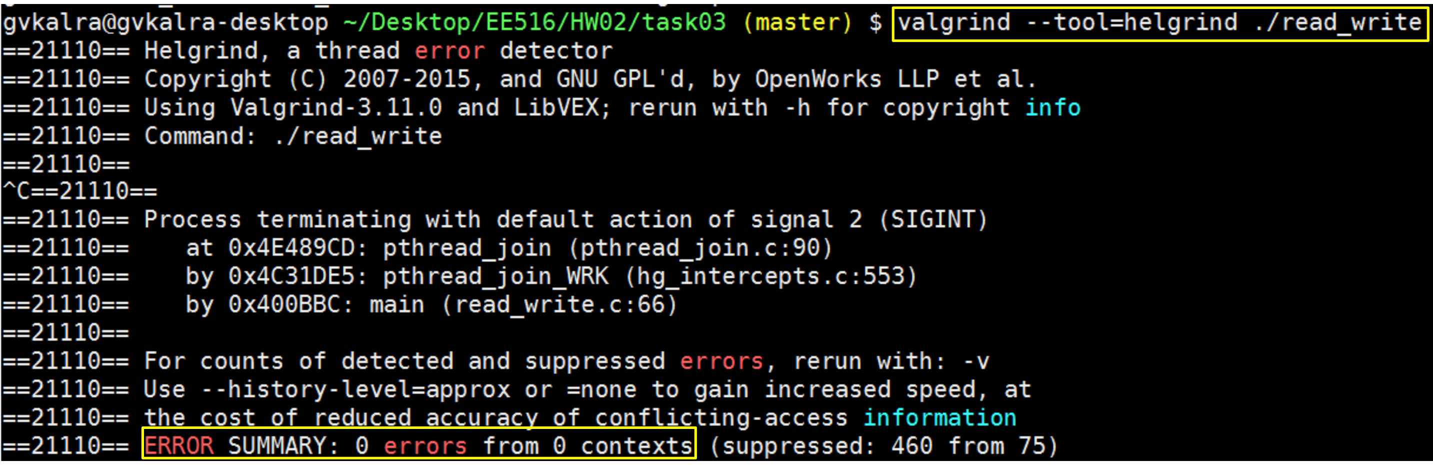
```
CC=gcc
CFLAGS=-Wall -Werror -g
LDFLAGS=-lpthread
BIN=read_write

all: $(BIN)

$(BIN):
	$(CC) -o $@ $@.c $(CFLAGS) $(LDFLAGS)

clean:
	rm -f $(BIN)

```

For debugging symbols

Figure 14: Makefile

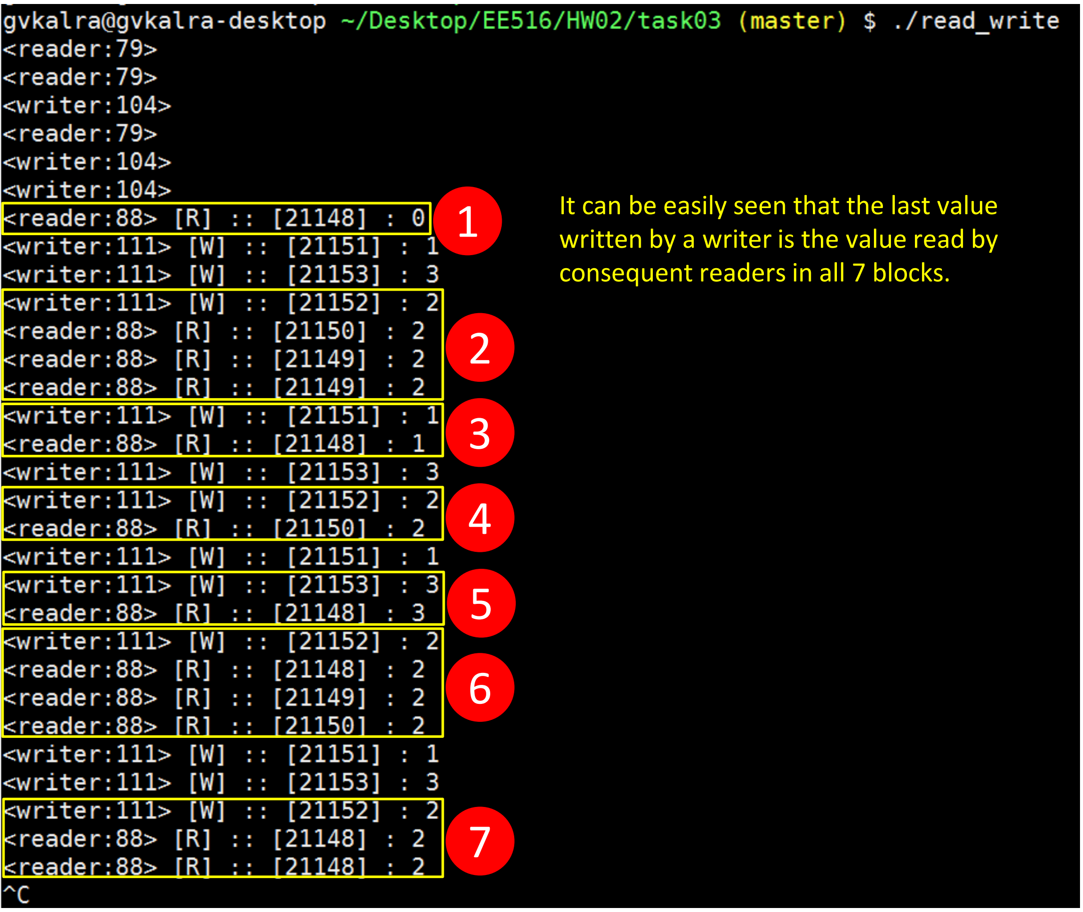


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task03 (master) $ make
gcc -o read_write read_write.c -Wall -Werror -g -lpthread
```



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task03 (master) $ valgrind --tool=helgrind ./read_write
==21110== Helgrind, a thread error detector
==21110== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==21110== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==21110== Command: ./read_write
==21110==
^C==21110==
==21110== Process terminating with default action of signal 2 (SIGINT)
==21110==    at 0x4E489CD: pthread_join (pthread_join.c:90)
==21110==    by 0x4C31DE5: pthread_join_WRK (hg_intercepts.c:553)
==21110==    by 0x400BBC: main (read_write.c:66)
==21110==
==21110== For counts of detected and suppressed errors, rerun with: -v
==21110== Use --history-level=approx or =none to gain increased speed, at
==21110== the cost of reduced accuracy of conflicting-access information
==21110== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 460 from 75)
```

Figure 15: Verification using “helgrind”



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW02/task03 (master) $ ./read_write
<reader:79>
<reader:79>
<writer:104>
<reader:79>
<writer:104>
<writer:104>
<reader:88> [R] :: [21148] : 0
<writer:111> [W] :: [21151] : 1
<writer:111> [W] :: [21153] : 3
<writer:111> [W] :: [21152] : 2
<reader:88> [R] :: [21150] : 2
<reader:88> [R] :: [21149] : 2
<reader:88> [R] :: [21149] : 2
<writer:111> [W] :: [21151] : 1
<reader:88> [R] :: [21148] : 1
<writer:111> [W] :: [21153] : 3
<writer:111> [W] :: [21152] : 2
<reader:88> [R] :: [21150] : 2
<writer:111> [W] :: [21151] : 1
<writer:111> [W] :: [21153] : 3
<reader:88> [R] :: [21148] : 3
<writer:111> [W] :: [21152] : 2
<reader:88> [R] :: [21148] : 2
<reader:88> [R] :: [21149] : 2
<reader:88> [R] :: [21150] : 2
<writer:111> [W] :: [21151] : 1
<writer:111> [W] :: [21153] : 3
<writer:111> [W] :: [21152] : 2
<reader:88> [R] :: [21148] : 2
<reader:88> [R] :: [21148] : 2
^C
```

It can be easily seen that the last value written by a writer is the value read by consequent readers in all 7 blocks.

Figure 16: Verification using logs