Lab 1 Writeup

Student Number: A0235422N

Name: Xiao Zhehao

Ex7

```
/******************************************************************************
 * ex789-prod-con-threads.cpp
 * Producer-consumer synchronisation problem in C++
 *****************************************************************************/

#include <cstdio>
#include <cstdlib>
#include <pthread.h>
#include <signal.h>

#define DO_LOGGING false

constexpr int PRODUCERS = 2;
constexpr int CONSUMERS = 1;

constexpr int MAX_BUFFER_LEN = 10;
int consumer_sum = 0;
int producer_buffer[MAX_BUFFER_LEN];
/* !!!!!! KEY ASSUMPTION !!!!!! That the program will not run long enought
for curr_prod or curr_cons to overflow. */
int curr_prod = 0; // Tracks production idx
int curr_cons = 0; // Tracks consumption idx
pthread_mutex_t shared_var_lock = PTHREAD_MUTEX_INITIALIZER; // locks both
the producer buffer and consumer sum
pthread_cond_t wait_not_full_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t wait_not_empty_cond = PTHREAD_COND_INITIALIZER;

volatile bool has_ended = false;

void *producer(void *threadid)
{
  int tid = *((int *)threadid);
    while (!has_ended) {
    int randInt = (rand() % 10) + 1; // random number between 1 to 10

    pthread_mutex_lock(&shared_var_lock); // Get Mutex
    // Wait for there to be empty space on the buffer, check on has_ended
also to prevent infinite wait on termination and full buffer
    while ((curr_prod - curr_cons >= MAX_BUFFER_LEN) && !has_ended) {
      pthread_cond_wait(&wait_not_full_cond, &shared_var_lock);
    }
    producer_buffer[curr_prod % MAX_BUFFER_LEN] = randInt;
    ++curr_prod;
```

```
    #if DO_LOGGING
    printf("Producer %d adding %d to buffer, curr buffer size %d.\n", tid,
randInt, curr_prod - curr_cons);
    #endif
    pthread_mutex_unlock(&shared_var_lock); // Release Mutex
    pthread_cond_broadcast(&wait_not_empty_cond); // As we added to buffer
it is definitely not empty
  }
  printf("Producer %d exiting.\n", tid);
  return NULL;
}

void *consumer(void *threadid)
{
  int tid = *((int *)threadid);
    while (!has_ended) {
    pthread_mutex_lock(&shared_var_lock); // Get Mutex
    // Wait for there to be value on the buffer, check on has_ended also
to prevent infinite wait on termination and empty buffer
    while ((curr_prod - curr_cons <= 0) && !has_ended) {
      pthread_cond_wait(&wait_not_empty_cond, &shared_var_lock);
    }
    int consume_idx = curr_cons % MAX_BUFFER_LEN;
    consumer_sum += producer_buffer[consume_idx];
    ++curr_cons;
    #if DO_LOGGING
    printf("Consumer %d taking %d from buffer, curr buffer size %d. Sum
now %d.\n", tid, producer_buffer[consume_idx], curr_prod - curr_cons,
consumer_sum);
    #endif
    pthread_mutex_unlock(&shared_var_lock); // Release Mutex
    pthread_cond_broadcast(&wait_not_full_cond); // As we took from buffer
it is definitely not full
  }
  printf("Consumer %d exiting.\n", tid);
  return NULL;
}

void handle_sigint(int sig) {
  printf("\nCaught signal %d (SIGINT). Exiting gracefully...\n", sig);
  has_ended = true;
}

int main(int argc, char *argv[])
{
    pthread_t producer_threads[PRODUCERS];
    pthread_t consumer_threads[CONSUMERS];
    int producer_threadid[PRODUCERS];
    int consumer_threadid[CONSUMERS];

    int rc;
    int t1, t2;

  sigset_t omask, mask;
```

```
  sigfillset(&mask);
  pthread_sigmask(SIG_SETMASK, &mask, &omask);

    for (t1 = 0; t1 < PRODUCERS; t1++)
    {
        int tid = t1;
        producer_threadid[tid] = tid;
        printf("Main: creating producer %d\n", tid);
        rc = pthread_create(&producer_threads[tid], NULL, producer,
                            (void *)&producer_threadid[tid]);
        if (rc)
        {
            printf("Error: Return code from pthread_create() is %d\n",
rc);
      has_ended = true;
            exit(-1);
        }
    }

    for (t2 = 0; t2 < CONSUMERS; t2++)
    {
        int tid = t2;
        consumer_threadid[tid] = tid;
        printf("Main: creating consumer %d\n", tid);
        rc = pthread_create(&consumer_threads[tid], NULL, consumer,
                            (void *)&consumer_threadid[tid]);
        if (rc)
        {
            printf("Error: Return code from pthread_create() is %d\n",
rc);
      has_ended = true;
            exit(-1);
        }
    }

  pthread_sigmask(SIG_SETMASK, &omask, NULL);
  signal(SIGINT, handle_sigint);
  for (int i = 0; i < PRODUCERS; ++i) {
    pthread_join(producer_threads[i], NULL);
  }
  for (int i = 0; i < CONSUMERS; ++i) {
    pthread_join(consumer_threads[i], NULL);
  }
  printf("Final Consumer Sum is: %d.\n", consumer_sum); // No need for
lock here as only thread left
    /*
                  some tips for this exercise:

                  1. you may want to handle SIGINT (ctrl-C) so that your
program
                                  can exit cleanly (by killing all
threads, or just calling
         exit)
```

```
                    1a. only one thread should handle the signal (POSIX
does not define
                                        *which* thread gets the signal), so
it's wise to mask out the
         signal on the worker threads (producer and consumer) and let the
main
         thread handle it
    */
}
```

Ex8

```
/*****************************************************************************
 * ex789-prod-con-threads.cpp
 * Producer-consumer synchronisation problem in C++
 *****************************************************************************/

#include <cstdio>
#include <cstdlib>
#include <signal.h>
#include <semaphore.h>
#include <sys/shm.h>
#include <sys/wait.h>

#define DO_LOGGING true

constexpr int PRODUCERS = 2;
constexpr int CONSUMERS = 1;

constexpr int MAX_BUFFER_LEN = 10;
constexpr int TOTAL_SHM_SIZE = MAX_BUFFER_LEN + 1 + 2;
constexpr int CONSUMER_SUM_OFFSET = 0;
constexpr int CURR_PROD_OFFSET = 1;
constexpr int CURR_CONS_OFFSET = 2;
constexpr int BUFFER_OFFSET = 3;
int (*shm)[TOTAL_SHM_SIZE] = NULL; // consumer_sum, curr_prod, curr_cons,
buffer baked into 1
int SHM_KEY = 420; // consumer_sum, curr_prod, curr_cons, buffer baked
into 1
/* !!!!!! KEY ASSUMPTION !!!!!! That the program will not run long enought
for curr_prod or curr_cons to overflow. */
constexpr int NUM_SEMS = 3;
sem_t (*sems)[NUM_SEMS]; // shm sem , prod sem and cons sem baked into 1
constexpr int SHM_SEM_OFFSET = 0;
constexpr int PROD_SEM_OFFSET = 1;
constexpr int CONS_SEM_OFFSET = 2;

volatile bool *has_ended = NULL;

void producer(int producer_id)
{
```

```
    while (!(*has_ended)) {
    int randInt = (rand() % 10) + 1; // random number between 1 to 10
    sem_wait(&((*sems)[PROD_SEM_OFFSET])); // Wait on buffer to not be
full
    sem_wait(&((*sems)[SHM_SEM_OFFSET])); // acquire SHM Semaphore
    (*shm)[BUFFER_OFFSET + ((*shm)[CURR_PROD_OFFSET] % MAX_BUFFER_LEN)] =
randInt;
    ++((*shm)[CURR_PROD_OFFSET]);
    #if DO_LOGGING
    printf("Producer %d adding %d to buffer, curr buffer size %d.\n",
producer_id, randInt, (*shm)[CURR_PROD_OFFSET] - (*shm)
[CURR_CONS_OFFSET]);
    #endif
    sem_post(&((*sems)[SHM_SEM_OFFSET])); // Release SHM Semaphore
    sem_post(&((*sems)[CONS_SEM_OFFSET])); // Post on cons sem as it is no
longer empty
  }
  printf("Producer %d exiting.\n", producer_id);
}

void consumer(int consumer_id)
{
    while (!(*has_ended)) {
    sem_wait(&((*sems)[CONS_SEM_OFFSET])); // Wait on buffer to not be
empty
    sem_wait(&((*sems)[SHM_SEM_OFFSET])); // acquire SHM Semaphore
    int consume_idx = (*shm)[CURR_CONS_OFFSET] % MAX_BUFFER_LEN;
    (*shm)[CONSUMER_SUM_OFFSET] += (*shm)[BUFFER_OFFSET + consume_idx];
    ++((*shm)[CURR_CONS_OFFSET]);
    #if DO_LOGGING
    printf("Consumer %d taking %d from buffer, curr buffer size %d. Sum
now %d.\n", consumer_id, (*shm)[BUFFER_OFFSET + consume_idx],
        (*shm)[CURR_PROD_OFFSET] - (*shm)[CURR_CONS_OFFSET], (*shm)
[CONSUMER_SUM_OFFSET]);
    #endif
    sem_post(&((*sems)[SHM_SEM_OFFSET])); // Release SHM Semaphore
    sem_post(&((*sems)[PROD_SEM_OFFSET])); // Post on prod sem as it is no
longer full
  }
  printf("Consumer %d exiting.\n", consumer_id);
}

void handle_sigint(int sig) {
  printf("\nCaught signal %d (SIGINT). Exiting gracefully...\n", sig);
  *has_ended = true;
}

bool shm_detach() {
  if (shmdt(shm) == -1) {
    printf("Failed to detach shared memory for prod cons.\n");
    return false;
  }

  if (shmdt((void *)has_ended) == -1) {
```

```
    printf("Failed to detach shared memory for sigint.\n");
    return false;
  }

  if (shmdt(sems) == -1) {
    printf("Failed to detach shared memory for sems.\n");
    return false;
  }

  return true;
}

int main(int argc, char *argv[])
{
  // Create sem shared memory
  int sem_shm_id = shmget(SHM_KEY++, sizeof(sem_t) * NUM_SEMS, IPC_CREAT |
0666);
  if (sem_shm_id < 0) {
      printf("Failed to create sem shared memory.\n");
      exit(-1);
  }
  sems = (sem_t (*)[NUM_SEMS])shmat(sem_shm_id, NULL, 0);
  printf("SHM shared memory segment created with ID: %d\n", sem_shm_id);

  // Create SHM Sem
  if (sem_init(&((*sems)[SHM_SEM_OFFSET]), 1, 1) == -1) {  // pshared = 1
for sharing between processes
    printf("Failed to initialize shm semaphore.\n");
    exit(-1);
  }

  // Create Prod Sem
  if (sem_init(&((*sems)[PROD_SEM_OFFSET]), 1, MAX_BUFFER_LEN) == -1) {
// pshared = 1 for sharing between processes
    printf("Failed to initialize prod semaphore.\n");
    exit(-1);
  }

  // Create Cons Sem
  if (sem_init(&((*sems)[CONS_SEM_OFFSET]), 1, 0) == -1) {  // pshared = 1
for sharing between processes
    printf("Failed to initialize cons semaphore.\n");
    exit(-1);
  }

  // Create prod cons shared memory
  int shm_id = shmget(SHM_KEY++, sizeof(int) * TOTAL_SHM_SIZE, IPC_CREAT |
0666);
  if (shm_id < 0) {
      printf("Failed to create prod cons shared memory.\n");
      exit(-1);
  }
  shm = (int (*)[TOTAL_SHM_SIZE])shmat(shm_id, NULL, 0);
  printf("Prod cons shared memory segment created with ID: %d\n", shm_id);
```

```
    // Initalise values for prod cons shm
    for (int i = 0; i < TOTAL_SHM_SIZE; ++i) {
      (*shm)[i] = 0;
    }

    // Create sigint shared memory
    int sigint_shm_id = shmget(SHM_KEY++, sizeof(bool), IPC_CREAT | 0666);
    if (sigint_shm_id < 0) {
        printf("Failed to create sigint shared memory.\n");
        exit(-1);
    }
    has_ended = (volatile bool *)shmat(sigint_shm_id, NULL, 0);
    printf("Shared memory segment created with ID: %d\n", sigint_shm_id);
    *has_ended = false; // initalise value of sigint check

      int producer_childid[PRODUCERS];
      int consumer_childid[CONSUMERS];

      for (int p = 0; p < PRODUCERS; p++)
      {
      int child_pid = fork();
      // Child
      if (child_pid == 0) {
        producer(p);
        exit(shm_detach() * -1);
      }

      // Handle Error
      if (child_pid < 0) {
              printf("Error: failed to create producer %d\n", p);
        exit(-1);
      }

      producer_childid[p] = child_pid;
          printf("Main: created producer %d with pid %d.\n", p, child_pid);
      }

      for (int c = 0; c < CONSUMERS; c++)
      {
      int child_pid = fork();
      // Child
      if (child_pid == 0) {
        consumer(c);
        exit(shm_detach() * -1);
      }

      // Handle Error
      if (child_pid < 0) {
              printf("Error: failed to create consumer %d\n", c);
        exit(-1);
      }

      consumer_childid[c] = child_pid;
          printf("Main: created consumer %d with pid %d.\n", c, child_pid);
```

```
    }

  signal(SIGINT, handle_sigint);
  for (int i = 0; i < PRODUCERS + CONSUMERS; ++i) {
    wait(NULL);
  }
  printf("Final Consumer Sum is: %d.\n", (*shm)[CONSUMER_SUM_OFFSET]); //
No need for sem here as only process left

  // int val;
  // sem_getvalue(&((*sems)[CONS_SEM_OFFSET]), &val);
  // printf("Final cons sem value is: %d.\n", val);
  // sem_getvalue(&((*sems)[PROD_SEM_OFFSET]), &val);
  // printf("Final prod sem value is: %d.\n", val);

  bool success = true;

  // Semaphore cleanup
  if (sem_destroy(&((*sems)[SHM_SEM_OFFSET])) == -1) {
    printf("Failed to destroy shm semaphore.\n");
    success = false;
  }
  printf("Destroyed shm semaphore.\n");

  if (sem_destroy(&((*sems)[PROD_SEM_OFFSET])) == -1) {
    printf("Failed to destroy prod semaphore.\n");
    success = false;
  }
  printf("Destroyed prod semaphore.\n");

  if (sem_destroy(&((*sems)[CONS_SEM_OFFSET])) == -1) {
    printf("Failed to destroy cons semaphore.\n");
    success = false;
  }
  printf("Destroyed cons semaphore.\n");

  // SHM cleanup
  success = shm_detach();

  if (shmctl(shm_id, IPC_RMID, NULL) == -1) {
    printf("Failed to delete prod cons shared memory.\n");
    success = false;
  }
  printf("Deleted prod cons shared memory.\n");


  if (shmctl(sigint_shm_id, IPC_RMID, NULL) == -1) {
    printf("Failed to delete sigint shared memory.\n");
    success = false;
  }
  printf("Deleted sigint shared memory.\n");

  if (shmctl(sem_shm_id, IPC_RMID, NULL) == -1) {
    printf("Failed to delete sems shared memory.\n");
```

```
    success = false;
  }
  printf("Deleted sems shared memory.\n");

  return success * -1;
    /*
                    some tips for this exercise:

                    1. you may want to handle SIGINT (ctrl-C) so that your
program
                                        can exit cleanly (by killing all
threads, or just calling
         exit)

                    1a. only one thread should handle the signal (POSIX
does not define
                                        *which* thread gets the signal), so
it's wise to mask out the
         signal on the worker threads (producer and consumer) and let the
main
         thread handle it
    */
}
```

Ex9 Threads Code

Modified to take in max iterations to run as the first command line argument

```
/**********************************************************************
 * ex789-prod-con-threads.cpp
 * Producer-consumer synchronisation problem in C++
 **********************************************************************/

#include <cstdio>
#include <cstdlib>
#include <pthread.h>
#include <signal.h>

#define DO_LOGGING false

constexpr int PRODUCERS = 2;
constexpr int CONSUMERS = 1;

constexpr int MAX_BUFFER_LEN = 10;
int consumer_sum = 0;
int producer_buffer[MAX_BUFFER_LEN];
/* !!!!!! KEY ASSUMPTION !!!!!! That the program will not run long enought
for curr_prod or curr_cons to overflow. */
int curr_prod = 0; // Tracks production idx
int curr_cons = 0; // Tracks consumption idx
pthread_mutex_t shared_var_lock = PTHREAD_MUTEX_INITIALIZER; // locks both
```

```
the producer buffer and consumer sum
pthread_cond_t wait_not_full_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t wait_not_empty_cond = PTHREAD_COND_INITIALIZER;

volatile bool has_ended = false;

// Testing stuff
int to_prod = 0, to_cons = 0;

void *producer(void *threadid)
{
  int tid = *((int *)threadid);
    while (!has_ended) {
    int randInt = (rand() % 10) + 1; // random number between 1 to 10

    pthread_mutex_lock(&shared_var_lock); // Get Mutex
    if (to_prod-- <= 0) {
      pthread_mutex_unlock(&shared_var_lock);
      break;
    }
    // Wait for there to be empty space on the buffer, check on has_ended
also to prevent infinite wait on termination and full buffer
    while ((curr_prod - curr_cons >= MAX_BUFFER_LEN) && !has_ended) {
      pthread_cond_wait(&wait_not_full_cond, &shared_var_lock);
    }
    producer_buffer[curr_prod % MAX_BUFFER_LEN] = randInt;
    ++curr_prod;
    #if DO_LOGGING
    printf("Producer %d adding %d to buffer, curr buffer size %d.\n", tid,
randInt, curr_prod - curr_cons);
    #endif
    pthread_mutex_unlock(&shared_var_lock); // Release Mutex
    pthread_cond_broadcast(&wait_not_empty_cond); // As we added to buffer
it is definitely not empty
  }
  printf("Producer %d exiting.\n", tid);
  return NULL;
}

void *consumer(void *threadid)
{
  int tid = *((int *)threadid);
    while (!has_ended) {
    pthread_mutex_lock(&shared_var_lock); // Get Mutex
    if (to_cons-- <= 0) {
      pthread_mutex_unlock(&shared_var_lock); // Release Mutex
      break;
    }
    // Wait for there to be value on the buffer, check on has_ended also
to prevent infinite wait on termination and empty buffer
    while ((curr_prod - curr_cons <= 0) && !has_ended) {
      pthread_cond_wait(&wait_not_empty_cond, &shared_var_lock);
    }
    int consume_idx = curr_cons % MAX_BUFFER_LEN;
```

```
    consumer_sum += producer_buffer[consume_idx];
    ++curr_cons;
    #if DO_LOGGING
    printf("Consumer %d taking %d from buffer, curr buffer size %d. Sum
now %d.\n", tid, producer_buffer[consume_idx], curr_prod - curr_cons,
consumer_sum);
    #endif
    pthread_mutex_unlock(&shared_var_lock); // Release Mutex
    pthread_cond_broadcast(&wait_not_full_cond); // As we took from buffer
it is definitely not full
  }
  printf("Consumer %d exiting.\n", tid);
  return NULL;
}

void handle_sigint(int sig) {
  printf("\nCaught signal %d (SIGINT). Exiting gracefully...\n", sig);
  has_ended = true;
}

int main(int argc, char *argv[])
{
  if (argc != 2) {
    printf("Please enter num of iterations to run.\n");
    return -1;
  }
  to_prod = atoi(argv[1]);
  to_cons = atoi(argv[1]);
    pthread_t producer_threads[PRODUCERS];
    pthread_t consumer_threads[CONSUMERS];
    int producer_threadid[PRODUCERS];
    int consumer_threadid[CONSUMERS];

    int rc;
    int t1, t2;

  sigset_t omask, mask;
  sigfillset(&mask);
  pthread_sigmask(SIG_SETMASK, &mask, &omask);

    for (t1 = 0; t1 < PRODUCERS; t1++)
    {
        int tid = t1;
        producer_threadid[tid] = tid;
        printf("Main: creating producer %d\n", tid);
        rc = pthread_create(&producer_threads[tid], NULL, producer,
                            (void *)&producer_threadid[tid]);
        if (rc)
        {
            printf("Error: Return code from pthread_create() is %d\n",
rc);
      has_ended = true;
            exit(-1);
        }
```

```
    }

    for (t2 = 0; t2 < CONSUMERS; t2++)
    {
        int tid = t2;
        consumer_threadid[tid] = tid;
        printf("Main: creating consumer %d\n", tid);
        rc = pthread_create(&consumer_threads[tid], NULL, consumer,
                            (void *)&consumer_threadid[tid]);
        if (rc)
        {
            printf("Error: Return code from pthread_create() is %d\n",
rc);
      has_ended = true;
            exit(-1);
        }
    }

  pthread_sigmask(SIG_SETMASK, &omask, NULL);
  signal(SIGINT, handle_sigint);
  for (int i = 0; i < PRODUCERS; ++i) {
    pthread_join(producer_threads[i], NULL);
  }
  for (int i = 0; i < CONSUMERS; ++i) {
    pthread_join(consumer_threads[i], NULL);
  }
  printf("Final Consumer Sum is: %d.\n", consumer_sum); // No need for
lock here as only thread left
    /*
                    some tips for this exercise:

                    1. you may want to handle SIGINT (ctrl-C) so that your
program
                                        can exit cleanly (by killing all
threads, or just calling
        exit)

                    1a. only one thread should handle the signal (POSIX
does not define
                                        *which* thread gets the signal), so
it's wise to mask out the
        signal on the worker threads (producer and consumer) and let the
main
        thread handle it
    */
}
```

Ex9 Threads Code

Modified to take in max iterations to run as the first command line argument

```cpp
/*****************************************************************
 * ex789-prod-con-threads.cpp
 * Producer-consumer synchronisation problem in C++
 *****************************************************************/

#include <cstdio>
#include <cstdlib>
#include <signal.h>
#include <semaphore.h>
#include <sys/shm.h>
#include <sys/wait.h>

#define DO_LOGGING false

constexpr int PRODUCERS = 2;
constexpr int CONSUMERS = 1;

constexpr int MAX_BUFFER_LEN = 10;
constexpr int TOTAL_SHM_SIZE = MAX_BUFFER_LEN + 1 + 2;
constexpr int CONSUMER_SUM_OFFSET = 0;
constexpr int CURR_PROD_OFFSET = 1;
constexpr int CURR_CONS_OFFSET = 2;
constexpr int BUFFER_OFFSET = 3;
int (*shm)[TOTAL_SHM_SIZE] = NULL; // consumer_sum, curr_prod, curr_cons,
buffer baked into 1
int SHM_KEY = 420; // consumer_sum, curr_prod, curr_cons, buffer baked
into 1
/* !!!!!! KEY ASSUMPTION !!!!!! That the program will not run long enought
for curr_prod or curr_cons to overflow. */
constexpr int NUM_SEMS = 3;
sem_t (*sems)[NUM_SEMS]; // shm sem , prod sem and cons sem baked into 1
constexpr int SHM_SEM_OFFSET = 0;
constexpr int PROD_SEM_OFFSET = 1;
constexpr int CONS_SEM_OFFSET = 2;

volatile bool *has_ended = NULL;

// Testing stuff
int (*todos)[2] = NULL; // to_prod, to_cons
constexpr int TO_PROD_OFFSET = 0;
constexpr int TO_CONS_OFFSET = 1;

void producer(int producer_id)
{
    while (!(*has_ended)) {
    int randInt = (rand() % 10) + 1; // random number between 1 to 10
    sem_wait(&((*sems)[PROD_SEM_OFFSET])); // Wait on buffer to not be
full
    sem_wait(&((*sems)[SHM_SEM_OFFSET])); // acquire SHM Semaphore
    if (((*todos)[TO_PROD_OFFSET])-- <= 0) {
      sem_post(&((*sems)[SHM_SEM_OFFSET]));
      sem_post(&((*sems)[CONS_SEM_OFFSET]));
      break;
```

```
    }

    (*shm)[BUFFER_OFFSET + ((*shm)[CURR_PROD_OFFSET] % MAX_BUFFER_LEN)] =
randInt;
    ++((*shm)[CURR_PROD_OFFSET]);
    #if DO_LOGGING
    printf("Producer %d adding %d to buffer, curr buffer size %d.\n",
producer_id, randInt, (*shm)[CURR_PROD_OFFSET] - (*shm)
[CURR_CONS_OFFSET]);
    #endif
    sem_post(&((*sems)[SHM_SEM_OFFSET])); // Release SHM Semaphore
    sem_post(&((*sems)[CONS_SEM_OFFSET])); // Post on cons sem as it is no
longer empty
  }
  printf("Producer %d exiting.\n", producer_id);
}

void consumer(int consumer_id)
{
    while (!(*has_ended)) {
    sem_wait(&((*sems)[CONS_SEM_OFFSET])); // Wait on buffer to not be
empty
    sem_wait(&((*sems)[SHM_SEM_OFFSET])); // acquire SHM Semaphore
    if (((*todos)[TO_CONS_OFFSET])-- <= 0) {
      sem_post(&((*sems)[SHM_SEM_OFFSET]));
      sem_post(&((*sems)[PROD_SEM_OFFSET]));
      break;
    }

    int consume_idx = (*shm)[CURR_CONS_OFFSET] % MAX_BUFFER_LEN;
    (*shm)[CONSUMER_SUM_OFFSET] += (*shm)[BUFFER_OFFSET + consume_idx];
    ++((*shm)[CURR_CONS_OFFSET]);
    #if DO_LOGGING
    printf("Consumer %d taking %d from buffer, curr buffer size %d. Sum
now %d.\n", consumer_id, (*shm)[BUFFER_OFFSET + consume_idx],
        (*shm)[CURR_PROD_OFFSET] - (*shm)[CURR_CONS_OFFSET], (*shm)
[CONSUMER_SUM_OFFSET]);
    #endif
    sem_post(&((*sems)[SHM_SEM_OFFSET])); // Release SHM Semaphore
    sem_post(&((*sems)[PROD_SEM_OFFSET])); // Post on prod sem as it is no
longer full
  }
  printf("Consumer %d exiting.\n", consumer_id);
}

void handle_sigint(int sig) {
  printf("\nCaught signal %d (SIGINT). Exiting gracefully...\n", sig);
  *has_ended = true;
}

bool shm_detach() {
  if (shmdt(shm) == -1) {
    printf("Failed to detach shared memory for prod cons.\n");
    return false;
```

```
  }

  if (shmdt(sems) == -1) {
    printf("Failed to detach shared memory for sems.\n");
    return false;
  }

  if (shmdt(todos) == -1) {
    printf("Failed to detach shared memory for todos.\n");
    return false;
  }

  return true;
}

int main(int argc, char *argv[])
{
  if (argc != 2) {
    printf("Please enter num of iterations to run.\n");
    return -1;
  }

  // Create sem shared memory
  int sem_shm_id = shmget(SHM_KEY++, sizeof(sem_t) * NUM_SEMS, IPC_CREAT |
0666);
  if (sem_shm_id < 0) {
      printf("Failed to create sem shared memory.\n");
      exit(-1);
  }
  sems = (sem_t (*)[NUM_SEMS])shmat(sem_shm_id, NULL, 0);
  printf("SHM shared memory segment created with ID: %d\n", sem_shm_id);

  // Create SHM Sem
  if (sem_init(&((*sems)[SHM_SEM_OFFSET]), 1, 1) == -1) {  // pshared = 1
for sharing between processes
    printf("Failed to initialize shm semaphore.\n");
    exit(-1);
  }

  // Create Prod Sem
  if (sem_init(&((*sems)[PROD_SEM_OFFSET]), 1, MAX_BUFFER_LEN) == -1) {
// pshared = 1 for sharing between processes
    printf("Failed to initialize prod semaphore.\n");
    exit(-1);
  }

  // Create Cons Sem
  if (sem_init(&((*sems)[CONS_SEM_OFFSET]), 1, 0) == -1) {  // pshared = 1
for sharing between processes
    printf("Failed to initialize cons semaphore.\n");
    exit(-1);
  }

  // Create prod cons shared memory
```

```
  int shm_id = shmget(SHM_KEY++, sizeof(int) * TOTAL_SHM_SIZE, IPC_CREAT |
0666);
  if (shm_id < 0) {
      printf("Failed to create prod cons shared memory.\n");
      exit(-1);
  }
  shm = (int (*)[TOTAL_SHM_SIZE])shmat(shm_id, NULL, 0);
  printf("Prod cons shared memory segment created with ID: %d\n", shm_id);
  // Initalise values for prod cons shm
  for (int i = 0; i < TOTAL_SHM_SIZE; ++i) {
    (*shm)[i] = 0;
  }

  // Create sigint shared memory
  int sigint_shm_id = shmget(SHM_KEY++, sizeof(bool), IPC_CREAT | 0666);
  if (sigint_shm_id < 0) {
      printf("Failed to create sigint shared memory.\n");
      exit(-1);
  }
  has_ended = (volatile bool *)shmat(sigint_shm_id, NULL, 0);
  printf("Shared memory segment created with ID: %d\n", sigint_shm_id);
  *has_ended = false; // initalise value of sigint check
  signal(SIGINT, handle_sigint); // set signal handler


  // Create testing shared memory
  int todo_shm_id = shmget(SHM_KEY++, sizeof(int) * 2, IPC_CREAT | 0666);
  if (todo_shm_id < 0) {
      printf("Failed to create sigint shared memory.\n");
      exit(-1);
  }
  todos = (int (*)[2])shmat(todo_shm_id, NULL, 0);
  printf("Shared memory segment created with ID: %d\n", todo_shm_id);
  // initalise value of todos
  (*todos)[TO_PROD_OFFSET] = atoi(argv[1]);
  (*todos)[TO_CONS_OFFSET] = atoi(argv[1]);

  int producer_childid[PRODUCERS];
  int consumer_childid[CONSUMERS];

  for (int p = 0; p < PRODUCERS; p++)
  {
    int child_pid = fork();
    // Child
    if (child_pid == 0) {
      producer(p);
      exit(shm_detach() * -1);
    }

    // Handle Error
    if (child_pid < 0) {
      printf("Error: failed to create producer %d\n", p);
      exit(-1);
    }
```

```
    producer_childid[p] = child_pid;
    printf("Main: created producer %d with pid %d.\n", p, child_pid);
  }

  for (int c = 0; c < CONSUMERS; c++)
  {
    int child_pid = fork();
    // Child
    if (child_pid == 0) {
      consumer(c);
      exit(shm_detach() * -1);
    }

    // Handle Error
    if (child_pid < 0) {
      printf("Error: failed to create consumer %d\n", c);
      exit(-1);
    }

    consumer_childid[c] = child_pid;
    printf("Main: created consumer %d with pid %d.\n", c, child_pid);
  }

  for (int i = 0; i < PRODUCERS + CONSUMERS; ++i) {
    wait(NULL);
  }
  printf("Final Consumer Sum is: %d.\n", (*shm)[CONSUMER_SUM_OFFSET]); //
No need for sem here as only process left

  // int val;
  // sem_getvalue(&((*sems)[CONS_SEM_OFFSET]), &val);
  // printf("Final cons sem value is: %d.\n", val);
  // sem_getvalue(&((*sems)[PROD_SEM_OFFSET]), &val);
  // printf("Final prod sem value is: %d.\n", val);

  bool success = true;

  // Semaphore cleanup
  if (sem_destroy(&((*sems)[SHM_SEM_OFFSET])) == -1) {
    printf("Failed to destroy shm semaphore.\n");
    success = false;
  }
  printf("Destroyed shm semaphore.\n");

  if (sem_destroy(&((*sems)[PROD_SEM_OFFSET])) == -1) {
    printf("Failed to destroy prod semaphore.\n");
    success = false;
  }
  printf("Destroyed prod semaphore.\n");

  if (sem_destroy(&((*sems)[CONS_SEM_OFFSET])) == -1) {
    printf("Failed to destroy cons semaphore.\n");
    success = false;
```

```
  }
  printf("Destroyed cons semaphore.\n");

  // SHM cleanup
  success = shm_detach();

  if (shmctl(shm_id, IPC_RMID, NULL) == -1) {
    printf("Failed to delete prod cons shared memory.\n");
    success = false;
  }
  printf("Deleted prod cons shared memory.\n");

  if (shmctl(sem_shm_id, IPC_RMID, NULL) == -1) {
    printf("Failed to delete sems shared memory.\n");
    success = false;
  }
  printf("Deleted sems shared memory.\n");

  // Cleanup sigint shm
  if (shmdt((void *)has_ended) == -1) {
    printf("Failed to detach shared memory for sigint.\n");
  }
  printf("Detached sigint shared memory.\n");

  if (shmctl(sigint_shm_id, IPC_RMID, NULL) == -1) {
    printf("Failed to delete sigint shared memory.\n");
  }
  printf("Deleted sigint shared memory.\n");

  // Cleanup todo shm
  if (shmctl(todo_shm_id, IPC_RMID, NULL) == -1) {
    printf("Failed to delete todos shared memory.\n");
  }
  printf("Deleted todos shared memory.\n");
    /*
                    some tips for this exercise:

                    1. you may want to handle SIGINT (ctrl-C) so that your
program
                                        can exit cleanly (by killing all
threads, or just calling
          exit)

                    1a. only one thread should handle the signal (POSIX
does not define
                                        *which* thread gets the signal), so
it's wise to mask out the
          signal on the worker threads (producer and consumer) and let the
main
          thread handle it
     */
}
```

Ex9 Benchmark Code

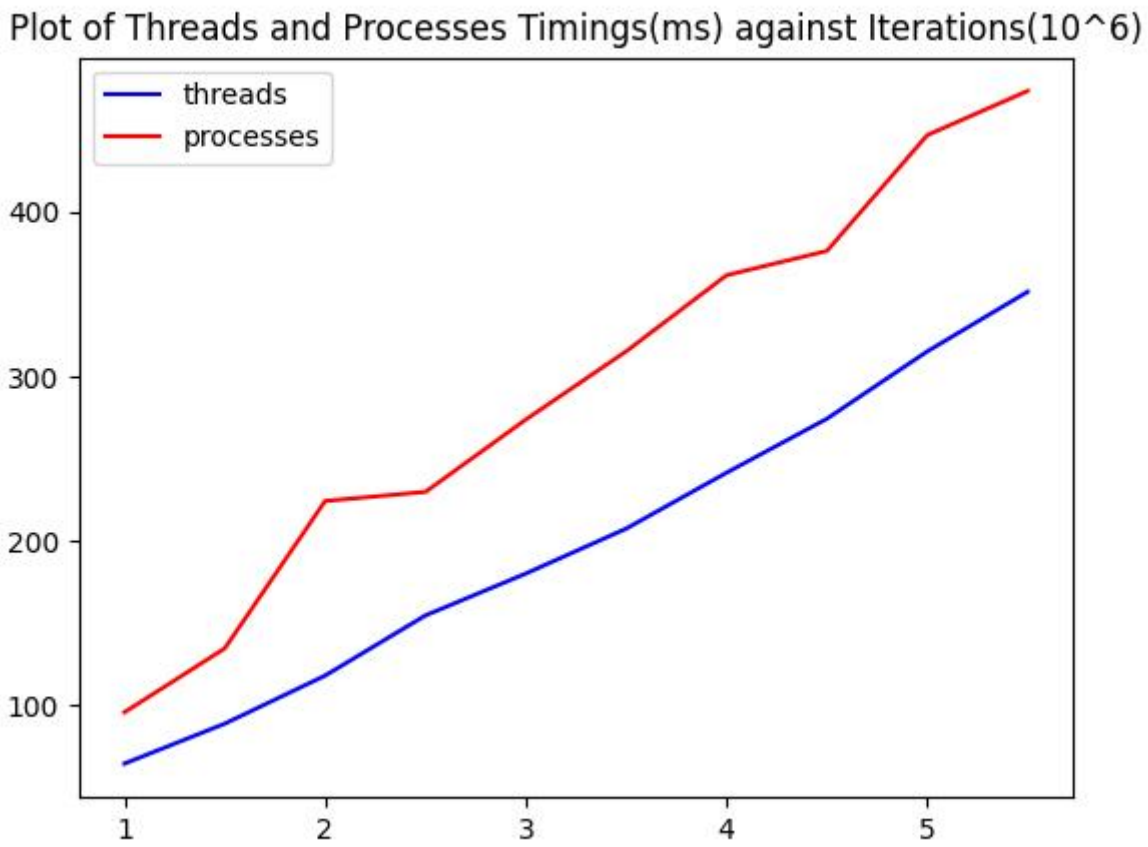
```
#include <cstdio>
#include <cstdlib>
#include <sys/wait.h>
#include <chrono>

constexpr int SAMPLES = 10;
constexpr int SAMPLE_REPEAT = 10;
constexpr int START_NUM_ITER = 100000;
constexpr int SAMPLE_DISTANCE = 50000;
constexpr int END_NUM_ITER = (SAMPLES * SAMPLE_DISTANCE) + START_NUM_ITER;

int main(int argc, char *argv[]) {
  // threads
  FILE *threads_log = fopen("./threads_timings.csv", "w+");
  fprintf(threads_log, "Iterations, Duration\n");
  for (int i = START_NUM_ITER; i < END_NUM_ITER; i += SAMPLE_DISTANCE) {
    long total_time = 0;
    for (int j = 0; j < SAMPLE_REPEAT; ++j) {
      auto start_time = std::chrono::system_clock::now();
      if (fork() == 0) {
        char str_number[20];
        snprintf(str_number, sizeof(str_number), "%d", i);
        execl("./ex9-prod-con-threads", "ex9-prod-con-threads",
str_number, (char *)NULL);
      }
      wait(NULL);
      total_time += (std::chrono::system_clock::now() -
start_time).count();
    }
    fprintf(threads_log, "%d, %ld\n", i, long(total_time /
SAMPLE_REPEAT));
  }
  fclose(threads_log);

  // processes
  FILE *processes_log = fopen("./processes_timings.csv", "w+");
  fprintf(processes_log, "Iterations, Duration\n");
  for (int i = START_NUM_ITER; i < END_NUM_ITER; i += SAMPLE_DISTANCE) {
    auto start_time = std::chrono::system_clock::now();
    if (fork() == 0) {
      char str_number[20];
      snprintf(str_number, sizeof(str_number), "%d", i);
      execl("./ex9-prod-con-processes", "ex9-prod-con-processes",
str_number, (char *)NULL);
    }
    wait(NULL);
    auto interval = std::chrono::system_clock::now() - start_time;
    fprintf(processes_log, "%d, %ld\n", i, interval.count());
  }
```

```
  fclose(processes_log);
}
```

Ex9 Plotting Code

```
import matplotlib.pyplot as plt
import csv

NANOSECONDS_TO_MICROSECONDS = 1 / 1000000
ITERATIONS_SCALER = 1 / 100000

if __name__ == "__main__":
  threads_timings = []
  processes_timings = []
  iterations = []

  with open("./threads_timings.csv", "+r") as threadsFile:
    reader = csv.reader(threadsFile)
    next(reader) # read header row
    for row in reader:
      iterations.append(int(row[0]) * ITERATIONS_SCALER)
      threads_timings.append(int(row[1]) * NANOSECONDS_TO_MICROSECONDS)

  with open("./processes_timings.csv", "+r") as processesFile:
    reader = csv.reader(processesFile)
    next(reader) # read header row
    for idx, row in enumerate(reader):
      assert iterations[idx] == int(row[0]) * ITERATIONS_SCALER,
"Iterations should be the same in threads and processes"
      processes_timings.append(int(row[1]) * NANOSECONDS_TO_MICROSECONDS)

  plt.plot(iterations, threads_timings, color = "blue", label = "threads")
  plt.plot(iterations, processes_timings, color = "red", label =
"processes")
  plt.legend()
  plt.title("Plot of Threads and Processes Timings(ms) against
Iterations(10^6)")
  plt.savefig("./comparison.jpg")
  plt.show()
```

Ex9 Writeup



I compared threads versus processes at 10 different iteration points, each sampled 10 times with the mean taken. The iterations begin at 100k and increase by 50k per sample. Based on the graph above, we can see that both threads and processes show a linear increase in runtime as the number of iterations increases. We see that the time taken for processes is in absolute terms greater than that for threads. Beyond that, the rate of increase in time taken is greater for the processes than the threads. This is the expected result as context switching for processes carries a greater overhead than switching between threads. In addition, the CPU used on node 11 is the i7-7700 which supports hyperthreading, the simultaneous running of 2 threads on 1 core, which further increases the threads program's lead in runtime. As our operation per cycle is relatively simple for both the producers and consumer, it is logical to reason that context switching takes up a larger proportion of our runtime compared to the actual operations. As such, the processes has a faster growth rate for timing as compared to threads due to the context switching overhead.