Homework 4 - Operating Systems (ICS431)

Alfaifi, Ammar

1 Guess the letter

This program uses `shmget` to get a shared memory segment, `shmat` to attach the segment to our data space, and `shmctl` with `IPC_RMID` to remove the shared memory segment.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>
#include <unistd.h>

#define SIZE 1 // size of the shared memory segment
#define MAX_GUESSES 5

int main() {
  int shmid;
  char *shmptr, guess;
  key_t key = IPC_PRIVATE;
  pid_t pid;
  int guessCount = 0;

  // Create the shared memory segment,
  // read and write permissions for user, group, and others.
  if ((shmid = shmget(key, SIZE, IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
  }

  // Attach the shared memory segment to our data space
  if ((shmptr = shmat(shmid, NULL, 0)) == (char *)-1) {
    perror("shmat");
    exit(1);
  }

  // Fork a child process
  pid = fork();
  if (pid < 0) {
    fprintf(stderr, "Fork Failed");
    return 1;
  } else if (pid > 0) { // Parent Process (Server)
    // Put a random letter in shared memory
    srand(time(0));
    *shmptr = 'A' + (rand() % 26);

    // Wait for the client to finish guessing
    wait(NULL);

    // Show the correct answer if client hasn't guessed correctly
    printf("The correct letter was %c\n", *shmptr);
  } else { // Child Process (Client)
    do {
      printf("Enter a letter to guess: ");
      scanf(" %c",
            &guess); // A space before %c is required to skip any leftover '\n'

      if (guess > *shmptr)
        printf("Your guess is too high.\n");
      else if (guess < *shmptr)
        printf("Your guess is too low.\n");
      else {
        printf("Congrats! You guessed the letter.\n");
        exit(0);
      }

      guessCount++;
```

```
    } while (guessCount < MAX_GUESSES);
  }

  // Detach from the shared memory segment
  if (shmdt(shmptr) == -1) {
    perror("shmdt");
    exit(1);
  }

  // Delete the shared memory segment
  if (shmctl(shmid, IPC_RMID, NULL) == -1) {
    perror("shmctl");
    exit(1);
  }

  return 0;
}
```

2 Two-threaded Program

2.1 Using mutex

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 5

int arr[SIZE];
pthread_mutex_t lock;
int idx = 0;
int produced = 0;

void *produce(void *arg) {
  srand(time(0));
  while (idx < SIZE) {
    if (produced == 0) {
      pthread_mutex_lock(&lock);
      int num = rand() % 100;
      arr[idx++] = num;
      printf("Produced: %d\n", num);
      produced = 1;
      pthread_mutex_unlock(&lock);
    }
  }
  return NULL;
}

void *consume(void *arg) {
  int curr_index = 0;
  while (curr_index < SIZE) {
    pthread_mutex_lock(&lock);
    if (curr_index < idx && produced == 1) {
      printf("Consumed: %d\n", arr[curr_index++]);
      produced = 0;
    }
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}

int main() {
  pthread_t producer, consumer;
  pthread_mutex_init(&lock, NULL);

  pthread_create(&producer, NULL, &produce, NULL);
  pthread_create(&consumer, NULL, &consume, NULL);

  pthread_join(producer, NULL);
  pthread_join(consumer, NULL);

  pthread_mutex_destroy(&lock);

  return 0;
}
```

2.2 Using semaphore

Some of used methods here are actually deprecated, but it worked on my machine.

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 5

int arr[SIZE];
sem_t empty;
sem_t full;
int idx = 0;
int produced = 0;

void *produce(void *arg) {
  srand(time(0));
  while (idx < SIZE) {
    if (produced == 0) {
      int num = rand() % 100;
      sem_wait(&empty);
      arr[idx++] = num;
      printf("Produced: %d\n", num);
      produced = 1;
      sem_post(&full);
    }
  }
  return NULL;
}

void *consume(void *arg) {
  int curr_index = 0;
  while (curr_index < SIZE) {
    if (produced == 1) {
      sem_wait(&full);
      printf("Consumed: %d\n", arr[curr_index++]);
      produced = 0;
      sem_post(&empty);
    }
  }
  return NULL;
}

int main() {
  pthread_t producer, consumer;

  sem_init(&empty, 0, SIZE);
  sem_init(&full, 0, 0);

  pthread_create(&producer, NULL, &produce, NULL);
  pthread_create(&consumer, NULL, &consume, NULL);

  pthread_join(producer, NULL);
  pthread_join(consumer, NULL);

  sem_destroy(&empty);
  sem_destroy(&full);

  return 0;
}
```

Theoretical Part

1 Code and Data Separation

We consider a system where a program can be partitioned into two components: code and data.

Advantages

- **Enhanced Security:** With the separation of instruction and data spaces, with the instruction space as read-only, this system can effectively thwart common security exploits by preventing unauthorized code modification.
- **Program Sharing:** As the instruction base-limit register pair is read-only, it allows safe sharing of the same instructions across multiple users, leading to efficient memory utilization for commonly used libraries or programs.
- **Protection Against Programming Errors:** Certain programming errors, such as accidental code overwrite due to buffer overflows, could be mitigated in this system.
- **Easier Memory Management:** The distinct separation between code and data could facilitate easier memory management by the operating system.

Disadvantages

- **Less Flexibility:** Certain programming paradigms, such as Just-in-Time (JIT) compilation or certain types of functional or logic programming, may blur the distinction between code and data. This could potentially restrict the effective execution of these types of programs.
- **Increased Hardware Complexity:** Implementing this scheme at the hardware level would likely augment the complexity of the CPU architecture, leading to higher costs and more potential points of failure.
- **Performance Overhead:** Managing separate spaces for code and data might introduce performance overhead, as the CPU would have to context-switch between handling code and data.
- **Resource Utilization:** This scheme might lead to inefficient memory usage if the allocations for code and data cannot be dynamically adjusted, potentially leading to wasted memory space.

2 Page Faults

The page reference string is: 7,2,3,1,2,5,3,4,6,7,7,1,0,5,4,6,2,3,0,1.

a. The total number of page faults with the LRU algorithm is 15

Page Request	7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
Frame Contents	[7,-,-]	[7,2,-]	[7,2,3]	[1,2,3]	[1,2,3]	[1,5,3]	[1,5,3]	[4,5,3]	[4,5,6]	[4,7,6]	[4,7,6]	[4,7,1]	[0,7,1]	[0,5,1]	[0,5,4]	[0,5,6]	[2,5,6]	[2,3,6]	[2,3,0]	[2,3,1]

b. The total number of page faults with the FIFO algorithm is 18

Page Request	7*	2*	3*	1*	2*	5*	3	4*	6*	7*	7	1*	0*	5*	4*	6*	2*	3*	0*	1*
Frame Contents	[7,-,-]	[7,2,-]	[7,2,3]	[1,2,3]	[1,2,3]	[1,5,3]	[1,5,3]	[1,5,4]	[6,5,4]	[7,6,4]	[7,6,4]	[1,6,4]	[0,1,6]	[5,0,1]	[4,5,0]	[6,4,5]	[2,6,4]	[3,2,6]	[0,3,2]	[1,0,3]

c. The total number of page faults with the Optimal algorithm is 9

Page Request	7*	2*	3*	1*	2	5*	3	4*	6*	7	7	1*	0*	5	4*	6	2	3*	0	1
Frame Contents	[7,-,-]	[7,2,-]	[7,2,3]	[1,2,3]	[1,2,3]	[1,5,3]	[1,5,3]	[1,5,4]	[6,5,4]	[7,6,4]	[7,6,4]	[1,7,4]	[0,1,7]	[0,1,7]	[4,0,1]	[4,0,1]	[4,0,1]	[4,0,3]	[4,0,3]	[4,0,3]

3 Dissuasion about disk-scheduling algorithms:

a The assertion is true because many disk-scheduling algorithms prioritize certain requests over others based on specific conditions. For instance, in Shortest Seek Time First (SSTF), requests with the least seek time from the current head position are processed first, which could lead to some requests with longer seek times never being processed (starvation).

b To modify algorithms like SCAN to ensure fairness, one could implement an age factor. If a request has been waiting for a certain amount of time, it could be promoted in priority, ensuring it will eventually be serviced. This method still retains the advantage of SCAN in reducing arm movement while ensuring that no request will be indefinitely postponed.

c Fairness is a crucial goal in multi-user systems because many users are relying on the same resources to accomplish their tasks. If the system is not fair, then some users may experience significant delays in completing their tasks while others may monopolize resources.

d There may be several instances where the operating system must be unfair in serving I/O requests:

(a) *Real-Time Systems*: In real-time systems, certain tasks have strict timing constraints.

(b) *Critical System Processes*: Certain system processes are critical for the overall functioning of the operating system.

4 Automatic deletion vs. manual deletion:

1. Automatic Deletion of Files

   - *Advantage*: This approach can help in maintaining privacy and security. It ensures that no sensitive data is left on a computer after a user is done with their session, which is particularly beneficial in shared or public computer systems.
   - *Disadvantage*: A major drawback is the risk of losing important data if the user forgets to explicitly request their files to be kept. Additionally, this might cause inconvenience for users who frequently need to use the same files across different sessions.

2. Keeping Files Unless Explicitly Deleted

   - *Advantage*: This approach is user-friendly, particularly for those who wish to access the same files over multiple sessions or forget to save their files. It reduces the risk of data loss due to forgetting to save files.
   - *Disadvantage*: However, it may lead to the accumulation of unnecessary files, consuming storage space. Furthermore, there may be privacy or security risks if sensitive files are left accessible after the user session ends.

5 Caches in OS

Caches in an Operating System (OS) act as intermediary stores between the high-speed processor and the relatively slower main memory. They help in reducing the average time to access data, thus improving system performance. The fundamental principles for their functionality are the spatial and temporal locality of reference:

- **Spatial Locality**: If a data item is referenced, it is likely that nearby data will be accessed soon.
- **Temporal Locality**: If a data item is referenced, it is likely that the same data will be accessed again soon.

Despite the clear benefits of cache memory, there are several reasons why systems do not simply use larger caches:

1. **Cost**: Cache memory is more expensive to produce than main memory or disk storage.
2. **Diminishing Returns**: As cache size increases, the hit rate improvements become less significant. This is known as the principle of locality. Beyond a certain point, the cost of adding more cache memory outweighs the performance benefit.
3. **Increased Latency**: Larger caches can lead to longer data access times, offsetting the speed advantage.
4. **Complexity**: Managing larger caches can add to the system's complexity, requiring more sophisticated algorithms and possibly slowing down the system.

6 About RAID

Under specific circumstances, RAID 1 can achieve better read performance than RAID 0. The reason is that RAID 1 can service multiple read requests simultaneously from different disks, assuming the requests are not for the same data. Thus, if there are many independent and concurrent read requests, RAID 1 may outperform RAID 0. However, this also heavily depends on the RAID controller's ability to handle multiple concurrent requests.