운영체제, 과제 1

#1

주요 수정 항목

```
if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *
{
  perror("In accept");
  exit(EXIT_FAILURE);
}
else
{
  pid_t pid;
  pid = fork();
  if (pid == 0)
  {
    char buffer[30000] = {0};
    valread = read(new_socket, buffer, 30000);
    printf("%s\n", buffer);
    // uncomment following line and connect many clients
    sleep(5);
    write(new_socket, hello, strlen(hello));
    printf("-------------Hello message sent---------------");
    close(new_socket);
    exit(0);
  }
  else
  {
    close(new_socket);
    printf("forked: %d", pid);
  }
}
```

HTTP 요청을 받아오면 새 자식 프로세스를 생성합니다. 자식 프로세스는 HTTP 요청을 처리하여 `hello` 변수에 담긴 값으로 응답하고, 원본 프로세스는 계속해서 새 요청을 대기합니다.

HTTP 요청에 대응하기 위해 대기하는 것은 원본 프로세스이므로, 원본 프로세스는 `sleep(5)` 의 영향을 받지 않습니다.

#2

주요 수정 항목

```
*new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&ad
if (*new_socket < 0)
```

```
{
  perror("In accept");
  exit(EXIT_FAILURE);
}

char buffer[30000] = {0};
valread = read(new_socket, buffer, 30000);
printf("%s\n", buffer);

pthread_t tid;
if (pthread_create(&tid, NULL, handler, (void *)new_socket) < 0)
{
  free(new_socket);
}
```

HTTP 요청을 받아오면 새 스레드를 생성합니다. 새로 생성된 스레드는 `handler` 함수를 실행합니다. `handler` 함수에서 HTTP 요청을 처리할 것을 기대할 수 있습니다.

```
void *handler(void *arg)
{
  int socket = *((int *)arg);
  sleep(5);
  write(socket, hello, strlen(hello));
  close(socket);
  free(arg);
  printf("-------------Hello message sent---------------");
  pthread_exit(0);
}
```

`handler` 함수는 생성 과정에서 HTTP 연결이 수립된 소켓의 파일 디스크립터를 전달받습니다. 전달받은 디스크립터를 사용하여 `hello` 변수에 담긴 값으로 HTTP 요청에 응답합니다.

```
close(socket);
free(arg);
```

`handler` 함수에는 소켓을 닫는 것 뿐 아니라 매개 변수 원본을 메모리에서 할당 해제하는 내용이 포함되어있습니다. 이것은 `accesses.py` 를 이용해 동시 접속을 발생시켰을 때, 연결이 제대로 닫히지 않는 문제가 파악되어 강제로 연결을 해제하기 위해 사용했습니다.

다른 좋은 방법을 찾아내고자 여러 구현을 실험해 보았으나, 마땅한 해결책을 찾을 수 없었습니다. 대표적으로는 다음 사례가 있었습니다.

- 메인 스레드에서 소켓을 닫지 않음
  - 연결이 제대로 종료되지 않음
- 메인 스레드에서도 소켓을 닫음

    - `handler` 스레드에서도 소켓을 사용하지 못함
- 메인 스레드에서 `handler` 스레드의 종료를 체크하고 소켓을 닫음
    - Blocking: `handler` 스레드가 종료될 때까지 메인 스레드가 더 이상 진행하지 않음

#3

주요 수정 항목

```
pthread_t request_thread, response_thread, optimizing_queue_thread;
pthread_create(&request_thread, NULL, handle_requests, &server_fd);
pthread_create(&response_thread, NULL, respond_requests, NULL);
// pthread_create(&optimizing_queue_thread, NULL, optimize_logic_queue, NULL)

pthread_join(request_thread, NULL);
pthread_join(response_thread, NULL);
// pthread_join(optimizing_queue_thread, NULL);
```

편의상 메인 스레드에서 HTTP 요청의 수신부와 응답부를 분리했습니다.

다양한 구현을 시도하면서 메인 스레드에 구현된 HTTP 요청 수신부가 코드 작성 과정에서 다소 혼잡했기 때문에 이를 해결하기 위해서 분리했습니다.

원래는 각 큐를 정리하고 로드밸런싱하는 작업(`optimizing_queue_thread`)도 별도의 스레드로 동작시키려고 했으나, 동시성 관리가 제대로 되지 않아 `response_thread`가 작업 후에 호출하는 것으로 수정하였습니다.

큐의 구현

```
typedef struct Node;
typedef struct Deque;
void init_deque(Deque *deque);
void is_empty(Deque *deque);
void push_back(Deque *deque, void *value)
void push_front(Deque *deque, void *value)
Node *get_node_at(Deque *deque, int index)
void *get_at(Deque *deque, int index)
void *pop_at(Deque *deque, int index)
int get_size(Deque *deque)
void *pop_front(Deque *deque)
void *get_back(Deque *deque)
void *pop_back(Deque *deque)
Node *get_next(Deque *deque, Node *node)
Node *get_prev(Deque *deque, Node *node)
```

몇가지 구현을 실험하고 검토한 결과, 목표를 달성하는데 사용할 큐는 덱을 구현하여 사용하게 되었습니다.

- 2차원 배열 $n \times 10$ 을 <로직 큐> $\times$ <메시지 큐>로 사용

    - pop 처리가 비효율적임
    - 결국 n의 크기도 정의해야함
- 큐를 구현하여 `Queue<Queue<>>` 와 비슷하게 사용
    - 큐 내의 각 원소에 대해, 원소의 앞에 위치한 원소를 참조하는데 어려움이 있음
- 덱을 구현하여 `Deque<Deque<>>` 와 비슷하게 사용

HTTP 요청 수신부

```
void *handle_requests(void *arg)
{
  int server_fd = *(int *)arg;
  struct sockaddr_in client_addr;
  socklen_t client_len = sizeof(client_addr);

  int i = 0;
  while (1)
  {
    // printf("\n+++++++ Waiting for new connection ++++++++\n\n");

    int *client_sock = malloc(sizeof(int));
    *client_sock = accept(server_fd, (struct sockaddr *)&client_addr, &client
    if (*client_sock < 0)
    {
      perror("accept failed");
      continue;
    }
    printf("%d received\n", i++);
    Deque *now;
    int i = 0, min_size = (int)1e6, index = 0;
    while (1)
    {
      if (get_size(&logic_queue) <= i)
      {
        break;
      }
      now = get_at(&logic_queue, i);
      int now_size = get_size(now);
      if (min_size > now_size)
      {
        min_size = now_size;
        index = i;
      }
      i++;
    }
    push_back(get_at(&logic_queue, index), client_sock);
  }
```

```
  return NULL;
}
```

HTTP 요청을 수신하면 다음 작업을 순차적으로 실행합니다.

1. `logic_queue`에서 관리하고 있는 메시지 큐 중 가장 많이 비어있는 큐를 찾습니다.
2. 찾은 큐에 소켓의 파일 디스크립터를 삽입합니다.

HTTP 요청 응답부

```
#define REQUEST_CHAR_BUFFER_SIZE 1024
// Function to respond to requests
void *respond_requests(void *arg)
{
  // get scheduling target
  int next = 0;
  while (1)
  {
    if (get_size(&logic_queue) <= 0)
    {
      continue; // wait unitl load
    }
    Deque *now_queue = (Deque *)get_at(&logic_queue, next % get_size(&logic_q
    if (get_size(now_queue) <= 0)
    {
      continue;
      // will be removed at optimizer
    }
    next = (next + 1) % get_size(&logic_queue);

    int client_sock = *((int *)pop_front(now_queue));
    printf("client sock %d (%d) at queue %d / %d\n", client_sock, get_size(no

    // Handle the request
    char buffer[REQUEST_CHAR_BUFFER_SIZE];
    ssize_t n = read(client_sock, buffer, sizeof(buffer) - 1);
    if (n < 0)
    {
      perror("read failed");
      close(client_sock);
      continue;
    }
    buffer[n] = '\0';
    // printf("Received request:\n%s\n", buffer);

    // Send response
    const char *response = hello;
```

```
    // sleep(1);
    write(client_sock, response, strlen(response));
    printf("processed: %d\n", next);

    close(client_sock);
    printf("%d\n", client_sock);
    optimize_logic_queue(NULL);
  }

  return NULL;
}
```

`response_requests`는 `logic_queue`를 순회하면서 `logic_queue`가 담고 있는 각 큐에 새 소켓 파일 디스크립터가 삽입되는지 확인합니다.

만약 확인하고 있는 큐에 소켓 파일 디스크립터가 존재한다면 다음 작업을 순차적으로 실행합니다.

1. 확인하고 있는 큐에서 소켓 파일 디스크립터를 pop
2. pop한 소켓 파일 디스크립터를 이용해 HTTP 응답 전송
3. pop한 소켓 파일 디스크립터를 닫고 최적화 작업(`optimize_logic_queue`) 실행

로직 큐 최적화 부

```
#define LOGIC_QUEUE_APPEND_LIMIT 8
#define LOGIC_QUEUE_REDUCE_LIMIT 0
void *optimize_logic_queue(void *arg)
{
  // while(1) {
  int q_index = 0;
  while (1)
  {
    if (q_index >= get_size(&logic_queue))
    {
      break;
    }
    Deque *each_socket_queue, *new_sock;
    each_socket_queue = (Deque *)get_at(&logic_queue, q_index);

    if (get_size(each_socket_queue) >= LOGIC_QUEUE_APPEND_LIMIT)
    {
      new_sock = append_logic_queue();
      for (int i = 0; i < get_size(each_socket_queue) / 2; i++)
      {
        int back = *((int *)pop_back(each_socket_queue));
        push_back(new_sock, &back);
      }
    }
    if (get_size(each_socket_queue) <= LOGIC_QUEUE_REDUCE_LIMIT)
```

```
    {
      if (q_index != 0)
      {
        // 여기서만 pop함을 보장해야함
        pop_at(&logic_queue, q_index);
      }
    }
    q_index++;
  }
  return NULL;
  // }
}
```

로직 큐 최적화 부분은 각 메시지 큐의 부하 상한/하한 기준(`LOGIC_QUEUE_APPEND_LIMIT` / `LOGIC_QUEUE_REDUCE_LIMIT`)에 맞춰 메시지 큐를 늘리거나 줄입니다.

만약 상한 기준을 충족한다면 새 메시지 큐를 하나 생성한 후, 기준을 충족한 메시지 큐의 절반을 새 메시지 큐에 할당합니다.

이슈 #1

문제의 요구사항과 몇몇 구현이 상이합니다.

- 메시지 큐의 크기가 10이 아니라 덱으로 구현하여 무한정 늘어날 수 있습니다.
- 코드 구현 상 부하 상한 하한 기준을 지정할 수는 있으나, 그것을 80%/20%가 아니라 80%/0% 로 지정해두었습니다.
- "새 서버 로직을 만든다"는 것이 "새 스레드를 만든다"라면 이 구현은 최소한 응답 부분의 스레드는 한 개이므로 적절한 답안이 아닙니다.

이슈 #2

```
$ python accesses.py 8089
>> [00] (Code: 200) My first web server!
>> [03] (Code: 200) My first web server!
>> [17] (Code: 200) My first web server!
>> [15] (Code: 200) My first web server!
>> [01] (Code: 200) My first web server!
>> [18] (Code: 200) My first web server!
>> [16] (Code: 200) My first web server!
>> [19] (Code: 200) My first web server!
>> [04] (Code: 200) My first web server!
>> [05] (Code: 200) My first web server!
>> [21] (Code: 200) My first web server!
>> [10] (Code: 200) My first web server!
>> [08] (Code: 200) My first web server!
>> [14] (Code: 200) My first web server!
>> [26] (Code: 200) My first web server!
>> [06] (Code: 200) My first web server!
```

```
>> [12] (Code: 200) My first web server!
>> [13] (Code: 200) My first web server!
>> [30] (Code: 200) My first web server!
>> [31] (Code: 200) My first web server!
>> [34] (Code: 200) My first web server!
>> [20] (Code: 200) My first web server!
>> [35] (Code: 200) My first web server!
>> [40] (Code: 200) My first web server!
>> [11] (Code: 200) My first web server!
>> [41] (Code: 200) My first web server!
>> [44] (Code: 200) My first web server!
>> [37] (Code: 200) My first web server!
>> [32] (Code: 200) My first web server!
>> [36] (Code: 200) My first web server!
>> [46] (Code: 200) My first web server!
>> [24] (Code: 200) My first web server!
>> [51] (Code: 200) My first web server!
>> [22] (Code: 200) My first web server!
>> [02] (Code: 200) My first web server!
>> [54] (Code: 200) My first web server!
>> [58] (Code: 200) My first web server!
>> [52] (Code: 200) My first web server!
>> [49] (Code: 200) My first web server!
>> [38] (Code: 200) My first web server!
>> [60] (Code: 200) My first web server!
>> [43] (Code: 200) My first web server!
>> [59] (Code: 200) My first web server!
>> [63] (Code: 200) My first web server!
>> [66] (Code: 200) My first web server!
>> [68] (Code: 200) My first web server!
>> [07] (Code: 200) My first web server!
>> [64] (Code: 200) My first web server!
>> [71] (Code: 200) My first web server!
>> [61] (Code: 200) My first web server!
>> [72] (Code: 200) My first web server!
>> [76] (Code: 200) My first web server!
>> [77] (Code: 200) My first web server!
>> [78] (Code: 200) My first web server!
>> [29] (Code: 200) My first web server!
>> [73] (Code: 200) My first web server!
>> [55] (Code: 200) My first web server!
>> [86] (Code: 200) My first web server!
>> [85] (Code: 200) My first web server!
>> [57] (Code: 200) My first web server!
>> [88] (Code: 200) My first web server!
>> [56] (Code: 200) My first web server!
>> [90] (Code: 200) My first web server!
>> [91] (Code: 200) My first web server!
```

```
>> [96] (Code: 200) My first web server!
>> [42] (Code: 200) My first web server!
>> [97] (Code: 200) My first web server!
>> [98] (Code: 200) My first web server!
>> [69] (Code: 200) My first web server!
>> [67] (Code: 200) My first web server!
>> [65] (Code: 200) My first web server!
>> [33] (Code: 200) My first web server!
>> [74] (Code: 200) My first web server!
>> [79] (Code: 200) My first web server!
>> [80] (Code: 200) My first web server!
>> [89] (Code: 200) My first web server!
>> [93] (Code: 200) My first web server!
>> [92] (Code: 200) My first web server!
>> [94] (Code: 200) My first web server!
>> [95] (Code: 200) My first web server!
>> [47] (Code: 200) My first web server!
>> [48] (Code: 200) My first web server!
>> [75] (Code: 200) My first web server!
>> [53] (Code: 200) My first web server!
>> [87] (Code: 200) My first web server!
>> [62] (Code: 200) My first web server!
>> [70] (Code: 200) My first web server!
>> [81] (Code: 200) My first web server!
>> [99] (Code: 200) My first web server!
>> [84] (Code: 200) My first web server!
>> [50] (Code: 200) My first web server!
>> [39] Connection error!
>> [45] Connection error!
>> [23] Connection error!
>> [25] Connection error!
>> [09] Connection error!
>> [83] Connection error!
>> [82] Connection error!
>> [28] Connection error!
>> [27] Connection error!
Elapsed time: 0:00:20.079573
```

몇 개 연결이 누락되다가 Timeout됩니다. 로직 큐 최적화 부분에서 문제가 발생할 것으로 추정하고 있으나, 정확한 원인 파악에는 이르지 못했습니다.

이슈 #3

```
void *pop_front(Deque *deque)
{
  (...)
  pthread_mutex_unlock(&deque->mutex);
```

```
    // free(now);
    return data;
}
```

pop 처리는 덱에서 원소가 제거된 것처럼 보이지만, 실제로 메모리에서 제거되지 않습니다. 제대로 파악하지 못한 원인으로 인해 `free` 시 비결정적으로 `SIGABRT` 가 발생하여 임시로 비활성화했는데, 이후 과제 마감시각까지 해결해내지 못했습니다.

결과적으로 이 코드는 메모리 누수를 유발합니다.

#4

원본 파일

과제 파일 `assignment01.zip` 의 `server.c` 코드는 `sys/socket.h` 와 `netinet/in.h` 를 이용하여 HTTP 요청을 처리하는 서버를 구현했습니다.

HTTP 요청을 수용할 포트를 바인딩한 후 루프를 만들어서, 루프를 탈출하기 전까지 계속해서 요청이 들어오는지 파악합니다.

루프 중에 요청을 성공적으로 받았다면 라이브러리가 생성한 파일 디스크립터를 이용하여 응답합니다.

#1 수정안

만약 `// sleep(5)` 의 주석을 해제하면 매 요청의 처리에 최소 5초가 소요되므로, 한번에 많은 요청이 들어오면 Timeout으로 제대로 수신되지 않은 요청이 발생할 수 있습니다. 제대로 수신되었다고 하더라도 상당한 시간이 소요됩니다.

당장 요청을 수신받는 부분으로부터 `sleep` 이 포함된 응답부를 분리하여 자식 프로세스로 동작하게 함으로서, 다시 말해 오래 걸리는 작업을 분리함으로서 응답성을 확보하였습니다.

#2 수정안

자식 프로세스를 생성하는 것도 결코 가벼운 작업은 아닙니다. PCB를 포함해 OS가 관리해야할 프로세스 개수가 늘어나는 것은 OS에 부하를 심하게 제공할 여지가 있으므로 되도록 경량화할 필요가 있습니다.

#1 시나리오에서 `accesses.py` 를 실행한다면 서버는 자식 프로세스까지 포함하여 101개의 프로세스를 동시에 실행할 수 있습니다. #2 시나리오는 #1 시나리오 상 자식 프로세스들의 공통된 부분을 공유하게 함으로서 조금 더 가볍게 동작할 수 있도록 합니다.

#1과 #2에는 어떤 차이가 존재하는가

#1에서 메인으로부터 분기된 작업들은 각각 독자적인 프로세스로서 동작합니다. 메인 프로세스가 종료된다고 하더라도 분기된 작업들 역시 독자적인 프로세스이므로 계속해서 동작합니다.

#2에서 메인으로부터 분기된 작업들은 분기되었다하더라도 하나의 프로세스입니다. 메인 작업의 프로세스가 곧 분기된 작업의 프로세스이므로 메인 작업 프로세스의 종료는 분기 작업의 프로세스의 종료를 의미합니다.

#1에서 분기된 작업들은 독립된 프로세스로서 갖춰야 할 모든 메모리 공간을 갖추고 있습니다. 따라서 #1과 같이 같은 내용을 가지고 분기된 작업들은 같은 내용의 데이터가 중복해서 메모리에 적재됩니다. 다시 말해 메모리가 낭비됩니다.

#2에서 분기된 작업들은 서로 많은 메모리 공간을 공유하고 있습니다. #1과 비교하여 메모리 상 중복된 내용이 적습니다.

낭비되는 자원은 메모리 뿐만이 아닙니다. OS가 관리해야하는 순수 프로세스의 양에서도 차이가 있으므로 #2에서는 다양한 컴퓨팅 자원을 절약할 수 있습니다.

#2는 #1보다 작업 간 데이터 공유가 쉽습니다. 작업이 독립되어있더라도 동일한 메모리 공간 위에 있으므로 #1에서와 같이 IPC를 위해 추가적인 작업을 할 소요가 줄어듭니다.