SWE2024

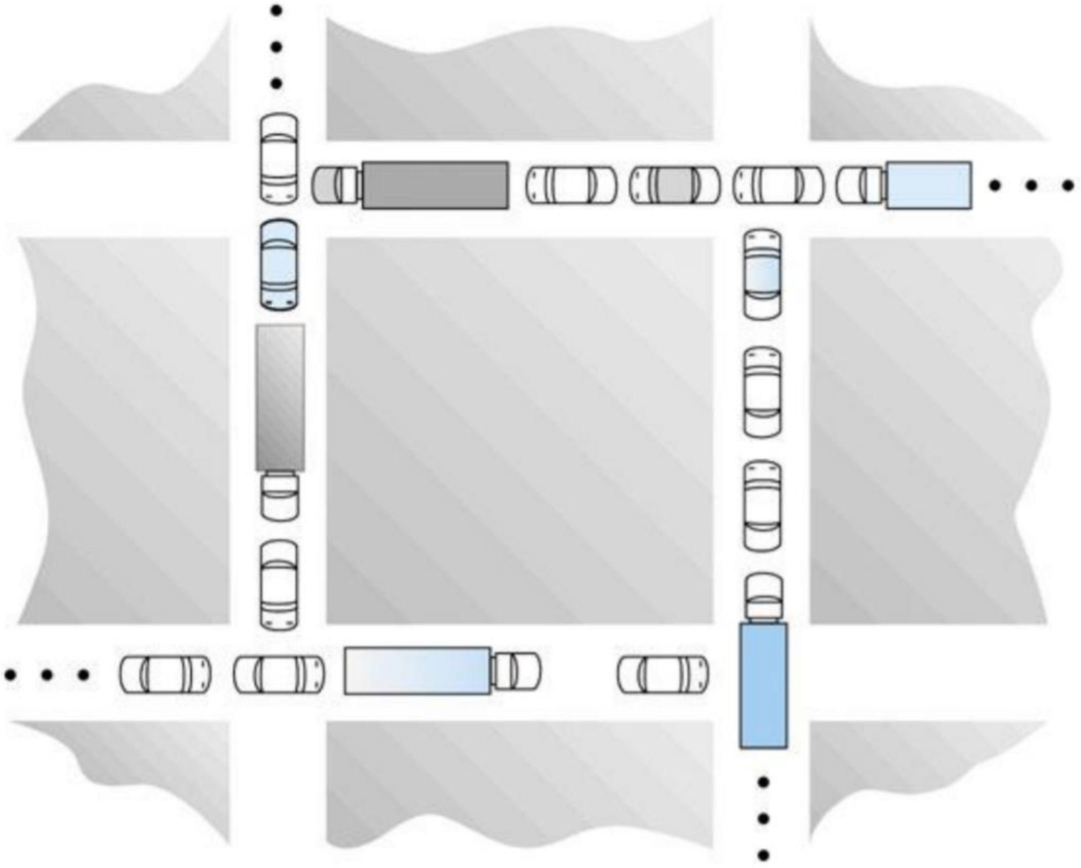
SUNG KYUN KWAN UNIVERSITY

Synchronization II

Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

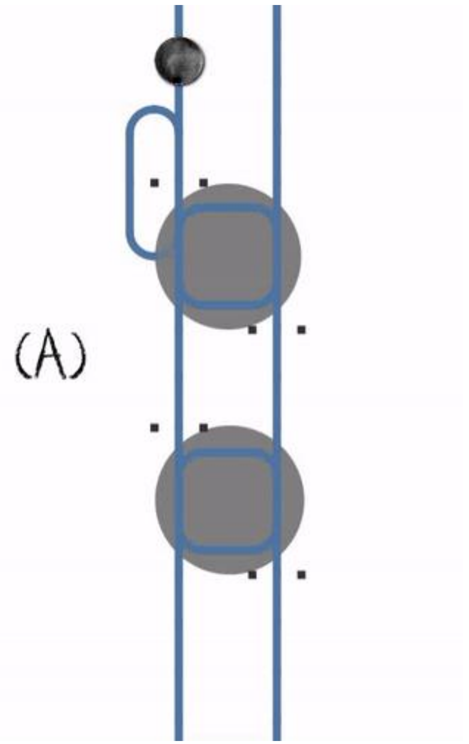
Deadlock (1)

- Traffic deadlock



Deadlock (2)

- Deadlock problem
  - A set of processes (threads) is blocked
    - **Reason:** Each process (thread) is holding a resource while waiting for another resource held by other(s).
  - Resources should be accessed in a reasonable order
    - More than one resource is sometimes required
    - Order in which the resources are allocated is important



Conditions for Deadlock (1)

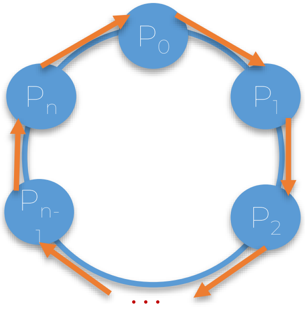
Deadlock can occur only if all four Coffman conditions hold simultaneously

- Mutual exclusion
  - Only one process at a time can use a resource
- Hold and wait
  - Process holding at least one resource is waiting to acquire additional resources held by other processes

Conditions for Deadlock (2)

- No preemption
  - Resource can be released only voluntarily by the process holding it, after that process has completed its task

- Circular wait
  - There must exist a set {P_0, P_1, ..., P_n} of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource held by P_2, etc.



Handling Deadlocks

- Strategies for handling deadlocks
  - Deadlock prevention
    - Restrain how requests are made
    - Ensure that at least one necessary condition cannot hold
  - Deadlock avoidance
    - Require additional information about how resources are to be requested
    - Decide to approve or disapprove requests on the fly
  - Deadlock detection and recovery
    - Allow the system to enter deadlock state and then recover
  - Just ignore the problem altogether!

Deadlock Prevention (1)

- Mutual exclusion
  - Not required for sharable resources, but must hold for non-sharable resources
- Hold and wait
  - Must guarantee that whenever process requests resource, it does not hold any other resources
    - Require process to request and be allocated all its resources before it begins execution
    - Allow process to request resources only when the process has none
  - Problems
    - May not know required resources at start of run
    - Low resource utilization
    - Starvation possible

Deadlock Prevention (2)

- No preemption
  - If process must wait for another resource, all resources currently being held are implicitly preempted
  - If requesting resources are allocated to some other process that is waiting for additional resources, preempt the desired resources from waiting process

- Circular wait
  - Impose total ordering of all resource types
  - Require that each process requests resources in an increasing order of enumeration

Deadlock Example (1)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <time.h>

int millisleep(long milliseconds) {
    return nanosleep((const struct timespec[])
        {{0, milliseconds * 1000000}}, NULL);
}
int main() {
    /* make thread and start thread */
    pthread_t thread_get, thread_set;
    pthread_create(&thread_set, NULL, &setter, NULL);
    millisleep(5);
    pthread_create(&thread_get, NULL, &getter, NULL);
    pthread_join(thread_set, NULL);
    pthread_join(thread_get, NULL);
    return 0;
}
```

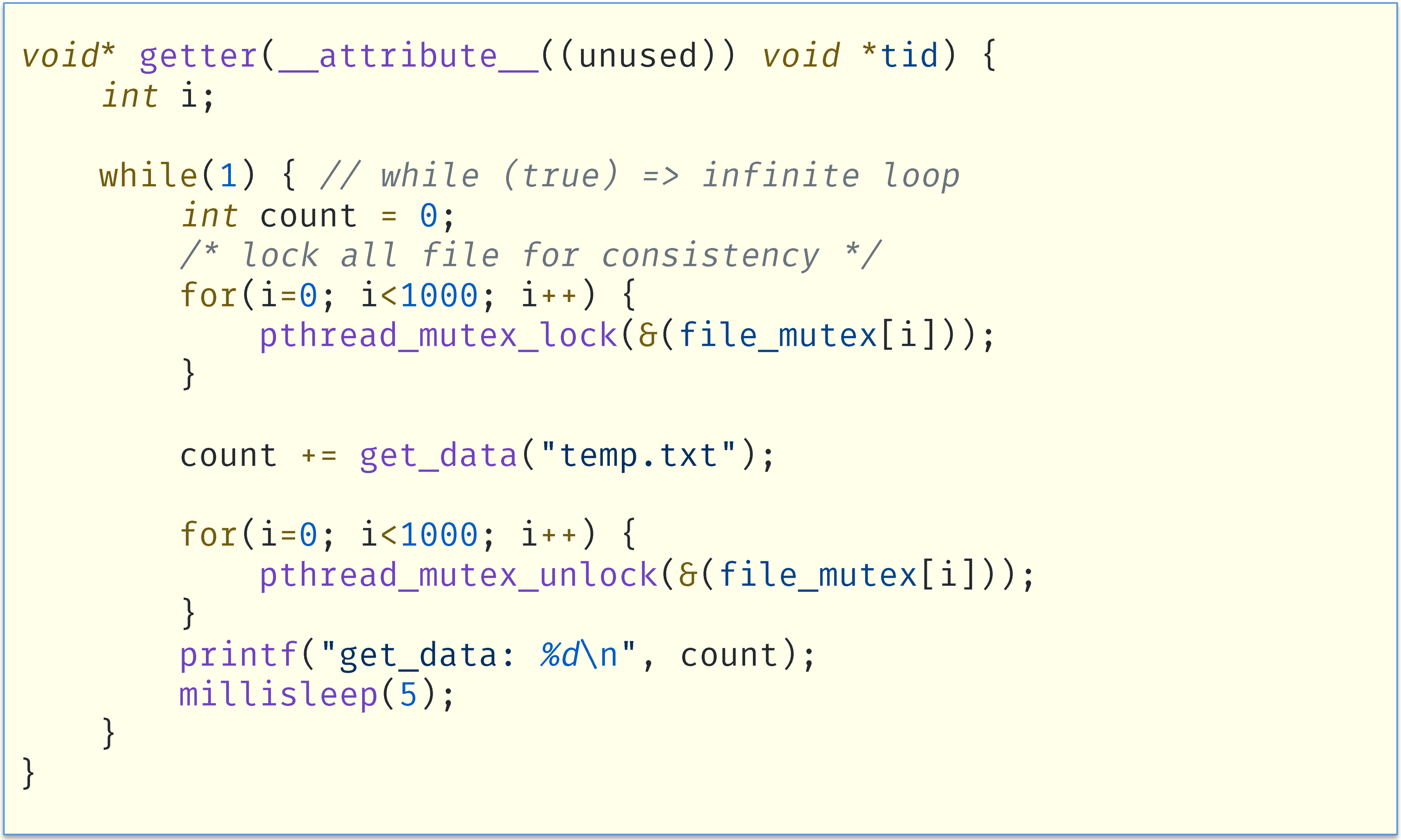
Deadlock Example (2)

```c
pthread_mutex_t file_mutex[1000]= {
        PTHREAD_MUTEX_INITIALIZER,
};

int get_data(char *path) {
    int fd = open(path, O_RDONLY);
    char buffer[2];
    read(fd, buffer, 1);
    close(fd);
    return (int)strtol(buffer, NULL, 10);
}

void set_data(char *path, int data) {
    int fd = open(path, O_WRONLY|O_CREAT, 0777);
    char buffer[2];
    memset(buffer, 0, 1);
    sprintf(buffer, "%d", data);
    write(fd, buffer, 1);
    close(fd);
    return;
}
```

Deadlock Example (3)

```c
void* setter(__attribute__((unused)) void *tid) {
    int count = 0;
    int i;

    while(1) { // while (true) => infinite loop
        /* lock all file for consistency */
        for(i=999; i>=0; i--) {
            pthread_mutex_lock(&(file_mutex[i]));
        }

        set_data("temp.txt", count);

        for(i=999; i>=0; i--) {
            pthread_mutex_unlock(&(file_mutex[i]));
        }
        count = (count + 1) % 10;
        millisleep(10);
    }
}
```

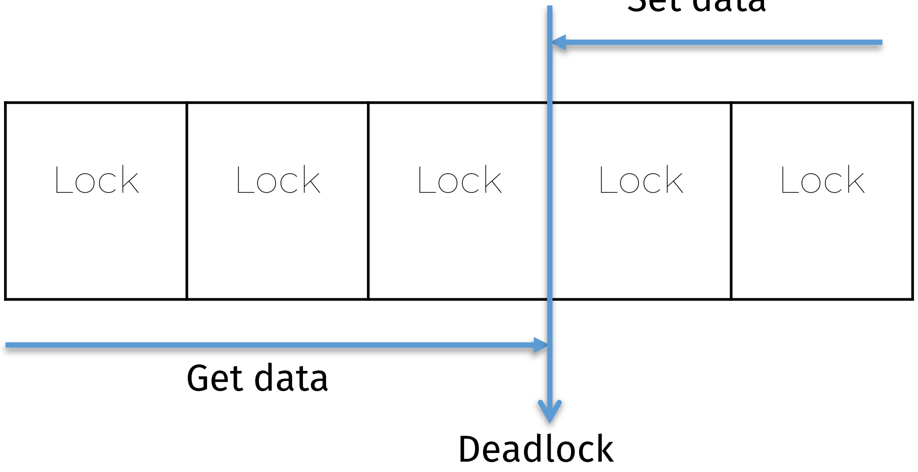
Deadlock Example (4)

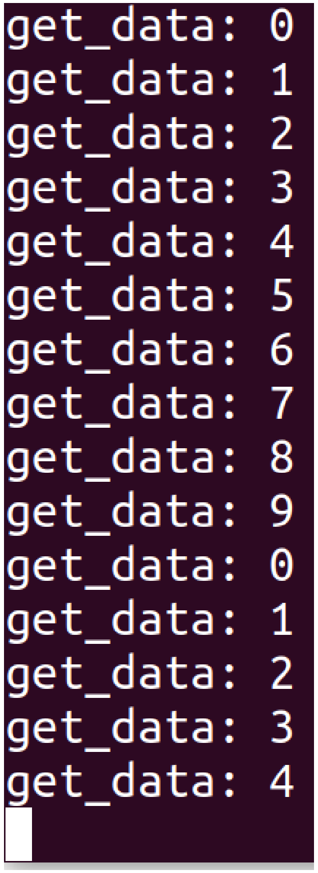


```
void* getter(__attribute__((unused)) void *tid) {
    int i;

    while(1) { // while (true) => infinite loop
        int count = 0;
        /* lock all file for consistency */
        for(i=0; i<1000; i++) {
            pthread_mutex_lock(&(file_mutex[i]));
        }

        count += get_data("temp.txt");

        for(i=0; i<1000; i++) {
            pthread_mutex_unlock(&(file_mutex[i]));
        }
        printf("get_data: %d\n", count);
        millisleep(5);
    }
}
```

Deadlock Example (5)

- Stop getting data
  - Deadlock





```
get_data: 0
get_data: 1
get_data: 2
get_data: 3
get_data: 4
get_data: 5
get_data: 6
get_data: 7
get_data: 8
get_data: 9
get_data: 0
get_data: 1
get_data: 2
get_data: 3
get_data: 4
```

Exercise

- Make producer & consumer working at same time
  - Implement `put_data()` and `get_data()` functions
    - `void put_data (queue_t* q, int d);`
    - `int  get_data (queue_t* q);`

- If the queue is full, producer cannot put anything to the queue, and if the queue is empty, consumer cannot get anything from the queue.
  - You should use a synchronization mechanism using CV

- All produced data must be consumed

- Skeleton code
  - `cp ~swe2024-41_23s/2023s/p14_skeleton.c ./`

Review: CV Creation / Termination API

- Static initialization
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- Dynamic initialization
  - `pthread_cond_t cond;`
  - `pthread_cond_init(&cond,(pthread_condattr_t*)NULL);`
- Condition variable termination
  - `pthread_cond_destroy(&cond);`
  - Destroy condition variable object
    - Free resources it might hold

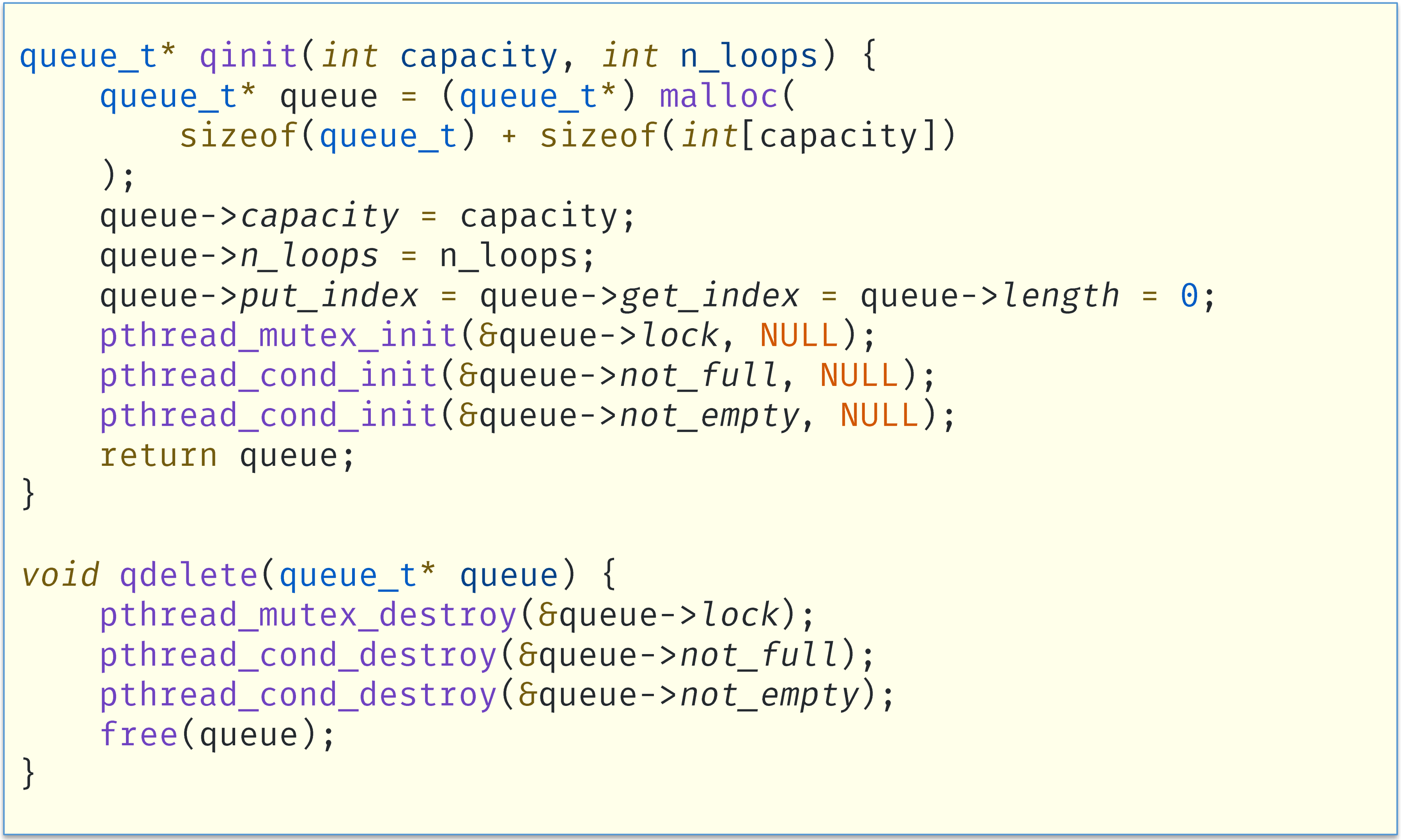
Review: CV Operation APIs

- `int pthread_cond_wait(pthread_cond_t* cond,`
  `                       pthread_mutex_t* mutex)`
  - Block calling thread until specified condition is signaled
  - This should be called while mutex is locked, and it will automatically release mutex while it waits
- `int pthread_cond_signal(pthread_cond_t* cond)`
  - Signal another thread which is waiting on CV
  - Calling thread should have a lock
- `int pthread_cond_broadcast(pthread_cond_t* cond)`
  - Used if more than one thread is in blocking wait state

Producer & Consumer (1)

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

#define QSIZE 5
#define LOOP 30

typedef struct {
    int put_index; // rear
    int get_index; // front
    int length;    // size
    int capacity;
    int n_loops;
    pthread_mutex_t lock;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;
    int data[];
} queue_t;

void* produce(void* args);
void* consume(void* args);
void put_data(queue_t* queue, int data);
int get_data(queue_t* queue);
```

Producer & Consumer (2)



```
queue_t* qinit(int capacity, int n_loops) {
    queue_t* queue = (queue_t*) malloc(
        sizeof(queue_t) + sizeof(int[capacity])
    );
    queue->capacity = capacity;
    queue->n_loops = n_loops;
    queue->put_index = queue->get_index = queue->length = 0;
    pthread_mutex_init(&queue->lock, NULL);
    pthread_cond_init(&queue->not_full, NULL);
    pthread_cond_init(&queue->not_empty, NULL);
    return queue;
}

void qdelete(queue_t* queue) {
    pthread_mutex_destroy(&queue->lock);
    pthread_cond_destroy(&queue->not_full);
    pthread_cond_destroy(&queue->not_empty);
    free(queue);
}
```

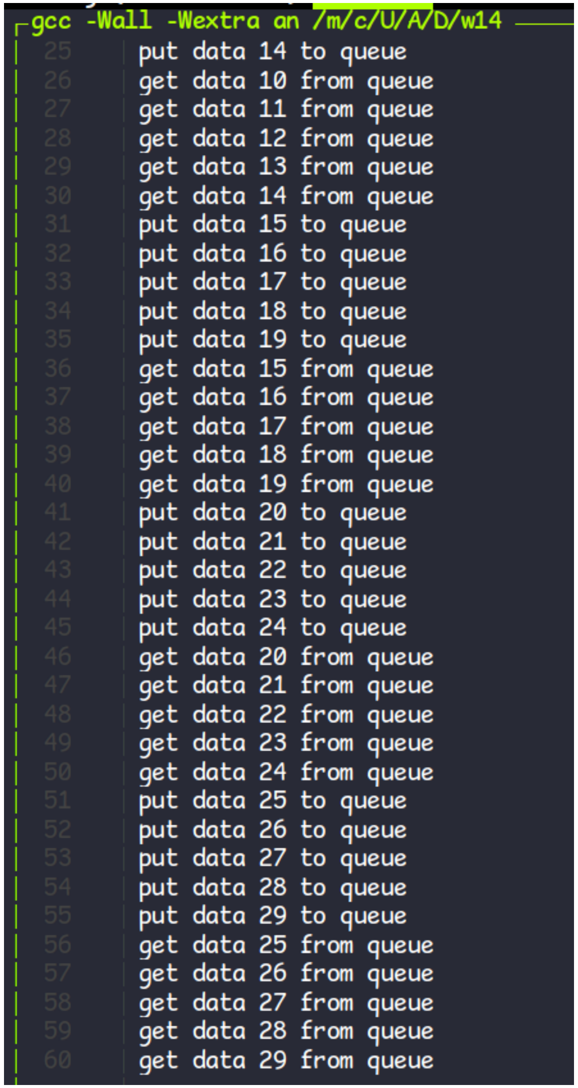
Producer & Consumer (3)

```c
int main() {
    queue_t* queue = qinit(QSIZE, LOOP);
    pthread_t producer, consumer;
    pthread_create(&producer, NULL, produce, (void *)queue);
    pthread_create(&consumer, NULL, consume, (void *)queue);
    pthread_join (producer, NULL);
    pthread_join (consumer, NULL);
    qdelete(queue);
}
```

Producer & Consumer (4)

```c
void* produce(void* args) {
    int i, data;
    queue_t* queue = (queue_t*)args;
    for (i = 0; i < queue->n_loops; i++) {
        data = i;
        put_data(queue, data);
        printf("put data %d to queue\n", data);
    }
    pthread_exit(NULL);
}

void* consume(void *args) {
    int i;
    queue_t* queue = (queue_t*)args;
    for (i = 0; i < queue->n_loops; i++) {
        printf("get data %d from queue\n", get_data(queue));
    }
    pthread_exit(NULL);
}
```

Exercise Example



```
gcc -Wall -Wextra an /m/c/U/A/D/w14
put data 14 to queue
get data 10 from queue
get data 11 from queue
get data 12 from queue
get data 13 from queue
get data 14 from queue
put data 15 to queue
put data 16 to queue
put data 17 to queue
put data 18 to queue
put data 19 to queue
get data 15 from queue
get data 16 from queue
get data 17 from queue
get data 18 from queue
get data 19 from queue
put data 20 to queue
put data 21 to queue
put data 22 to queue
put data 23 to queue
put data 24 to queue
get data 20 from queue
get data 21 from queue
get data 22 from queue
get data 23 from queue
get data 24 from queue
put data 25 to queue
put data 26 to queue
put data 27 to queue
put data 28 to queue
put data 29 to queue
get data 25 from queue
get data 26 from queue
get data 27 from queue
get data 28 from queue
get data 29 from queue
```

Exercise Submission

- Submit your exercise code and Makefile
  - InUiYeJi cluster
  - Submit the folder into p14
  - ~swe2024-41_23s/bin/submit p14 p14
  - Due date: Tonight, 31 May 2023, 23:59
  - We will compile by using command *make*
    - If compilation fails, your points for this exercise will be **zero**

Summary Report

- Write a summary report on deadlock
  - **Definition**
  - **Condition**
  - **Prevention**

- Submission form
  - A4 size PDF format (No page limitation)
  - [SWE2024 Report-14] studentID_name
  - e.g) [SWE2024 Report-14] 2022XXXXXX_홍길동
  - Submit to iCampus
  - Due by Friday, 2 June 2023, 23:59