1 producer consumer meutex

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int count = 0;

pthread\_mutex\_t mutex;

pthread\_cond\_t cond\_producer, cond\_consumer;

void\* producer(void\* arg) {

for (int i = 0; i < 10; i++) {

pthread\_mutex\_lock(&mutex);

while (count == BUFFER\_SIZE) {

pthread\_cond\_wait(&cond\_producer, &mutex);

}

buffer[count++] = i;

printf("Produced: %d\n", i);

pthread\_cond\_signal(&cond\_consumer);

pthread\_mutex\_unlock(&mutex);

}

return NULL;

}

void\* consumer(void\* arg) {

for (int i = 0; i < 10; i++) {

pthread\_mutex\_lock(&mutex);

while (count == 0) {

pthread\_cond\_wait(&cond\_consumer, &mutex);

}

int item = buffer[--count];

printf("Consumed: %d\n", item);

pthread\_cond\_signal(&cond\_producer);

pthread\_mutex\_unlock(&mutex);

}

return NULL;

}

int main() {

pthread\_t prod, cons;

pthread\_mutex\_init(&mutex, NULL);

pthread\_cond\_init(&cond\_producer, NULL);

pthread\_cond\_init(&cond\_consumer, NULL);

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

pthread\_mutex\_destroy(&mutex);

pthread\_cond\_destroy(&cond\_producer);

pthread\_cond\_destroy(&cond\_consumer);

return 0;

}

2 reader writer meutex

#include <stdio.h>

#include <pthread.h>

pthread\_mutex\_t mutex;

int data = 0;

void\* writer(void\* arg) {

pthread\_mutex\_lock(&mutex);

data++;

printf("Writer updated data to %d\n", data);

pthread\_mutex\_unlock(&mutex);

return NULL;

}

void\* reader(void\* arg) {

pthread\_mutex\_lock(&mutex);

printf("Reader read data as %d\n", data);

pthread\_mutex\_unlock(&mutex);

return NULL;

}

int main() {

pthread\_t writers[2], readers[2];

pthread\_mutex\_init(&mutex, NULL);

// Create writer threads

for (int i = 0; i < 2; i++)

pthread\_create(&writers[i], NULL, writer, NULL);

// Create reader threads

for (int i = 0; i < 2; i++)

pthread\_create(&readers[i], NULL, reader, NULL);

// Join threads

for (int i = 0; i < 2; i++) {

pthread\_join(writers[i], NULL);

pthread\_join(readers[i], NULL);

}

pthread\_mutex\_destroy(&mutex);

return 0;

}

3 FCFS

#include <stdio.h>

#include <stdlib.h>

void FCFS(int requests[], int n, int head) {

int seek\_time = 0;

printf("Sequence of disk access:\n%d", head);

for (int i = 0; i < n; i++) {

printf(" -> %d", requests[i]);

seek\_time += abs(requests[i] - head);

head = requests[i];

}

printf("\nTotal seek time: %d\n", seek\_time);

}

int main() {

int n, head;

printf("Enter the number of requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the request queue: ");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter initial head position: ");

scanf("%d", &head);

FCFS(requests, n, head);

return 0;

}

4 FIFO

#include <stdio.h>

void FIFO(int pages[], int n, int capacity) {

int frames[capacity];

int count = 0, page\_faults = 0;

for (int i = 0; i < capacity; i++) frames[i] = -1;

for (int i = 0; i < n; i++) {

int page = pages[i];

int found = 0;

for (int j = 0; j < capacity; j++) {

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

frames[count % capacity] = page;

count++;

page\_faults++;

}

printf("Frames: ");

for (int j = 0; j < capacity; j++) {

if (frames[j] != -1)

printf("%d ", frames[j]);

else

printf("- ");

}

printf("\n");

}

printf("Total Page Faults: %d\n", page\_faults);

}

int main() {

int n, capacity;

printf("Enter the number of page requests: ");

scanf("%d", &n);

int pages[n];

printf("Enter the pages: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pages[i]);

}

printf("Enter the frame capacity: ");

scanf("%d", &capacity);

FIFO(pages, n, capacity);

return 0;

}

5 LRU

#include <stdio.h>

void LRU(int pages[], int n, int capacity) {

int frames[capacity];

int time[capacity];

int page\_faults = 0, oldest;

for (int i = 0; i < capacity; i++) frames[i] = -1;

for (int i = 0; i < n; i++) {

int page = pages[i];

int found = 0;

for (int j = 0; j < capacity; j++) {

if (frames[j] == page) {

found = 1;

time[j] = i;

break;

}

}

if (!found) {

oldest = 0;

for (int j = 1; j < capacity; j++) {

if (time[j] < time[oldest]) oldest = j;

}

frames[oldest] = page;

time[oldest] = i;

page\_faults++;

}

printf("Frames: ");

for (int j = 0; j < capacity; j++) {

if (frames[j] != -1)

printf("%d ", frames[j]);

else

printf("- ");

}

printf("\n");

}

printf("Total Page Faults: %d\n", page\_faults);

}

int main() {

int n, capacity;

printf("Enter the number of page requests: ");

scanf("%d", &n);

int pages[n];

printf("Enter the pages: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pages[i]);

}

printf("Enter the frame capacity: ");

scanf("%d", &capacity);

LRU(pages, n, capacity);

return 0;

}

6 Optimal page replacement

#include <stdio.h>

int predict(int pages[], int n, int page, int index) {

int farthest = index, res = -1;

for (int i = 0; i < n; i++) {

int j;

for (j = index; j < n; j++) {

if (pages[j] == page) {

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

if (j == n) return i;

}

return res == -1 ? 0 : res;

}

void OptimalPageReplacement(int pages[], int n, int capacity) {

int frames[capacity];

int page\_faults = 0;

for (int i = 0; i < capacity; i++) frames[i] = -1;

for (int i = 0; i < n; i++) {

int page = pages[i], found = 0;

for (int j = 0; j < capacity; j++) {

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

if (i < capacity) {

frames[i] = page;

} else {

int j = predict(pages, n, frames, i + 1);

frames[j] = page;

}

page\_faults++;

}

printf("Frames: ");

for (int j = 0; j < capacity; j++) {

if (frames[j] != -1)

printf("%d ", frames[j]);

else

printf("- ");

}

printf("\n");

}

printf("Total Page Faults: %d\n", page\_faults);

}

int main() {

int n, capacity;

printf("Enter the number of page requests: ");

scanf("%d", &n);

int pages[n];

printf("Enter the pages: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pages[i]);

}

printf("Enter the frame capacity: ");

scanf("%d", &capacity);

OptimalPageReplacement(pages, n, capacity);

return 0;

}

7 dining phil semaphore

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM\_PHILOSOPHERS 5

sem\_t forks[NUM\_PHILOSOPHERS];

pthread\_t philosophers[NUM\_PHILOSOPHERS];

void\* philosopher(void\* num) {

int id = \*(int\*)num;

printf("Philosopher %d is thinking.\n", id);

sem\_wait(&forks[id]);

sem\_wait(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

printf("Philosopher %d is eating.\n", id);

sem\_post(&forks[id]);

sem\_post(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

printf("Philosopher %d finished eating.\n", id);

return NULL;

}

int main() {

int ids[NUM\_PHILOSOPHERS];

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_init(&forks[i], 0, 1);

ids[i] = i;

}

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_join(philosophers[i], NULL);

}

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_destroy(&forks[i]);

}

return 0;

}

8 deadlock

#include <stdio.h>

void findDeadlock(int processes, int resources, int allocation[][resources], int request[][resources], int available[]) {

int finish[processes], safe\_sequence[processes];

int index = 0;

for (int i = 0; i < processes; i++) finish[i] = 0;

for (int k = 0; k < processes; k++) {

for (int i = 0; i < processes; i++) {

if (finish[i] == 0) {

int can\_execute = 1;

for (int j = 0; j < resources; j++) {

if (request[i][j] > available[j]) {

can\_execute = 0;

break;

}

}

if (can\_execute) {

for (int j = 0; j < resources; j++) {

available[j] += allocation[i][j];

}

safe\_sequence[index++] = i;

finish[i] = 1;

}

}

}

}

int deadlock = 0;

for (int i = 0; i < processes; i++) {

if (finish[i] == 0) {

deadlock = 1;

printf("System is in deadlock due to process %d.\n", i);

}

}

if (!deadlock) {

printf("System is not in deadlock. Safe sequence is: ");

for (int i = 0; i < processes; i++) {

printf("%d ", safe\_sequence[i]);

}

printf("\n");

}

}

int main() {

int processes, resources;

printf("Enter number of processes: ");

scanf("%d", &processes);

printf("Enter number of resources: ");

scanf("%d", &resources);

int allocation[processes][resources], request[processes][resources], available[resources];

printf("Enter allocation matrix:\n");

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

scanf("%d", &allocation[i][j]);

}

}

printf("Enter request matrix:\n");

for (int i = 0; i < processes; i++) {

for (int j = 0; j < resources; j++) {

scanf("%d", &request[i][j]);

}

}

printf("Enter available resources: ");

for (int i = 0; i < resources; i++) {

scanf("%d", &available[i]);

}

findDeadlock(processes, resources, allocation, request, available);

return 0;

}

9 SCAN

#include <stdio.h>

#include <stdlib.h>

void SCAN(int requests[], int n, int head, int disk\_size, int direction) {

int seek\_time = 0;

int distance, cur\_track;

int left[n], right[n];

int l = 0, r = 0;

// Requests left and right of the head

for (int i = 0; i < n; i++) {

if (requests[i] < head)

left[l++] = requests[i];

else

right[r++] = requests[i];

}

// Sorting the requests

qsort(left, l, sizeof(int), (int (\*)(const void \*, const void \*))cmpfunc);

qsort(right, r, sizeof(int), (int (\*)(const void \*, const void \*))cmpfunc);

printf("Seek sequence: %d", head);

// Service requests in the current direction

if (direction == 1) {

for (int i = 0; i < r; i++) {

cur\_track = right[i];

distance = abs(cur\_track - head);

seek\_time += distance;

head = cur\_track;

printf(" -> %d", head);

}

head = disk\_size - 1;

printf(" -> %d", head);

for (int i = l - 1; i >= 0; i--) {

cur\_track = left[i];

distance = abs(cur\_track - head);

seek\_time += distance;

head = cur\_track;

printf(" -> %d", head);

}

} else {

for (int i = l - 1; i >= 0; i--) {

cur\_track = left[i];

distance = abs(cur\_track - head);

seek\_time += distance;

head = cur\_track;

printf(" -> %d", head);

}

head = 0;

printf(" -> %d", head);

for (int i = 0; i < r; i++) {

cur\_track = right[i];

distance = abs(cur\_track - head);

seek\_time += distance;

head = cur\_track;

printf(" -> %d", head);

}

}

printf("\nTotal seek time: %d\n", seek\_time);

}

int cmpfunc(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b);

}

int main() {

int n, head, disk\_size, direction;

printf("Enter the number of requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the request sequence: ");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter initial head position: ");

scanf("%d", &head);

printf("Enter disk size: ");

scanf("%d", &disk\_size);

printf("Enter direction (1 for high, 0 for low): ");

scanf("%d", &direction);

SCAN(requests, n, head, disk\_size, direction);

return 0;

}

10 C-SCAN

#include <stdio.h>

#include <stdlib.h>

void CSCAN(int requests[], int n, int head, int disk\_size) {

int seek\_time = 0;

int distance, cur\_track;

int left[n], right[n];

int l = 0, r = 0;

for (int i = 0; i < n; i++) {

if (requests[i] < head)

left[l++] = requests[i];

else

right[r++] = requests[i];

}

qsort(left, l, sizeof(int), (int (\*)(const void \*, const void \*))cmpfunc);

qsort(right, r, sizeof(int), (int (\*)(const void \*, const void \*))cmpfunc);

printf("Seek sequence: %d", head);

for (int i = 0; i < r; i++) {

cur\_track = right[i];

distance = abs(cur\_track - head);

seek\_time += distance;

head = cur\_track;

printf(" -> %d", head);

}

head = 0;

seek\_time += disk\_size - 1;

printf(" -> 0 -> %d", disk\_size - 1);

for (int i = 0; i < l; i++) {

cur\_track = left[i];

distance = abs(cur\_track - head);

seek\_time += distance;

head = cur\_track;

printf(" -> %d", head);

}

printf("\nTotal seek time: %d\n", seek\_time);

}

int cmpfunc(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b);

}

int main() {

int n, head, disk\_size;

printf("Enter the number of requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the request sequence: ");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter initial head position: ");

scanf("%d", &head);

printf("Enter disk size: ");

scanf("%d", &disk\_size);

CSCAN(requests, n, head, disk\_size);

return 0;

}

11 SSTF

#include <stdio.h>

#include <stdlib.h>

int findClosestRequest(int requests[], int n, int head, int processed[]) {

int min\_distance = 1e5;

int closest\_index = -1;

for (int i = 0; i < n; i++) {

if (!processed[i]) {

int distance = abs(requests[i] - head);

if (distance < min\_distance) {

min\_distance = distance;

closest\_index = i;

}

}

}

return closest\_index;

}

void SSTF(int requests[], int n, int head) {

int seek\_time = 0;

int processed[n];

for (int i = 0; i < n; i++) processed[i] = 0;

printf("Seek sequence: %d", head);

for (int i = 0; i < n; i++) {

int index = findClosestRequest(requests, n, head, processed);

if (index == -1) break;

processed[index] = 1;

seek\_time += abs(requests[index] - head);

head = requests[index];

printf(" -> %d", head);

}

printf("\nTotal seek time: %d\n", seek\_time);

}

int main() {

int n, head;

printf("Enter the number of requests: ");

scanf("%d", &n);

int requests[n];

printf("Enter the request sequence: ");

for (int i = 0; i < n; i++) {

scanf("%d", &requests[i]);

}

printf("Enter initial head position: ");

scanf("%d", &head);

SSTF(requests, n, head);

return 0;

}

12 Round Robin

#include <stdio.h>

struct Process {

int pid;

int burst\_time;

int remaining\_time;

int waiting\_time;

int turnaround\_time;

};

void RoundRobin(struct Process processes[], int n, int quantum) {

int completed = 0, time = 0;

while (completed < n) {

int progress = 0;

for (int i = 0; i < n; i++) {

if (processes[i].remaining\_time > 0) {

if (processes[i].remaining\_time > quantum) {

time += quantum;

processes[i].remaining\_time -= quantum;

} else {

time += processes[i].remaining\_time;

processes[i].waiting\_time = time - processes[i].burst\_time;

processes[i].turnaround\_time = time;

processes[i].remaining\_time = 0;

completed++;

}

progress = 1;

}

}

if (!progress) break; // All processes completed

}

}

int main() {

int n, quantum;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

for (int i = 0; i < n; i++) {

processes[i].pid = i + 1;

printf("Enter burst time for process %d: ", i + 1);

scanf("%d", &processes[i].burst\_time);

processes[i].remaining\_time = processes[i].burst\_time;

}

printf("Enter time quantum: ");

scanf("%d", &quantum);

RoundRobin(processes, n, quantum);

printf("PID\tBurst\tWaiting\tTurnaround\n");

for (int i = 0; i < n; i++) {

printf("%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].burst\_time,

processes[i].waiting\_time, processes[i].turnaround\_time);

}

return 0;

}

13 producer consumer semaphore

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int count = 0;

sem\_t empty\_slots; // Tracks empty slots in the buffer

sem\_t full\_slots; // Tracks full slots in the buffer

pthread\_mutex\_t mutex;

void\* producer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&empty\_slots); // Wait if buffer is full

pthread\_mutex\_lock(&mutex); // Lock the buffer

buffer[count++] = i; // Produce an item

printf("Produced: %d\n", i);

pthread\_mutex\_unlock(&mutex); // Unlock the buffer

sem\_post(&full\_slots); // Increment full slots

}

return NULL;

}

void\* consumer(void\* arg) {

for (int i = 0; i < 10; i++) {

sem\_wait(&full\_slots); // Wait if buffer is empty

pthread\_mutex\_lock(&mutex); // Lock the buffer

int item = buffer[--count]; // Consume an item

printf("Consumed: %d\n", item);

pthread\_mutex\_unlock(&mutex); // Unlock the buffer

sem\_post(&empty\_slots); // Increment empty slots

}

return NULL;

}

int main() {

pthread\_t prod, cons;

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&empty\_slots, 0, BUFFER\_SIZE); // Buffer initially empty

sem\_init(&full\_slots, 0, 0); // No items initially

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&empty\_slots);

sem\_destroy(&full\_slots);

return 0;

}

14 dining philosopher semaphore

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM\_PHILOSOPHERS 5

sem\_t forks[NUM\_PHILOSOPHERS];

pthread\_t philosophers[NUM\_PHILOSOPHERS];

void\* philosopher(void\* num) {

int id = \*(int\*)num;

// Philosopher thinking

printf("Philosopher %d is thinking.\n", id);

// Pick up forks

sem\_wait(&forks[id]);

sem\_wait(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

// Eating

printf("Philosopher %d is eating.\n", id);

// Put down forks

sem\_post(&forks[id]);

sem\_post(&forks[(id + 1) % NUM\_PHILOSOPHERS]);

// Finished eating

printf("Philosopher %d finished eating.\n", id);

return NULL;

}

int main() {

int ids[NUM\_PHILOSOPHERS];

// Initialize semaphores for each fork

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_init(&forks[i], 0, 1);

ids[i] = i;

}

// Create threads for each philosopher

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

// Wait for all philosopher threads to finish

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_join(philosophers[i], NULL);

}

// Destroy semaphores

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

sem\_destroy(&forks[i]);

}

return 0;

}

15 zombie process

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid = fork();

if (pid > 0) {

// Parent process

printf("Parent process, PID: %d\n", getpid());

sleep(5); // Parent waits, making the child process a zombie

} else if (pid == 0) {

// Child process

printf("Child process (zombie), PID: %d\n", getpid());

exit(0); // Child exits, becoming a zombie until parent calls wait

} else {

printf("Fork failed.\n");

}

return 0;

}

16 orphan

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

pid\_t pid = fork();

if (pid > 0) {

// Parent process

printf("Parent process, PID: %d\n", getpid());

exit(0); // Parent exits, leaving the child process orphaned

} else if (pid == 0) {

// Child process

sleep(5); // Wait to let the parent exit first

printf("Child process (orphan), PID: %d, Parent PID: %d\n", getpid(), getppid());

} else {

printf("Fork failed.\n");

}

return 0;

}

17 sum of even…

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

void calculate\_even\_sum(int arr[], int n) {

int sum = 0;

for (int i = 0; i < n; i++) {

if (arr[i] % 2 == 0) {

sum += arr[i];

}

}

printf("Sum of even numbers (Parent): %d\n", sum);

}

void calculate\_odd\_sum(int arr[], int n) {

int sum = 0;

for (int i = 0; i < n; i++) {

if (arr[i] % 2 != 0) {

sum += arr[i];

}

}

printf("Sum of odd numbers (Child): %d\n", sum);

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int n = sizeof(arr) / sizeof(arr[0]);

pid\_t pid = fork();

if (pid > 0) {

// Parent process calculates sum of even numbers

wait(NULL); // Wait for the child process to finish

calculate\_even\_sum(arr, n);

} else if (pid == 0) {

// Child process calculates sum of odd numbers

calculate\_odd\_sum(arr, n);

exit(0);

} else {

printf("Fork failed.\n");

}

return 0;

}