#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid = fork();

if (pid < 0) {

perror("Fork failed");

return 1;

} else if (pid == 0) {

for (int i = 1; i <= 10; i++) {

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

sleep(1); }

} else {

wait(NULL);

}

return 0;

}

2.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid1 = fork(); // Create the first child process

if (pid1 < 0) {

// Error occurred in creating the first child

perror("Fork failed");

return 1;

} else if (pid1 == 0) {

// This block executes in the first child process

printf("First Child Process:\n");

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

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

exit(0); // Exit first child

} else {

// Create the second child process

pid\_t pid2 = fork();

if (pid2 < 0) {

// Error occurred in creating the second child

perror("Fork failed");

return 1;

} else if (pid2 == 0) {

// This block executes in the second child process

printf("Second Child Process:\n");

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

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

exit(0); // Exit second child

} else {

// This block executes in the parent process

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

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

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

}

}

return 0;

}

3.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

for (int i = 0; i < 3; i++) {

pid = fork(); // Create a child process

if (pid < 0) {

// Error occurred

perror("Fork failed");

return 1;

} else if (pid == 0) {

// In the child process

printf("Child Process:\n");

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

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

// Let the child create the next one

} else {

// In the parent process

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

break; // Break out of the loop, since only the parent should loop

}

}

return 0;

}

4. #include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

int main() {

pid\_t pid;

// Print the PID of the current process (parent)

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

// Create a child process

pid = fork();

if (pid < 0) {

// Fork failed

perror("Fork failed");

exit(1);

} else if (pid == 0) {

// Child process

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

} else {

// Parent process

printf("Parent Process: PID = %d, Child PID = %d\n", getpid(), pid);

}

return 0;

}

Assignment:

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t chopsticks[NUM\_PHILOSOPHERS];

void think(int philosopher\_id) {

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

usleep((rand() % 100 + 50) \* 1000);

}

void eat(int philosopher\_id) {

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

usleep((rand() % 100 + 50) \* 1000);

}

void\* philosopher(void\* arg) {

int philosopher\_id = \*(int\*)arg;

int left\_chopstick = philosopher\_id;

int right\_chopstick = (philosopher\_id + 1) % NUM\_PHILOSOPHERS;

while (1) {

think(philosopher\_id);

// Lock chopsticks in a consistent order to avoid deadlock

if (philosopher\_id % 2 == 0) {

pthread\_mutex\_lock(&chopsticks[left\_chopstick]);

pthread\_mutex\_lock(&chopsticks[right\_chopstick]);

} else {

pthread\_mutex\_lock(&chopsticks[right\_chopstick]);

pthread\_mutex\_lock(&chopsticks[left\_chopstick]);

}

eat(philosopher\_id);

pthread\_mutex\_unlock(&chopsticks[left\_chopstick]);

pthread\_mutex\_unlock(&chopsticks[right\_chopstick]);

}

return NULL;

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int philosopher\_ids[NUM\_PHILOSOPHERS];

// Initialize mutexes for chopsticks

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

pthread\_mutex\_init(&chopsticks[i], NULL);

}

// Create philosopher threads

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

philosopher\_ids[i] = i;

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

}

// Join philosopher threads

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

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

}

// Destroy mutexes

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

pthread\_mutex\_destroy(&chopsticks[i]);

}

return 0;

}