Winston Shine

Operating Systems

Lab 8

5/21/2024

For this lab I worked with Nathan Deanon and Tyrese

A semaphore can be used as either a lock or condition variables, you can pass it a condition variable and also use it as a lock to ensure a thread waits until a condition is met.

It seems tricky to use it as a lock, I understand that it can be but I think the roles are reversed. for example with locks one threads locks as it enters a critical section, preventing other threads from entering. With semaphores a thread would call wait to check if its ok to proceed before entering a critical section, and then post when it's finished.

1. The correct value to pass sem init was 0, settings it to 1 I believe caused the parent to not wait at all since the condition was already set.

```c
void *child(void *arg) {
    printf("child\n");
    // use semaphore here
    sem_post(&s);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init semaphore here
    sem_init(&s, 0, 0);
    Pthread_create(&p, NULL, child, NULL);
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

2. The solution was to to have each threads do reverse ordered post/waits on 2 semaphores. this way these two threads will always meet in middle of their function.

```c
void *child_1(void *arg) {
    printf("child 1: before\n");
    // what goes here?
    sleep(1);
    sem_wait(&s2); // wait on s2
    sleep(1);
    sem_post(&s1); // signal s1 is ok to go
    sleep(1);
    printf("child 1: after\n");
    return NULL;
}

void *child_2(void *arg) {
    printf("child 2: before\n");
    // what goes here?
    sleep(1);
    sem_post(&s2); // signal s2 is ok to go
    sleep(1);
    sem_wait(&s1); // wait on s1
    sleep(1);
    printf("child 2: after\n");
    return NULL;
}
```

3.