OPERATING SYSTEM PROJECT
ABSTRACT

Project Group
Team Members

Harshitha Yerraguntla(013800280)
Nithya Kuchadi (013769665)
Koushik Kumar Kamala(013766571)

Table of Contents

ABSTRACT:

Most operating systems identify processes according to a unique process identifier (or pid), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel. An operating system's pid manager is responsible for managing process identifiers. When a process is first created, it is assigned a unique pid by the pid manager. The pid is returned to the pid manager when the process completes execution, and the manager may later reassign this pid. The essential characteristic of process identifiers is that they must be unique, i.e., no two active processes can have the same pid.

MOTIVATION AND GOAL(REQUIREMENTS):

To Implement a PID Manager that allocates a unique process identifier to each process, using a data structure that represents the availability of process identifiers that are safe from race conditions.

METHODOLOGY

Designed a PID manager that will assign pid values ranging from 300 to 5000. This program creates the requested number of threads and releases the process identifiers. API methods are created for allocating and releasing a pid. The program uses Pthreads mutex locks to to ensure that the data structure used to represent the availability of process identifiers is safe from race conditions and Bitmap data structure in which a value of 0 at position i indicates that a process id is available and a value of 1 indicates that the process id is currently in use. Implemented a multithreaded program in which each thread will request a pid, sleep for a random period of time and then release the pid. Once released the respective thread is uninitialized.

IMPLEMENTATION:

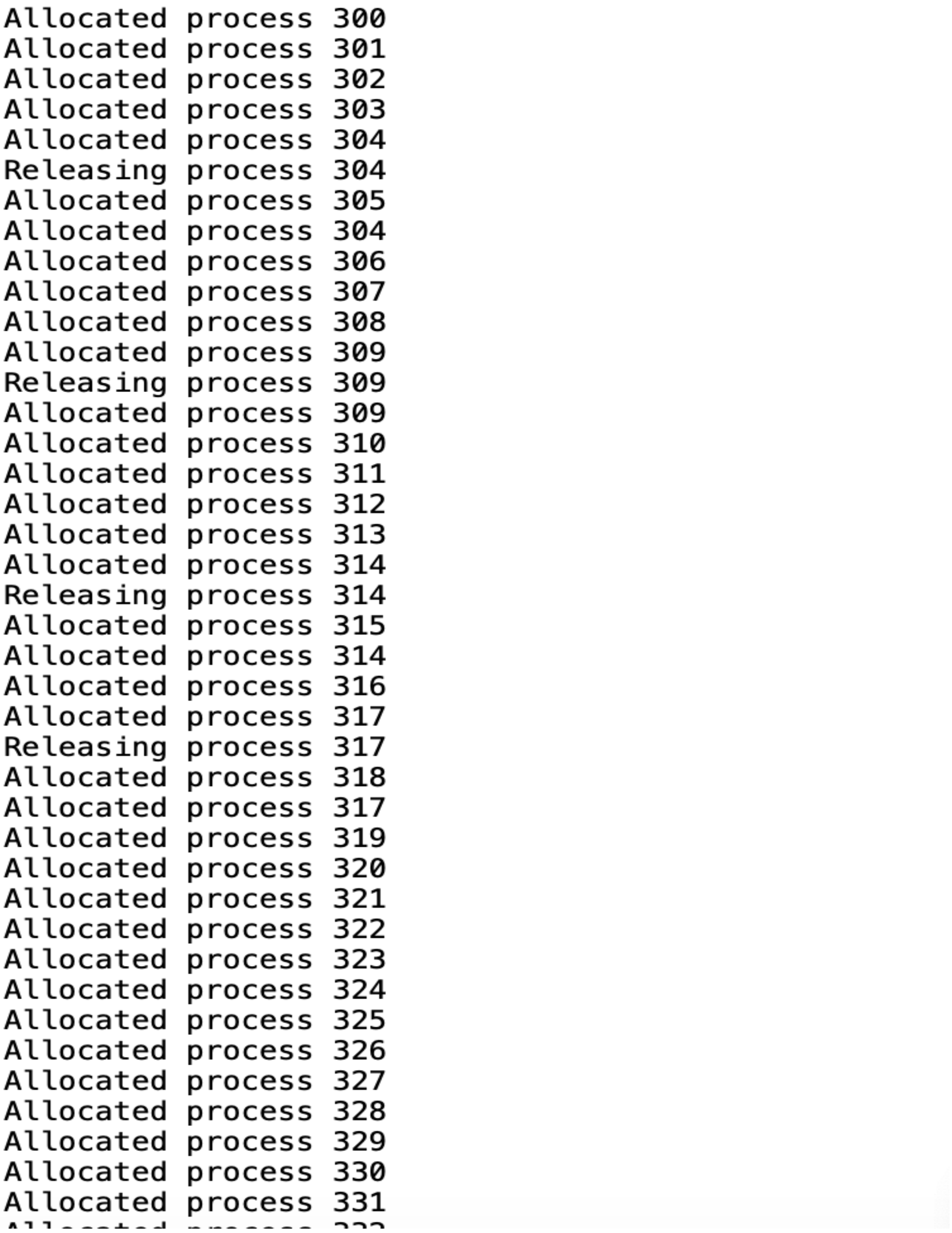
In main() thread of type pthread_t is declared. The program uses pthread_create() function to create a new thread and makes it executable. It takes four arguments. The first argument is a pointer to thread_id which is set by this function. The second argument specifies attributes which we want the new thread to contain. If the value is NULL, then default attributes shall be used. The third argument is the name of the function to be executed for the thread to be created. It could be a function pointer. The fourth argument is used to pass arguments to the function. The void type was chosen because if a function accepts more than one argument, then this pointer could be a pointer to a structure that may contain these arguments.
Implemented mutex concept for synchronization and to protect the shared resources. pthread_mutex_init is used to initialize the mutex variable .pthread_mutex_lock and pthread_mutex_unlock functions are used to lock the thread. If the mutex is locked already by another thread, the thread waits for the mutex to become available. The thread

that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. pthread_mutex_destroy function destroys the mutex variable. The function allocate_map does allocate and initializes the data structure for representing pids. Moreover, the allocate_pid method does allocate a pid to the new process and release_pid releases the pid. Moreover, After getting a pid, it acquires the mutex lock using pthread_mutex_lock, goes for sleep by calling the method sleep(10000) and then releases the lock using pthread_mutex_unlock. Finally, the thread gets uninitialized using pthread_mutex_destroy It shall be safe to destroy an initialized mutex that is unlocked — attempting to destroy locked mutex results in undefined behavior.

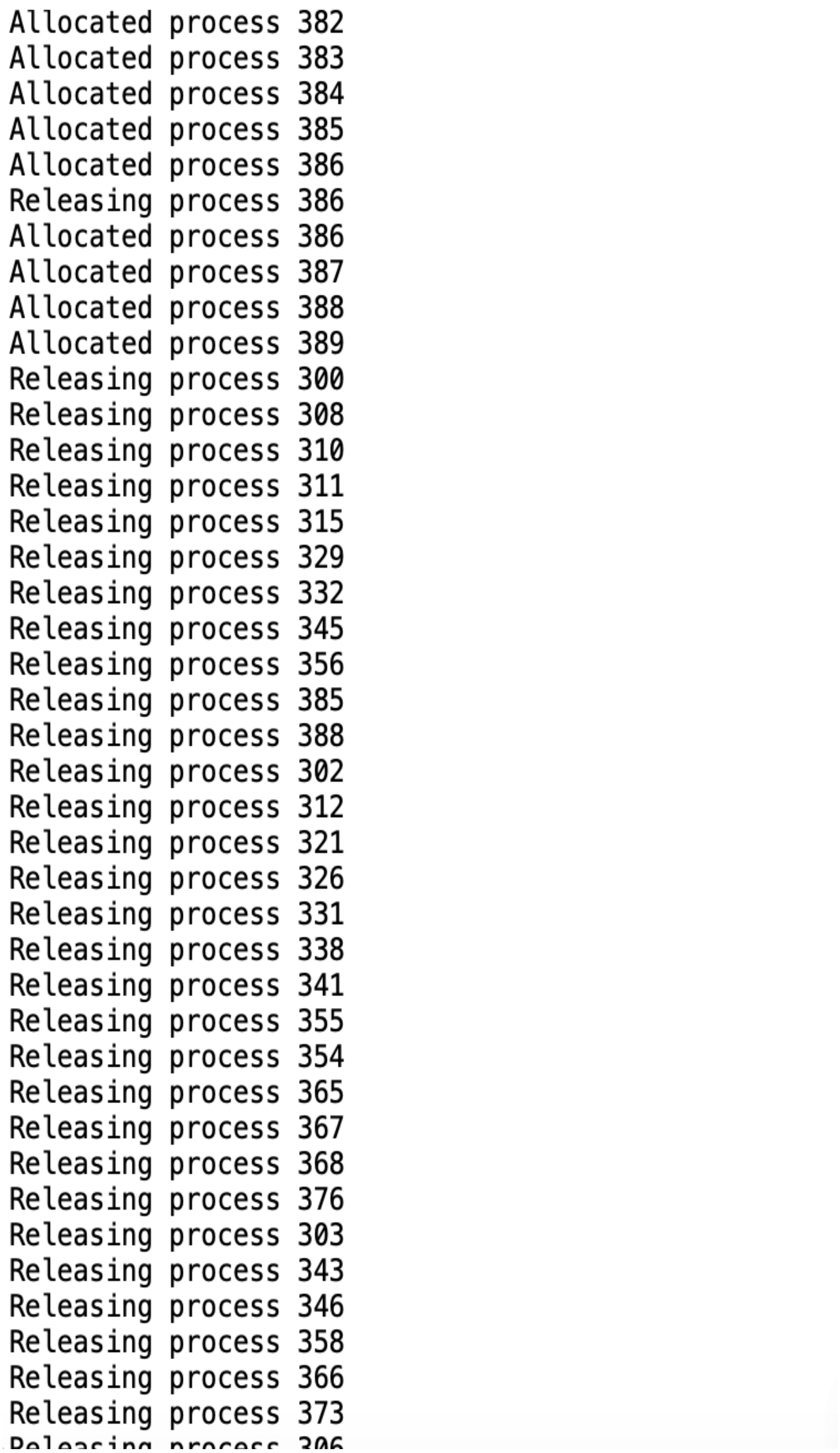
TESTING:

This program creates a requested number of threads and ensures that the data structure used to represent the availability of process identifiers is safe from race conditions. The program creates a thread, and each thread regularly requests a PID in multiple iterations. Once PID is allocated, thread sleeps for a random period of time, and then the process gets terminating releasing the PID. Output reflects the random acquiring and releasing of PID.

RESULTS:



```
Allocated process 300
Allocated process 301
Allocated process 302
Allocated process 303
Allocated process 304
Releasing process 304
Allocated process 305
Allocated process 304
Allocated process 306
Allocated process 307
Allocated process 308
Allocated process 309
Releasing process 309
Allocated process 309
Allocated process 310
Allocated process 311
Allocated process 312
Allocated process 313
Allocated process 314
Releasing process 314
Allocated process 315
Allocated process 314
Allocated process 316
Allocated process 317
Releasing process 317
Allocated process 318
Allocated process 317
Allocated process 319
Allocated process 320
Allocated process 321
Allocated process 322
Allocated process 323
Allocated process 324
Allocated process 325
Allocated process 326
Allocated process 327
Allocated process 328
Allocated process 329
Allocated process 330
Allocated process 331
```



```
Allocated process 382
Allocated process 383
Allocated process 384
Allocated process 385
Allocated process 386
Releasing process 386
Allocated process 386
Allocated process 387
Allocated process 388
Allocated process 389
Releasing process 300
Releasing process 308
Releasing process 310
Releasing process 311
Releasing process 315
Releasing process 329
Releasing process 332
Releasing process 345
Releasing process 356
Releasing process 385
Releasing process 388
Releasing process 302
Releasing process 312
Releasing process 321
Releasing process 326
Releasing process 331
Releasing process 338
Releasing process 341
Releasing process 355
Releasing process 354
Releasing process 365
Releasing process 367
Releasing process 368
Releasing process 376
Releasing process 303
Releasing process 343
Releasing process 346
Releasing process 358
Releasing process 366
Releasing process 373
Releasing process 306
```

APPENDIX:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#include<pthread.h>
#include <stdbool.h>

#define MIN_PID 300
#define MAX_PID 5000
#define THREAD_COUNT 100

pthread_mutex_t mutex;

struct Pid{
    int PID;
    bool bitmap;
}*pId;


int allocate_pid(){
  int i=0 ;
  while(i<MAX_PID-MIN_PID+1)
  {
        if(pId[i].bitmap==0){
            pId[i].bitmap=1;
            return pId[i].PID;
        }
        i++;
  }
    return -1;
}
void release_pid(int pid){
    pId[pid-MIN_PID].bitmap=0;
}
int allocate_map(){
    int i=1;
    pId=(struct Pid *)calloc((MAX_PID-MIN_PID+1),sizeof(struct Pid));
    if(pId==NULL)
        return -1;
    pId[0].PID=MIN_PID;
    pId[0].bitmap=0;
    while(i<MAX_PID-MIN_PID+1)
    {
    pId[i].PID=(pId[i-1].PID)+1;
     pId[i].bitmap=0;
     i++;
    }
    return 1;
}
```

```
void *processStart(void *id)
{

        int pid,sleep;
        sleep=rand()%10;
        pthread_mutex_lock(&mutex);
        pid=allocate_pid();
        usleep(1000);
        pthread_mutex_unlock(&mutex);
        if(pid!=-1){
                printf("Allocated process %d \n",pid);
                sleep(sleep);
                printf("Releasing process %d \n",pid);
                release_pid(pid);
        }
        pthread_exit(NULL);
}
int main(){
  allocate_map();
  srand(time(NULL));
  long i=0;
  int ret=0;
  pthread_t thread[100];
  pthread_mutex_init(&mutex,NULL);
  while(i<THREAD_COUNT)
              {
        ret=pthread_create(&thread[i],NULL,processStart,(void *)(i+1));
        if(ret)
        {
       printf("Thread creation failed\n");
       exit(1);}
        i++;
  }
  pthread_exit(NULL);
  pthread_mutex_destroy(&mutex);
  return 0;
}
```