**Student Name:** Pratik Ramesh Dhule

**Student Id:** 11814018

**Email Address:** pratikdhule5@gmail.com

**GitHub Link:** <https://github.com/pratikdhule5/operatingsystem>

**Description:** Here I have designed a new synchronization primitive that will allow multiple processes to block on an event until some other process signals the event. When a process signals the event, all the process that are blocked on the event are unblocked .If no processes are blocked on an event when it is signaled ,then thesignal has no effect.

**Code snippet:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <pthread.h> /\* required for pthreads \*/

#include <semaphore.h> /\* required for semaphores \*/

/\* On Solaris, compile with cc -v -mt semaphore\_sample.c -lpthread -lrt

\* or gcc -Wall -Wextra semaphore\_sample.c -lpthread -lrt

\* On Mac OS X, compile with gcc -Wall -Wextra semaphore\_sample.c

\* On Linux, compile with gcc -Wall -Wextra semaphore\_sample.c -lpthread

\*

\* Note that all the compiler warnings from GCC are about printf format

\* mismatches. These are easy to fix, but we didn't feel like taking the

\* time, so the grade should be lowered some more.

\*

\* Run with a.out

\* a.out 1

\* a.out 2

\* a.out 3

\* a.out 4

\* a.out 5

\* a.out 6

\* a.out 7

\* The operand selects which method to use to end the threads and process.

\*/

/\* function prototype for Pthread creation (a start function) \*/

void \* thread\_func(void \* arg);

/\* example of data passed to and from a Pthread start function \*/

typedef struct thread\_data {

int in; /\* used as an input value \*/

int out; /\* used as an output value \*/

} thread\_data\_t;

/\* semaphore for shared static local variable in thread\_func()

\* see the explanation in thread\_func() comments

\*/

static sem\_t count\_sem;

/\* shared global variable, semaphore not necessary because of the way it is used,

\* but see the additional notes at the end of the file.

\*/

int quit = 0;

int main(int argc, char \* argv[])

{

pthread\_t tid[2]; /\* thread ID, to be set by pthread\_create() \*/

thread\_data\_t a[2];

thread\_data\_t \*b[2] = { NULL, NULL };

/\* &a[i] passed to thread i as argument by pthread\_create()

\* b[i] received from thread i as return value by pthread\_join()

\*/

int ret, v;

a[0].in = 2; a[0].out = 0;

a[1].in = 4; a[1].out = 0;

/\* initialize semaphore for use by thread\_func()

\* arg 1, pointer to semaphore

\* arg 2,

\* for concurrent threads in the same process, second argument should be 0

\* for concurrent processes, second argument should be nonzero

\* arg 3, allow at most one thread to have access

\*/

// Mac OS X does not actually implement sem\_init()

if (sem\_init(&count\_sem, 0, 1) == -1)

{ printf("sem\_init: failed: %s\n", strerror(errno)); }

/\* at this point, there is only one thread, and quit can be safely modified \*/

if (argc > 1) quit = atoi(argv[1]);

/\* after this point, quit is not changed, so it does not need semaphore protection \*/

pthread\_create(&tid[0], NULL, thread\_func, (void \*)&a[0]);

pthread\_create(&tid[1], NULL, thread\_func, (void \*)&a[1]);

printf("main: process id %d, thread id = %d\n", getpid(), pthread\_self());

printf("main: tid[0] = %d, tid[1] = %d\n", tid[0], tid[1]);

printf("main: &a[0] = 0x%08lx, &a[1] = 0x%08lx\n", &a[0], &a[1]);

printf("main: b[0] = 0x%08lx, b[1] = 0x%08lx\n", b[0], b[1]);

printf("main: a[0].in = %d, a[1].in = %d\n", a[0].in, a[1].in);

printf("main: a[0].out = %d, a[1].out = %d\n", a[0].out, a[1].out);

// Mac OS X does not actually implement sem\_getvalue()

if (sem\_getvalue(&count\_sem, &v) == -1)

{ printf("sem\_getvalue: failed: %s\n", strerror(errno)); }

else

{ printf("main: count\_sem value = %d\n", v); }

if (quit == 1) pthread\_exit(NULL);

if (quit == 2) exit(0);

if (quit == 3) pthread\_cancel(tid[1]);

pthread\_join(tid[0], (void \*\*)&b[0]);

if ((ret = pthread\_join(tid[1], (void \*\*)&b[1])) != 0)

{

printf("main: pthread\_join(tid[1],...) failed on return, %s\n", strerror(ret));

printf("main: b[1] will be reset from 0x%08x to 0x%08x\n", b[1], NULL);

b[1] = NULL;

}

else if (b[1] != &a[1])

{

printf("main: pthread\_join(tid[1],...) failed on address, %s\n", strerror(-(int)b[1]));

printf("main: b[1] will be reset from 0x%08x to 0x%08x\n", b[1], NULL);

b[1] = NULL;

}

printf("\n");

printf("main: process id %d, thread id = %d\n", getpid(), pthread\_self());

printf("main: tid[0] = %d, tid[1] = %d\n", tid[0], tid[1]);

printf("main: &a[0] = 0x%08lx, &a[1] = 0x%08lx\n", &a[0], &a[1]);

printf("main: b[0] = 0x%08lx, b[1] = 0x%08lx\n", b[0], b[1]);

printf("main: a[0].in = %d, a[1].in = %d\n", a[0].in, a[1].in);

printf("main: a[0].out = %d, a[1].out = %d\n", a[0].out, a[1].out);

if (b[0] == NULL && b[1] == NULL) {

printf("main: b[0] is NULL, b[1] is NULL\n");

} else if (b[0] == NULL) {

printf("main: b[0] is NULL, b[1]->in = %d, b[1]->out = %d\n", b[1]->in, b[1]->out);

} else if (b[1] == NULL) {

printf("main: b[0]->in = %d, b[0]->out = %d, b[1] is NULL\n", b[0]->in, b[0]->out);

} else {

printf("main: b[0]->in = %d, b[1]->in = %d\n", b[0]->in, b[1]->in);

printf("main: b[0]->out = %d, b[1]->out = %d\n", b[0]->out, b[1]->out);

}

if (quit == 3 && b[1] != NULL && b[1]->out < 0)

printf("main: thread %d returned b[1]->out = %d\n", tid[1], b[1]->out);

if (quit == 6) pthread\_exit(NULL);

if (quit == 7) exit(0);

return 0;

}

/\* standard thread function format for Pthreads \*/

void \* thread\_func(void \* arg)

{

/\* shared local variable, one instance among all threads using this function,

\* so a semaphore is needed to enforce mutual exclusion. Because the

\* semaphore must be initialized, only once, the initialization takes place

\* in the main thread, and the semaphore is declared globally. You should

\* be able to explain why the semaphore cannot safely be declared inside

\* thread\_func() and initialized from within the thread function itself.

\* This problem could be corrected with a C++ constructor for a semaphore

\* class.

\*/

static int count = 0;

/\* static sem\_t count\_sem; \*/

/\* local variable, one instance per thread using this function \*/

int s = 0, t, v;

/\* pointer to input and output struct \*/

thread\_data\_t \*p = (thread\_data\_t \*)arg;

/\* enforce mutual exclusion for access to count \*/

sem\_wait(&count\_sem); /\* start of critical section \*/

count++;

sem\_post(&count\_sem); /\* end of critical section \*/

s++;

/\* should this use of count be protected by the semaphore? \*/

printf("thread: process id %d, thread id = %d, p->in = %d, count = %d, s = %d\n",

getpid(), pthread\_self(), p->in, count, s);

// Mac OS X does not actually implement sem\_getvalue()

if (sem\_getvalue(&count\_sem, &v) == -1)

{ printf("sem\_getvalue: failed: %s\n", strerror(errno)); }

else

{ printf("thread: count\_sem value = %d\n", v); }

sleep(p->in);

sem\_wait(&count\_sem);

count++; t = count;

sem\_post(&count\_sem);

s++;

/\* because count was copied to t, no semaphore is needed here \*/

printf("thread: process id %d, thread id = %d, arg = %d, count = %d, s = %d\n",

getpid(), pthread\_self(), p->in, t, s);

if (quit == 4)

{

p->out = -(int)pthread\_self();

pthread\_exit(arg);

}

if (quit == 5) exit(0);

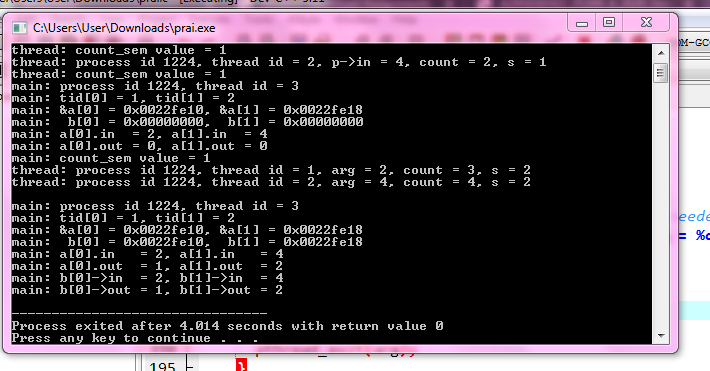
p->out = (int)pthread\_self();

return arg;

}

**GitHub Link:** <https://github.com/pratikdhule5/operatingsystem>

**Screenshot of output:**

****