Threads

a new abstraction of program execution

Disadvantage of MUTEX

- Deadlock can occur if:
  - Thread locking the same MUTEX twice
  - Thread1 holding lock on mutex1 wants to get lock on mutex2 while thread2 holding lock on mutex2 wants to get lock on mutex1
  - Solution: As much as possible use pthread_mutex_trylock function
- Only one thread is allowed to lock mutex but in some situations such as Reader-Write problem or Dinning-Philosopher problem multiple locks should be possible
  - Solution: Reader-Writer Locks and Semaphores

Example Solution without race-condition

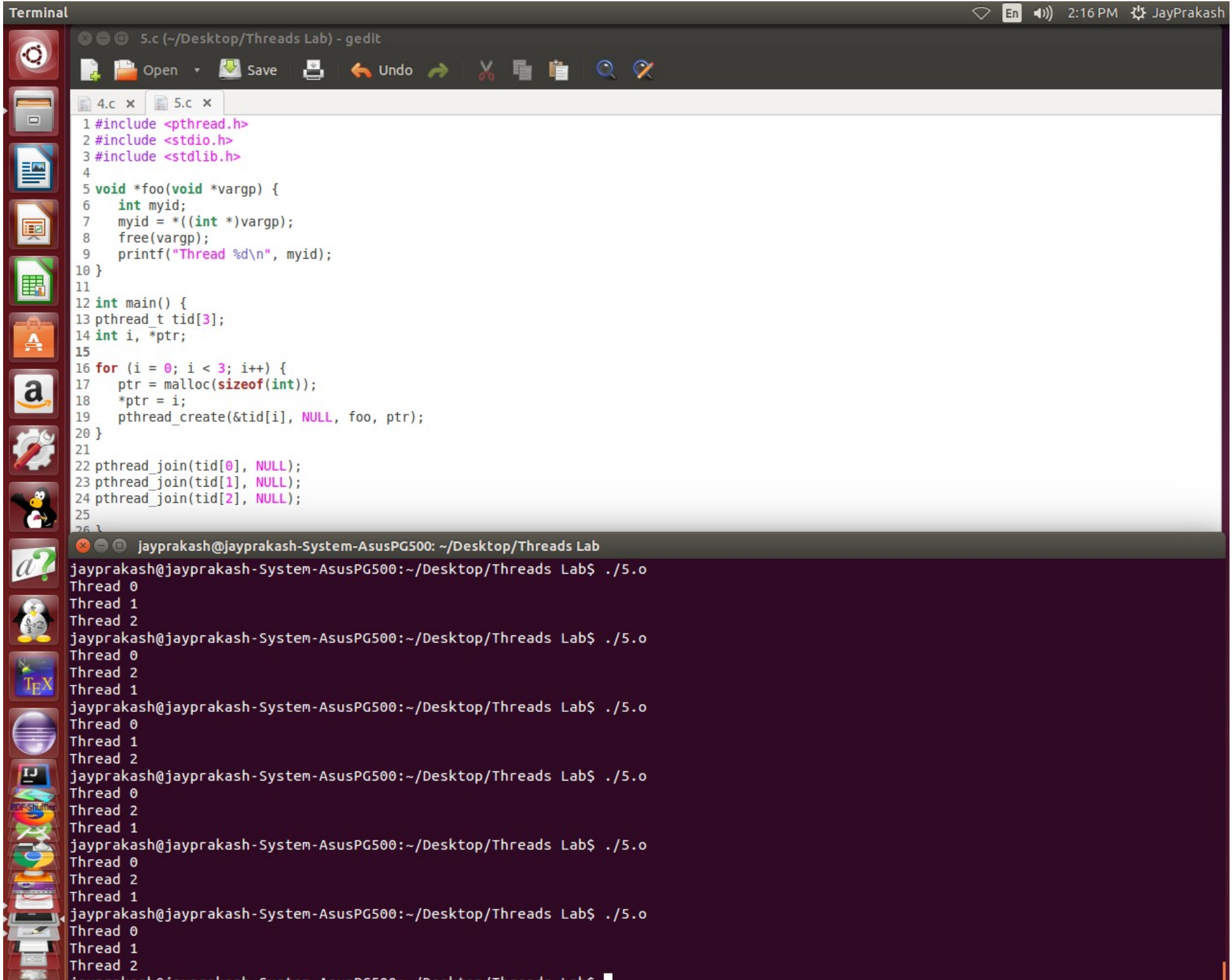
Q: Are there any race conditions in this code?

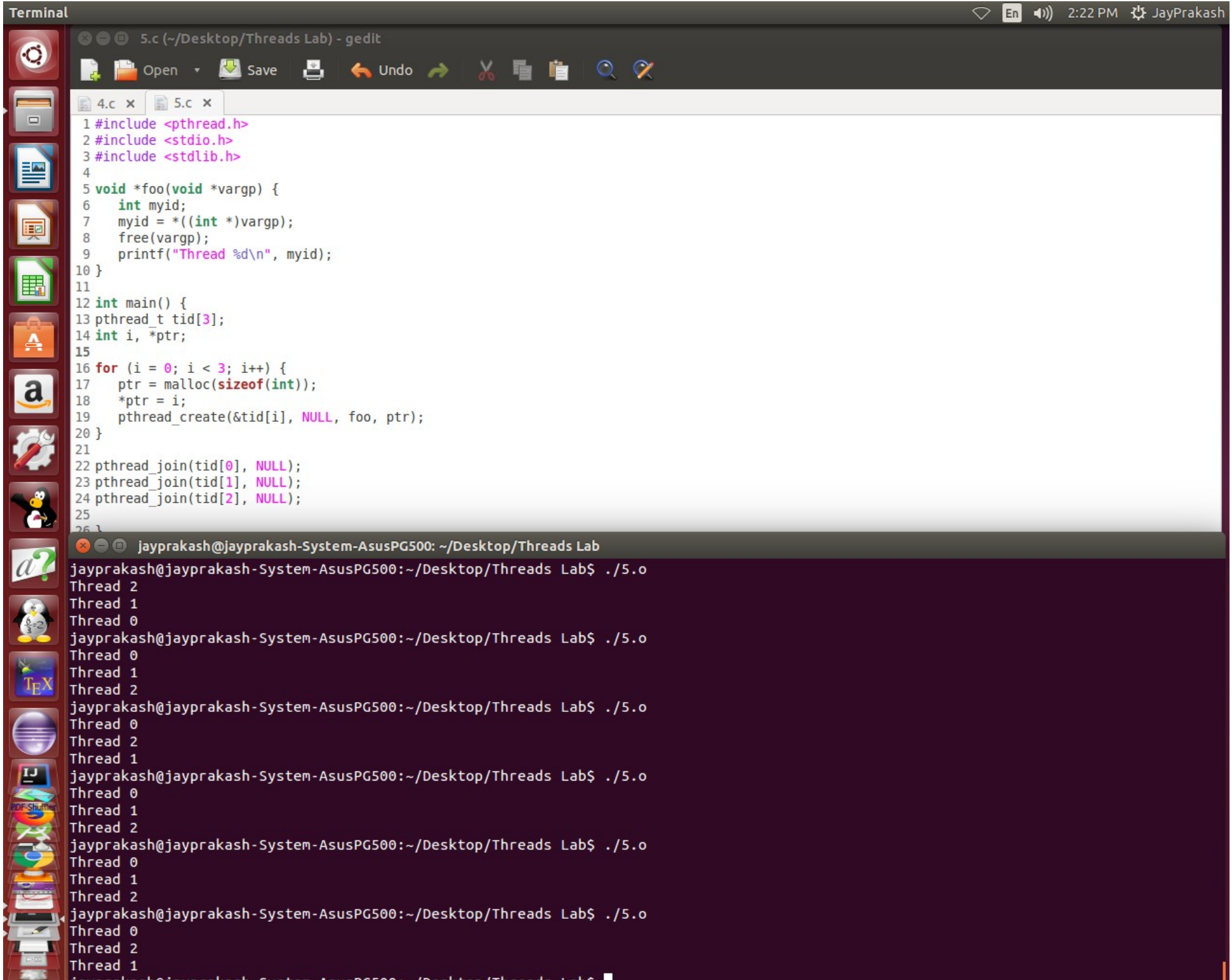
```
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    Free(vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i, *ptr;

    for (i = 0; i < 2; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, foo, ptr);
    }
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
}
```

No!


Terminal
En
2:16 PM
JayPrakash
5.c (~/Desktop/Threads Lab) - gedit
Open
Save
Undo
4.c
5.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void *foo(void *vargp) {
6 int myid;
7 myid = *((int *)vargp);
8 free(vargp);
9 printf("Thread %d\n", myid);
10 }
11
12 int main() {
13 pthread_t tid[3];
14 int i, *ptr;
15
16 for (i = 0; i < 3; i++) {
17 ptr = malloc(sizeof(int));
18 *ptr = i;
19 pthread_create(&tid[i], NULL, foo, ptr);
20 }
21
22 pthread_join(tid[0], NULL);
23 pthread_join(tid[1], NULL);
24 pthread_join(tid[2], NULL);
25
26 }
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 1
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 2
Thread 1
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 1
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 2
Thread 1
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 2
Thread 1
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 1
Thread 2


Terminal
En
2:22 PM
JayPrakash
5.c (~/Desktop/Threads Lab) - gedit
Open
Save
Undo
4.c
5.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void *foo(void *vargp) {
6 int myid;
7 myid = *((int *)vargp);
8 free(vargp);
9 printf("Thread %d\n", myid);
10 }
11
12 int main() {
13 pthread_t tid[3];
14 int i, *ptr;
15
16 for (i = 0; i < 3; i++) {
17 ptr = malloc(sizeof(int));
18 *ptr = i;
19 pthread_create(&tid[i], NULL, foo, ptr);
20 }
21
22 pthread_join(tid[0], NULL);
23 pthread_join(tid[1], NULL);
24 pthread_join(tid[2], NULL);
25
26 }
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 2
Thread 1
Thread 0
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 1
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 2
Thread 1
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 1
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 1
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./5.o
Thread 0
Thread 2
Thread 1

Question

- Outline an approach to avoid race conditions that does not use malloc/free

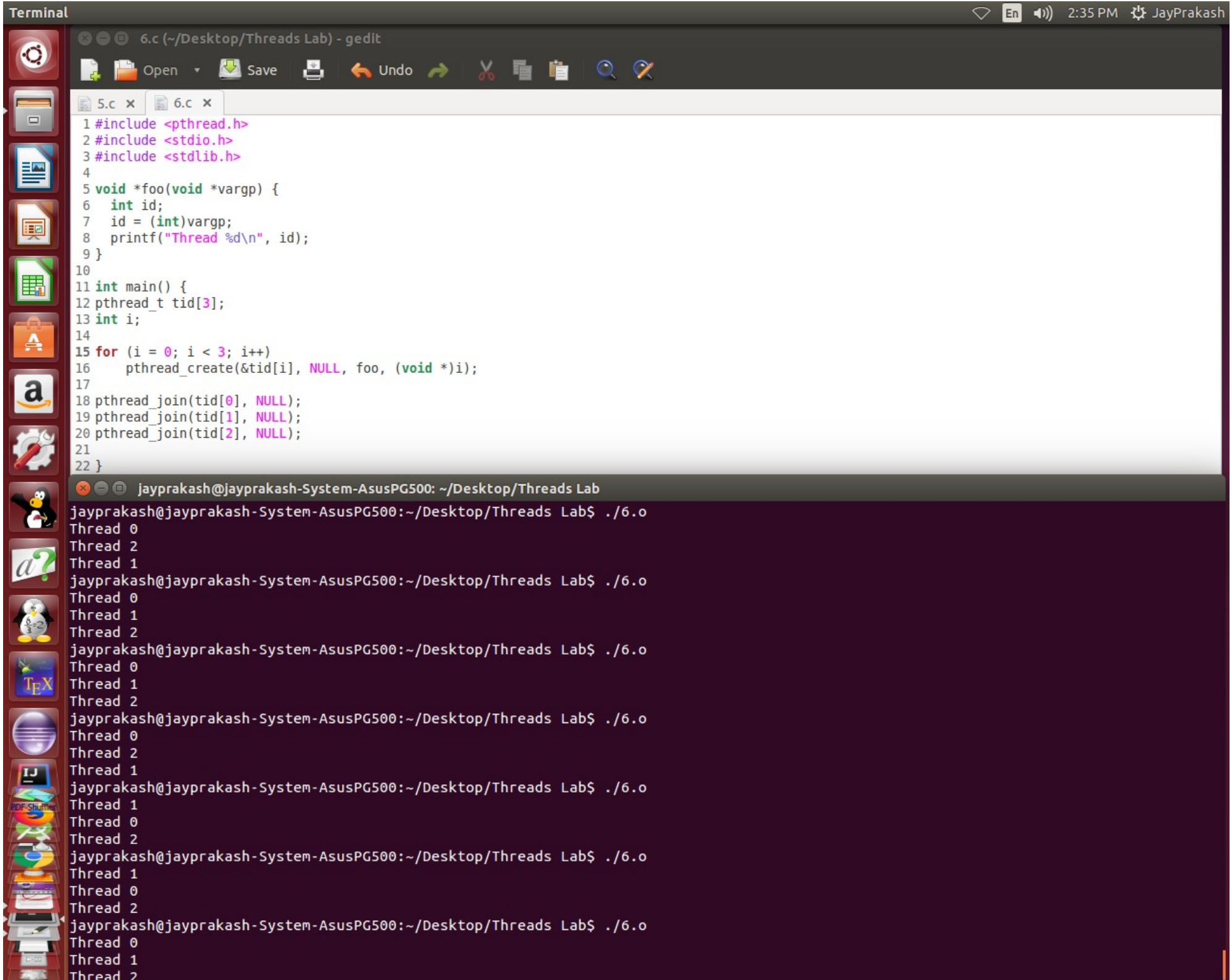
Example Solution without malloc()

Answer: Simply pass in the int directly!

```
void *foo(void *vargp) {
    int id;
    id = (int)vargp;
    printf("Thread %d\n", id);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], NULL, foo, (void *)i);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
}
```


Terminal
En
2:35 PM
JayPrakash
6.c (~/Desktop/Threads Lab) - gedit
Open
Save
Undo
5.c
6.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void *foo(void *vargp) {
6 int id;
7 id = (int)vargp;
8 printf("Thread %d\n", id);
9 }
10
11 int main() {
12 pthread_t tid[3];
13 int i;
14
15 for (i = 0; i < 3; i++)
16 pthread_create(&tid[i], NULL, foo, (void *)i);
17
18 pthread_join(tid[0], NULL);
19 pthread_join(tid[1], NULL);
20 pthread_join(tid[2], NULL);
21
22 }
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./6.o
Thread 0
Thread 2
Thread 1
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./6.o
Thread 0
Thread 1
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./6.o
Thread 0
Thread 1
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./6.o
Thread 0
Thread 2
Thread 1
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./6.o
Thread 1
Thread 0
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./6.o
Thread 1
Thread 0
Thread 2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./6.o
Thread 0
Thread 1
Thread 2

Previous example

- Pro: No added overhead due to malloc/free
- Con: Assumes that pointer datatype is at least bigger than size of int
  - May not be true on all systems

Threads are not hierarchical

- Unlike processes, a thread doesn’t have a parent or children – threads are peers
- Any thread can pthread_join any other thread

Threads are not hierarchical

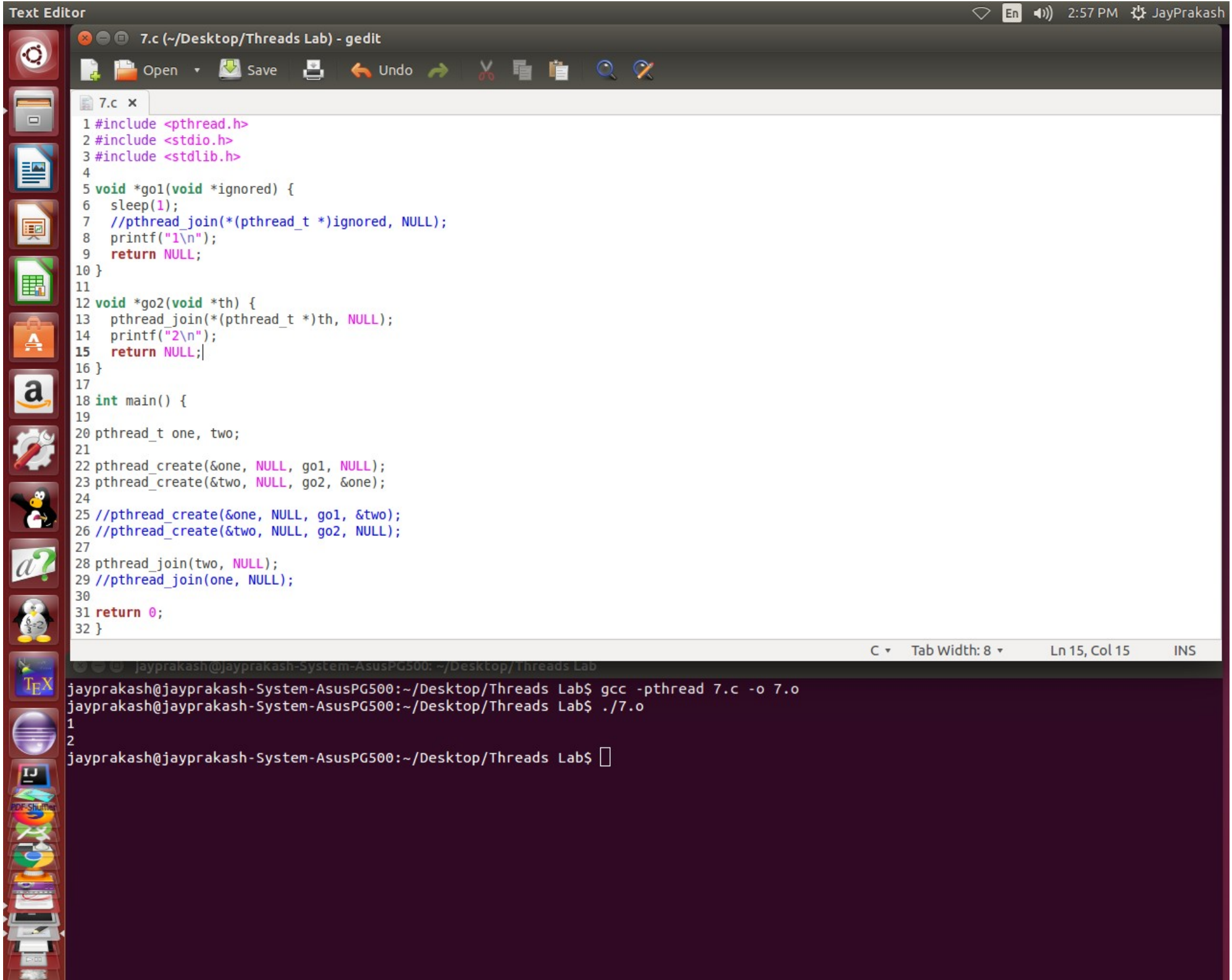
```
void *go1(void *ignored) {
    sleep(1);
    printf("1\n");
    return NULL;
}

void *go2(void *th) {
    pthread_join(*(pthread_t *)th, NULL);
    printf("2\n");
    return NULL;
}

int main() {
    pthread_t one, two;

    pthread_create(&one, NULL, go1, NULL);
    pthread_create(&two, NULL, go2, &one);
    pthread_join(two, NULL);
    return 0;
}
```

Prints 1 then 2


Text Editor
En
2:57 PM
JayPrakash
7.c (~/Desktop/Threads Lab) - gedit
Open
Save
Undo
7.c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *go1(void *ignored) {
sleep(1);
//pthread_join(*(pthread_t *)ignored, NULL);
printf("1\n");
return NULL;
}
void *go2(void *th) {
pthread_join(*(pthread_t *)th, NULL);
printf("2\n");
return NULL;
}
int main() {
pthread_t one, two;
pthread_create(&one, NULL, go1, NULL);
pthread_create(&two, NULL, go2, &one);
//pthread_create(&one, NULL, go1, &two);
//pthread_create(&two, NULL, go2, NULL);
pthread_join(two, NULL);
//pthread_join(one, NULL);
return 0;
}
C
Tab Width: 8
Ln 15, Col 15
INS
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ gcc -pthread 7.c -o 7.o
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./7.o
1
2
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$

Dining-Philosopher Problem

- Five philosophers spend their time eating and thinking.
- They are sitting in front of a round table with spaghetti served.
- There are five plates at the table and five chopsticks set between the plates.
- Eating the spaghetti requires the use of two chopsticks which the philosophers pick up one at a time.
- Philosophers do not talk to each other.
- Semaphore chopstick [5] initialized to 1

Semaphores

- Allow multiple locks
- Semaphore S → integer variable
- Modified by two operations → wait() and signal()
  - wait() – originally called P() for Dutch word “proberen” which means try
  - signal() – originally called V() for Dutch word “verhogen” which means increase

```
wait(S) {
        while S <= 0; // no-op
        S--;
}
signal(S) {
        S++;
}
```

Note wait() and signal() are atomic operation

Thread Synchronization using Semaphores

```
//Thread1:
int t;
wait(sem)
sum = sum + x;
t = sum;
…
signal(sem);
```

```
//Thread2:
int t;
wait(sem)
sum = sum + y;
t = sum;
…
signal(sem);
```

Dining-Philosopher Problem: Solution

The structure of Philosopher i:

```
do {
  wait ( chopstick[i] ); //lock
  wait ( chopStick[ (i + 1) % 5] ); //lock
  // eat
  signal ( chopstick[i] ); //unlock
  signal (chopstick[ (i + 1) % 5] );
//unlock
  // think
} while (true) ;
```

- Problem is deadlock : when all philosopher decides to eat at the same time each will pick up one of the 2 chopsticks they need and wait for the other
- Solution: Allow to lock if both chopsticks are available at the same time
  - Lock the 1st chopstick
  - For 2nd chopstick, check if it can be locked otherwise release the 1st one

POSIX: Semaphores

- creating a semaphore:

int sem_init(sem_t *sem, int pshared, unsigned int value);

  - initializes a semaphore object pointed to by sem to integer value of “value”
  - pshared is a sharing option; a value of 0 means the semaphore is local to the calling process i.e. shared by all threads of the same process; a +ve number indicates it can be shared across multiple processes using shared memory instructions

- terminating a semaphore:

int sem_destroy(sem_t *sem);

  - frees the resources allocated to the semaphore sem
  - usually called after pthread_join()
  - an error will occur if a semaphore is destroyed for which a thread is waiting

POSIX: Semaphores

- semaphore control:

int sem_post(sem_t *sem); → same as signal()

sem_post atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)

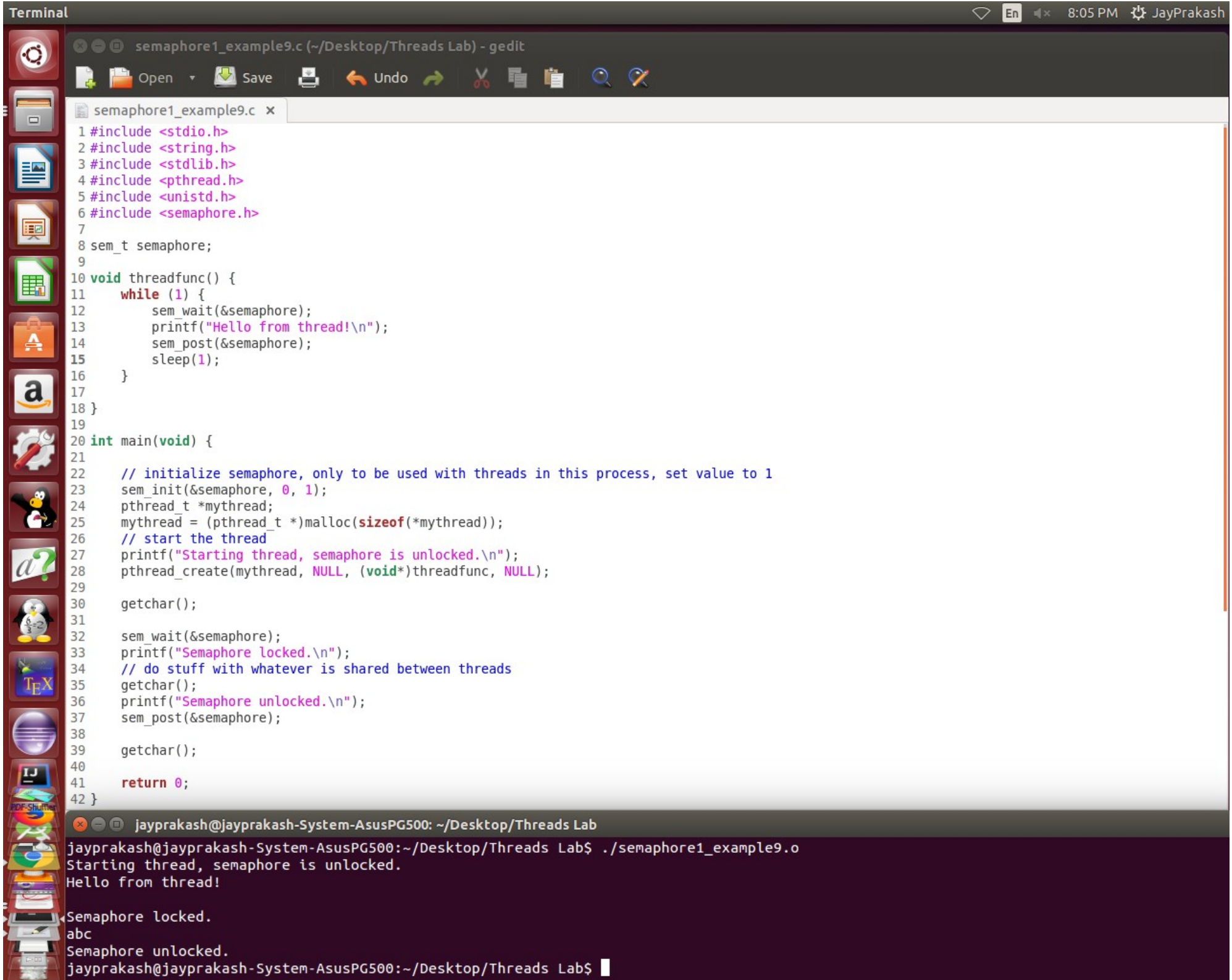
int sem_wait(sem_t *sem);

sem_wait atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Semaphore Example

semaphore1.c

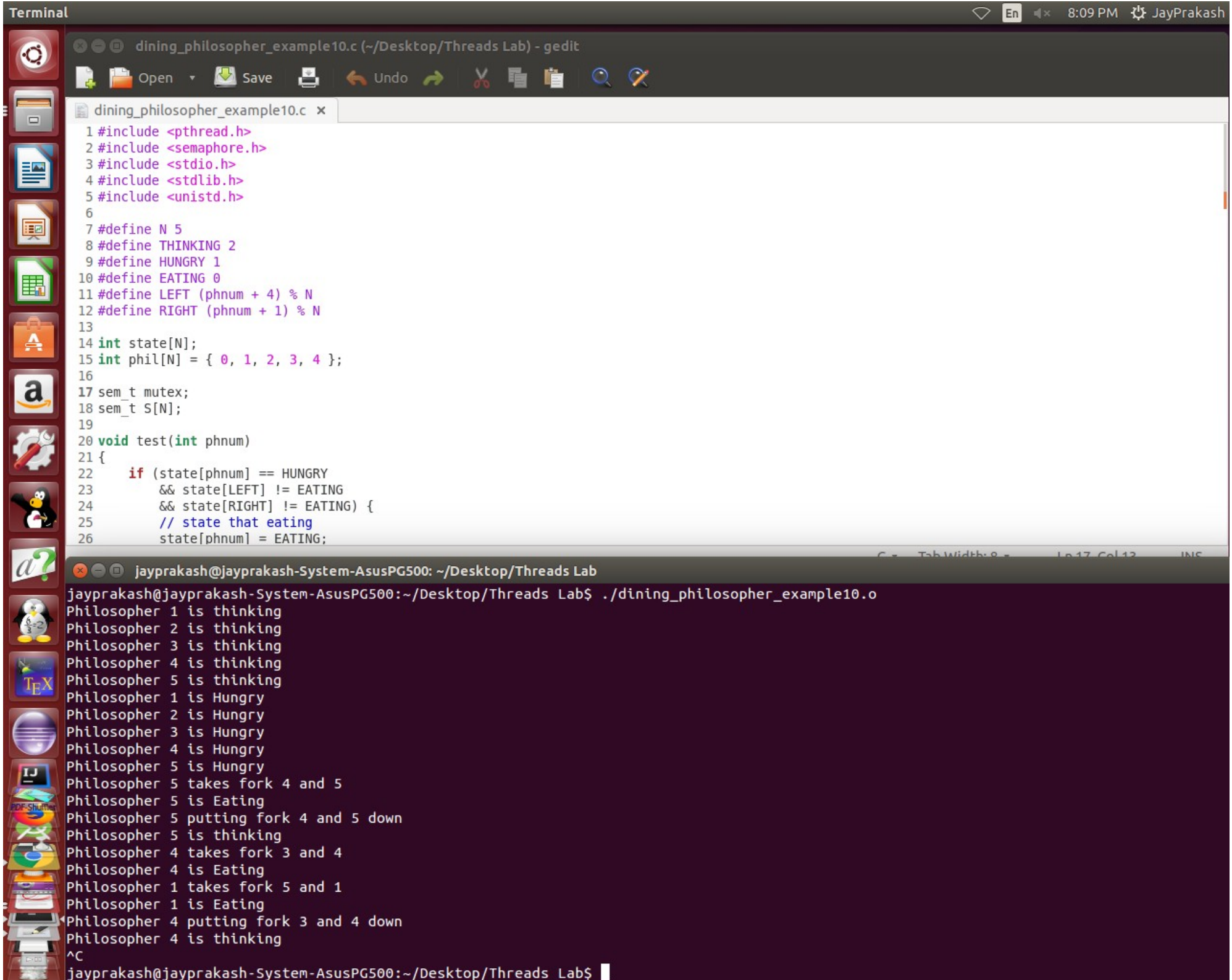
```
$ ./semaphore1.out
Starting thread, semaphore is unlocked.
Hello from thread!
Hello from thread!
Hello from thread!
Hello from thread!
Semaphore locked.
Semaphore unlocked.
Hello from thread!
Hello from thread!
Hello from thread!
$
```


Terminal
En
8:05 PM
JayPrakash
semaphore1_example9.c (~/Desktop/Threads Lab) - gedit
Open
Save
Undo
semaphore1_example9.c
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <pthread.h>
5 #include <unistd.h>
6 #include <semaphore.h>
7
8 sem_t semaphore;
9
10 void threadfunc() {
11 while (1) {
12 sem_wait(&semaphore);
13 printf("Hello from thread!\n");
14 sem_post(&semaphore);
15 sleep(1);
16 }
17
18 }
19
20 int main(void) {
21
22 // initialize semaphore, only to be used with threads in this process, set value to 1
23 sem_init(&semaphore, 0, 1);
24 pthread_t *mythread;
25 mythread = (pthread_t *)malloc(sizeof(*mythread));
26 // start the thread
27 printf("Starting thread, semaphore is unlocked.\n");
28 pthread_create(mythread, NULL, (void*)threadfunc, NULL);
29
30 getchar();
31
32 sem_wait(&semaphore);
33 printf("Semaphore locked.\n");
34 // do stuff with whatever is shared between threads
35 getchar();
36 printf("Semaphore unlocked.\n");
37 sem_post(&semaphore);
38
39 getchar();
40
41 return 0;
42 }
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./semaphore1_example9.o
Starting thread, semaphore is unlocked.
Hello from thread!
Semaphore locked.
abc
Semaphore unlocked.
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$

Dining-Philosopher Problem: Solution

dining-philosopher.c

Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
…….


Terminal
8:09 PM
JayPrakash
dining_philosopher_example10.c (~/Desktop/Threads Lab) - gedit
Open
Save
Undo
dining_philosopher_example10.c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N
int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];
void test(int phnum)
{
if (state[phnum] == HUNGRY
&& state[LEFT] != EATING
&& state[RIGHT] != EATING) {
// state that eating
state[phnum] = EATING;
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./dining_philosopher_example10.o
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
^C
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$