TU Dublin
Operating Systems
COMP 3602

Lecture 6

Process&Thread
SYNCHRONIZATION

Software tools to restrict access to global variables and to provide synchronization between threads

- Mutexes:
  - provide exclusive access to code segments/variables with the use of locks (mutual exclusion)

- Semaphores:
  - provide exclusive access to code segments/variables with the use of locks (mutual exclusion)
  - provide synchronization and communication between threads that share a mutex

PROBLEM: unrestricted access to global variables!

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable



void * inc(void * arg){
   for(long i=0;i<100000000;i++)

      balance++;

   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++)

      balance--;

   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

UNDESIRED OUTPUTS:

```
Value of balance is: -986105

Value of balance is: -771273

Value of balance is: 2822774

Value of balance is: 2791508
```

PROBLEM: unrestricted access to global variables!

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){

           wordcount++;

   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }

   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

UNDESIRED OUTPUTS:

```
Number of characters in both files: 1295

Number of characters in both files: 1302

Number of characters in both files: 1280
```

Race condition

A **race condition** is the behaviour of a software system where the output is dependent on the sequence or timing of other unpredicible events (e.g. threads that may be interrupted at any time)

Two threads want to increment the value of a global integer variable by one:

Ideal sequence of operations:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Two threads running without synchronization:

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Maintaining critical sections

- A **Critical Section** is a code segment that accesses shared variables.
- Processes, threads can execute concurrently
  - may be interrupted at any time, partially completing execution
- Each process or thread has its critical section where it may be modifying shared variables (e.g. increment, decrement)
- At a given point of time, only one process must be executing its **critical section**.
- Maintaining critical sections requires mechanisms to ensure the orderly execution of concurrent processes

Synchronization using Mutex

Synchronization using Mutex

- A mutex is a MUTual Exclusion software tool for protecting shared variables from concurrent modifications in the critical sections
- A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). It can never be owned by two different threads simultaneously.
- A thread attempting to lock a mutex that is already locked by another thread is suspended until the owner unlocks the mutex
- Linux guarantees that race condition do not occur among threads attempting to lock a mutex.

Mutex initialization

- Static method of creating “mut” object (default Mutex attributes)

```
int pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

- Dynamic method of creating “mut” object (Mutex attributes specified in “attr” ; If “attr” is NULL then default Mutex attributes)

```
pthread_mutex_t mut;
int pthread_mutex_init(&mut, &attr);
```

Locking, unlocking and destroying mutex

- The lock() call will lock the *p_thread_mutex_t* object referenced by *mptr*. If mutex is already locked, the calling thread blocks until the mutex is unlocked.
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`

- The unlock() call will release the mutex object. If there are threads block on the mutex object when unlock() one of them will acquire the mutex.
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

- The destry() call will destroy the mutex

  - pthread_mutex_destroy(pthread_mutex_t *mutex);

PROBLEM: unrestricted access to global variables!

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable



void * inc(void * arg){
   for(long i=0;i<100000000;i++)

      balance++;


   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++)

      balance--;


   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

UNDESIRED OUTPUTS:

```
Value of balance is: -986105

Value of balance is: -771273

Value of balance is: 2822774

Value of balance is: 2791508
```

Handling the Critical Section with mutex (balance)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable



void * inc(void * arg){
   for(long i=0;i<100000000;i++)

      balance++;


   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++)

      balance--;


   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

Handling the Critical Section with mutex (balance)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable



void * inc(void * arg){
   for(long i=0;i<100000000;i++)

      balance++;


   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++)

      balance--;


   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

Handling the Critical Section with mutex (balance)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;


void * inc(void * arg){
   for(long i=0;i<100000000;i++)

      balance++;


   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++)

      balance--;


   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

Handling the Critical Section with mutex (balance)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;


void * inc(void * arg){
   for(long i=0;i<100000000;i++){
      pthread_mutex_lock(&mut);
      balance++;
      pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++)

      balance--;


   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

Handling the Critical Section with mutex (balance)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;


void * inc(void * arg){
   for(long i=0;i<100000000;i++){
      pthread_mutex_lock(&mut);
      balance++;
      pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++){
      pthread_mutex_lock(&mut);
      balance--;
      pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

Handling the Critical Section with mutex (balance)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

long balance = 0; //global variable
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;


void * inc(void * arg){
   for(long i=0;i<100000000;i++){
      pthread_mutex_lock(&mut);
      balance++;
      pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
};

void * dec(void * arg){
   for(long j=0;j<100000000;j++){
      pthread_mutex_lock(&mut);
      balance--;
      pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
};
```

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

OUTPUT:

```
Value of balance is: 0
```

PROBLEM: unrestricted access to global variables!

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){

           wordcount++;

   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }

   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

UNDESIRED OUTPUTS:

```
Number of characters in both files: 1295

Number of characters in both files: 1302

Number of characters in both files: 1280
```

Handling the Critical Section with mutex (characters)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){

           wordcount++;

   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }

   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

Handling the Critical Section with mutex (characters)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

pthread_mutex_t mut;
int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){

           wordcount++;

   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }

   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

Handling the Critical Section with mutex (characters)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

pthread_mutex_t mut;
int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){

           wordcount++;

   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }
   pthread_mutex_init(&mut,NULL);
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

Handling the Critical Section with mutex (characters)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

pthread_mutex_t mut;
int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){
           pthread_mutex_lock(&mut);
           wordcount++;
           pthread_mutex_unlock(&mut);
   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }
   pthread_mutex_init(&mut,NULL);
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

Handling the Critical Section with mutex (characters)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

pthread_mutex_t mut;
int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){
           pthread_mutex_lock(&mut);
           wordcount++;
           pthread_mutex_unlock(&mut);
   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }
   pthread_mutex_init(&mut,NULL);
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

OUTPUT:

$ ./racemutex2 race1.c race2.c

Number of characters in both files: 1302

Handling the Critical Section with mutex (characters)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

pthread_mutex_t mut;
int wordcount = 0;

void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);

   while((read(fd, &ch, 1)) != 0){
           pthread_mutex_lock(&mut);
           wordcount++;
           pthread_mutex_unlock(&mut);
   }

   close(fd);
   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[]){

   if(argc != 3){
      printf("Input two text filenames.\n");
      exit(1);
   }
   pthread_mutex_init(&mut,NULL);
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);

   printf("Number of characters in both files: %d\n", wordcount);

   return 0;
}
```

OUTPUT:

$ ./racemutex2 race1.c race2.c
Number of characters in both files: 1302

$ wc -c race1.c race2.c
528 race1.c
774 race2.c
1302 total

Synchronization using Semaphores

Introduction to Semaphores

- Semaphores are software tools (i.e. non-negative integers) shared between processes and threads. They are used to solve the critical section problem and to achieve process (or thread) synchronization in the multiprocessing (or multithreading) environment.

  Semaphores can be initialized with any non-negative value. After initialization semaphores can be modified only by calling two functions:

  sem_post() - increments their integer value

  sem_wait() - decrement their integer value

  When any process/thread attempts to decrement the semaphore, and the semaphore has the value zero then that process/thread becomes blocked (waiting)

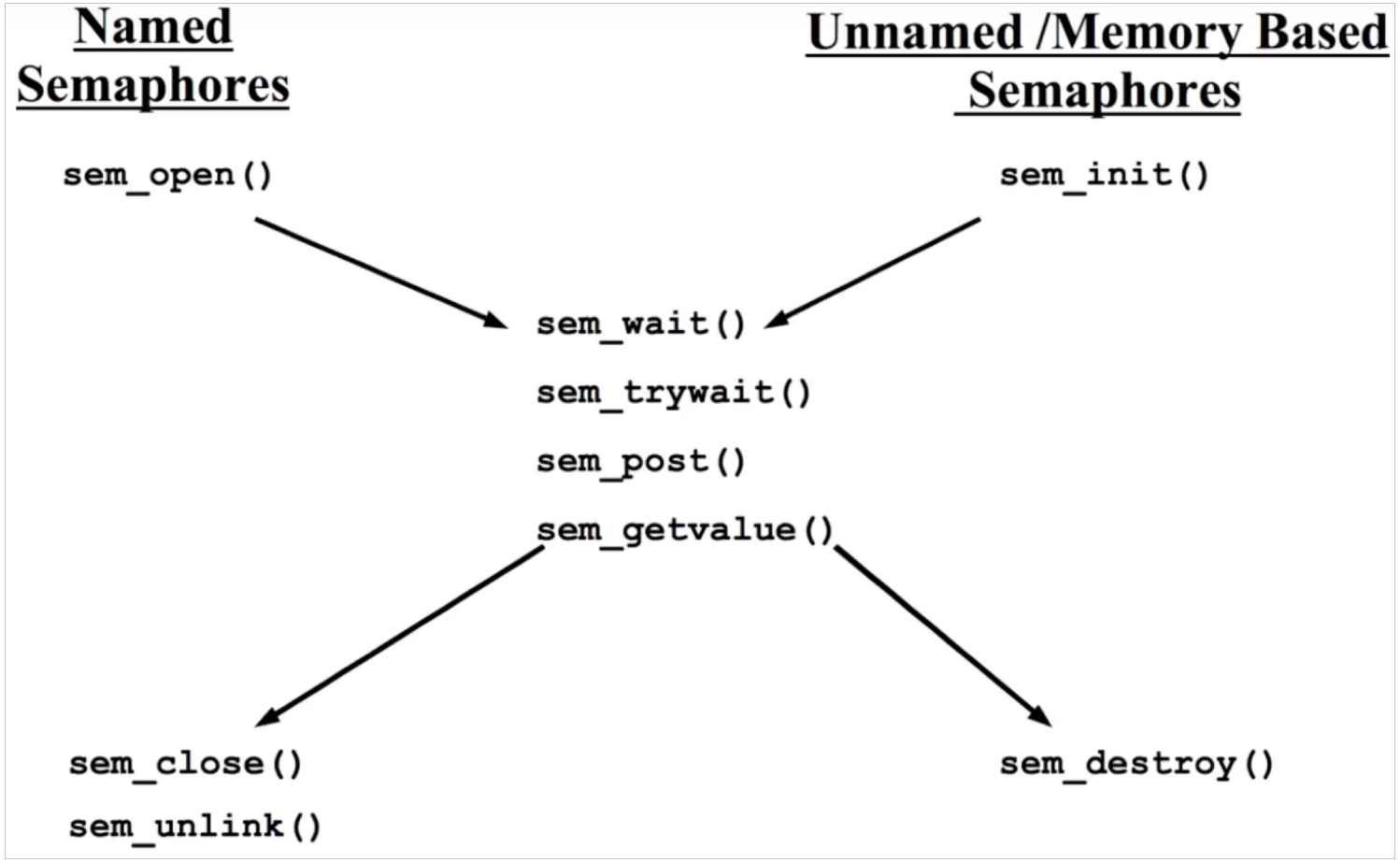
  When any process/thread increments that semaphore then the blocked (waiting) process/threads gets unblocked

Semaphores vs. Mutexes

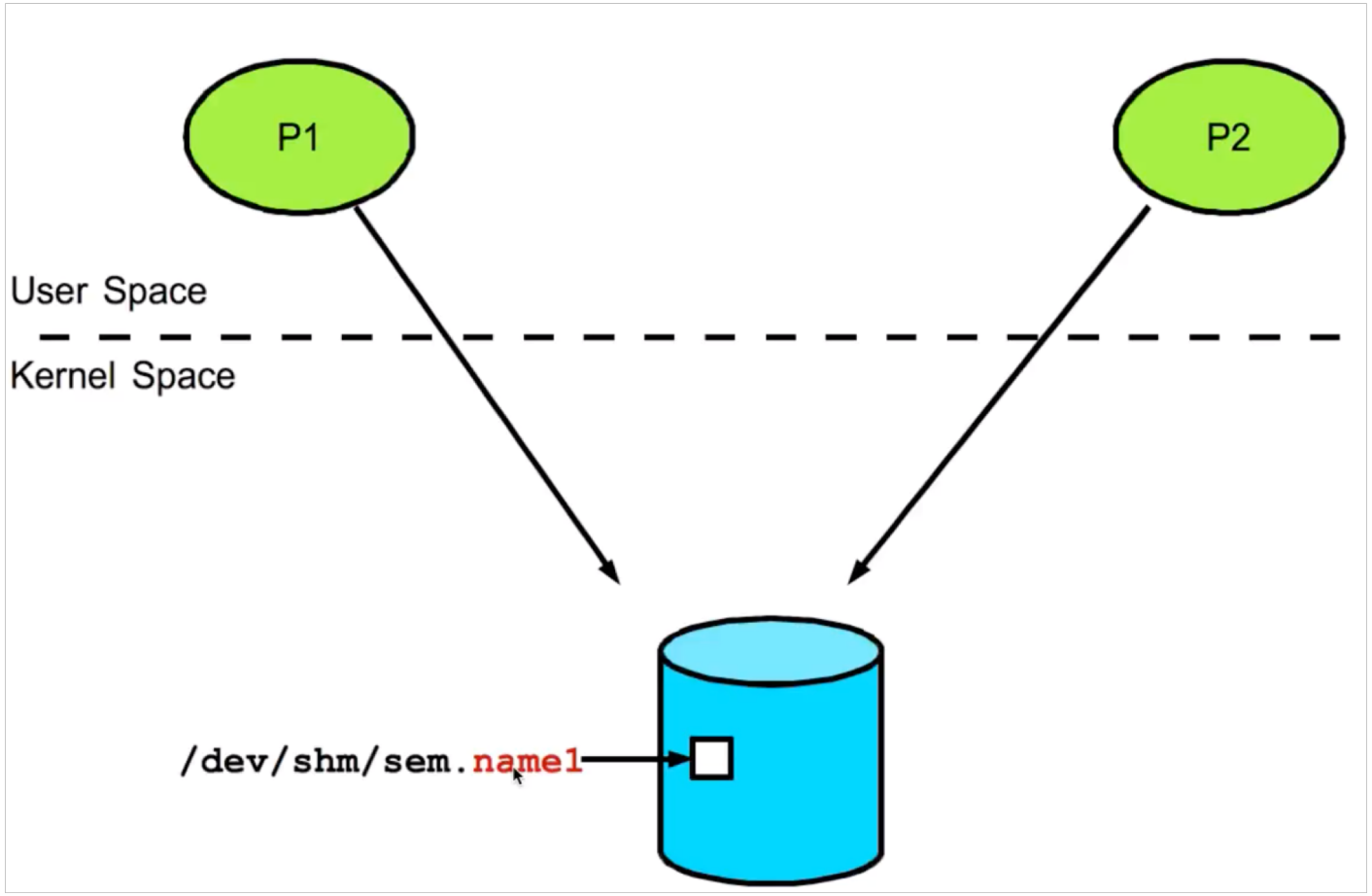
- Semaphores can have many values (i.e. non-negative integers).
- Used to achieve mutual exclusion and provide synchronization
- A semaphore can be modified by any process/thread
- Semaphores are optimized for both: locking and synchronization

- Mutexes can have only two values: 0 or 1
- Used to achieve mutual exclusion
- A mutex must always be unlocked by the process that locked the mutex
- Mutexes are optimized for locking

Implementations of POSIX semaphores



Creating a named semaphore



Creating a named semaphore

```
sem_t *mutex;
char* name = "/name1";
mutex = sem_open(name, O_CREAT, 0666, 0);
```

- The ***sem_open()*** call creates a new semaphore (or opens existing) identified by its "name" (i.e. first argument containing an initial slash).
- The second and third arguments, i.e. ***O_CREAT*** represents creating a new semaphore with specified ***read/write/execute*** permisstions
- The fourth argument specifies the initial value
- The return value (mutex) is a pointer to the semaphore

Incrementing and Decrementing Semaphores

```
int sem_wait(mutex);
int sem_post(mutex);
```

- The ***sem_wait(mutex)*** call decrements the semaphore. If greater than zero, then the decrement proceeds. If zero, then the call ***sem_wait()*** call blocks the decrementing process/thread until the value of semaphore rises above zero.

- The ***sem_post(mutex)*** call increments the semaphore. If greater than zero, the process/thread blocked in a ***sem_wait()*** call will be woken up.

- On success both the functions return 0. On error a -1 is returned.

Closing and removing semaphores

```
sem_close(mutex);
sem_unlink(name);
```

- The ***sem_close(mutex)*** call closes the semaphore. A semaphore is automatically closed on process/thread termination.

- The ***sem_unlink(name)*** removes the semaphore identified by its “name” from the system.

- Using Linux shell you can use the ***rm*** command to delete the related file in the ***/dev/shm*** directory

PROBLEM:
unsynchronized access to shared memory!

PROBLEM: unsynchronized access to shared memory!

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
long *balance;
void inc(){  //inc() function
long i;
   for(i=0;i<10000000;i++){

      *balance = *balance + 1;

   }
}
void dec(){  //dec() function
long j;
   for(j=0;j<10000000;j++){

      *balance = *balance - 1;

    }
}
```

```
int main(){
   int shm_id1=shmget(111, 8, 0777|IPC_CREAT);
   balance = (long*)shmat(shm_id1, NULL, 0); //attaching balance var.
   *balance=0;   //initializing balance value
```

```
int cpid = fork();
   if (cpid == 0){
       inc(); // calling inc() function
       shmdt(balance); //detaching balance

       exit(0);
   }
   else{  //parent process
       dec(); //calling dec() function
       wait(NULL);
       printf("Value of balance is: %ld\n", *balance);
       shmdt(balance); //detaching balance

       return 0;
   }
}
```

UNDESIRED OUTPUT:

Value of balance is: -2362984

Handling the Critical Section (shared memory)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>



long *balance;
void inc(){  //inc() function
long i;
   for(i=0;i<10000000;i++){

      *balance = *balance + 1;

   }
}
void dec(){  //dec() function
long j;
   for(j=0;j<10000000;j++){

      *balance = *balance - 1;

    }
}
```

```
int main(){
   int shm_id1=shmget(111, 8, 0777|IPC_CREAT);
   balance = (long*)shmat(shm_id1, NULL, 0); //attaching balance var.
   *balance=0;   //initializing balance value



int cpid = fork();
   if (cpid == 0){
       inc(); // calling inc() function
       shmdt(balance); //detaching balance

       exit(0);
   }
   else{  //parent process
       dec(); //calling dec() function
       wait(NULL);
       printf("Value of balance is: %ld\n", *balance);
       shmdt(balance); //detaching balance

       return 0;
   }
}
```

Handling the Critical Section (shared memory)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
sem_t *mutex;
long *balance;
void inc(){  //inc() function
long i;
   for(i=0;i<10000000;i++){

      *balance = *balance + 1;

   }
}
void dec(){  //dec() function
long j;
   for(j=0;j<10000000;j++){

      *balance = *balance - 1;

    }
}
```

```
int main(){
   int shm_id1=shmget(111, 8, 0777|IPC_CREAT);
   balance = (long*)shmat(shm_id1, NULL, 0); //attaching balance var.
   *balance=0;   //initializing balance value



int cpid = fork();
   if (cpid == 0){
       inc(); // calling inc() function
       shmdt(balance); //detaching balance

       exit(0);
   }
   else{  //parent process
       dec(); //calling dec() function
       wait(NULL);
       printf("Value of balance is: %ld\n", *balance);
       shmdt(balance); //detaching balance

       return 0;
   }
}
```

Handling the Critical Section (shared memory)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
sem_t *mutex;
long *balance;
void inc(){  //inc() function
long i;
   for(i=0;i<10000000;i++){

      *balance = *balance + 1;

   }
}
void dec(){  //dec() function
long j;
   for(j=0;j<10000000;j++){

      *balance = *balance - 1;

    }
}
```

```
int main(){
   int shm_id1=shmget(111, 8, 0777|IPC_CREAT);
   balance = (long*)shmat(shm_id1, NULL, 0); //attaching balance var.
   *balance=0;   //initializing balance value

   //creating a named semaphore
   char *name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0777, 1);

int cpid = fork();
   if (cpid == 0){
       inc(); // calling inc() function
       shmdt(balance); //detaching balance

       exit(0);
   }
   else{  //parent process
       dec(); //calling dec() function
       wait(NULL);
       printf("Value of balance is: %ld\n", *balance);
       shmdt(balance); //detaching balance

       return 0;
   }
}
```

Handling the Critical Section (shared memory)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
sem_t *mutex;
long *balance;
void inc(){  //inc() function
long i;
   for(i=0;i<10000000;i++){
      sem_wait(mutex);
      *balance = *balance + 1;
      sem_post(mutex);
   }
}
void dec(){  //dec() function
long j;
   for(j=0;j<10000000;j++){
      sem_wait(mutex);
      *balance = *balance - 1;
      sem_post(mutex);
    }
}
```

```
int main(){
   int shm_id1=shmget(111, 8, 0777|IPC_CREAT);
   balance = (long*)shmat(shm_id1, NULL, 0); //attaching balance var.
   *balance=0;   //initializing balance value

   //creating a named semaphore
   char *name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0777, 1);

int cpid = fork();
   if (cpid == 0){
       inc(); // calling inc() function
       shmdt(balance); //detaching balance

       exit(0);
   }
   else{  //parent process
       dec(); //calling dec() function
       wait(NULL);
       printf("Value of balance is: %ld\n", *balance);
       shmdt(balance); //detaching balance

       return 0;
   }
}
```

Handling the Critical Section (shared memory)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
sem_t *mutex;
long *balance;
void inc(){  //inc() function
long i;
   for(i=0;i<10000000;i++){
      sem_wait(mutex);
       *balance = *balance + 1;
      sem_post(mutex);
   }
}
void dec(){  //dec() function
long j;
   for(j=0;j<10000000;j++){
      sem_wait(mutex);
      *balance = *balance - 1;
      sem_post(mutex);
    }
}
```

```
int main(){
   int shm_id1=shmget(111, 8, 0777|IPC_CREAT);
   balance = (long*)shmat(shm_id1, NULL, 0); //attaching balance var.
   *balance=0;   //initializing balance value

   //creating a named semaphore
   char *name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0777, 1);

int cpid = fork();
   if (cpid == 0){
       inc(); // calling inc() function
       shmdt(balance); //detaching balance
       sem_close(mutex);
       exit(0);
   }
   else{  //parent process
       dec(); //calling dec() function
       wait(NULL);
       printf("Value of balance is: %ld\n", *balance);
       shmdt(balance); //detaching balance
       sem_close(mutex);
       sem_unlink(name);
       return 0;
   }
}
```

Handling the Critical Section (shared memory)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
sem_t *mutex;
long *balance;
void inc(){  //inc() function
long i;
   for(i=0;i<10000000;i++){
      sem_wait(mutex);
      *balance = *balance + 1;
      sem_post(mutex);
   }
}
void dec(){  //dec() function
long j;
   for(j=0;j<10000000;j++){
      sem_wait(mutex);
      *balance = *balance - 1;
      sem_post(mutex);
    }
}
```

```
int main(){
   int shm_id1=shmget(111, 8, 0777|IPC_CREAT);
   balance = (long*)shmat(shm_id1, NULL, 0); //attaching balance var.
   *balance=0;   //initializing balance value

   //creating a named semaphore
   char *name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0777, 1);

int cpid = fork();
   if (cpid == 0){
       inc(); // calling inc() function
       shmdt(balance); //detaching balance
       sem_close(mutex);
       exit(0);
   }
   else{  //parent process
       dec(); //calling dec() function
       wait(NULL);
       printf("Value of balance is: %ld\n", *balance);
       shmdt(balance); //detaching balance
       sem_close(mutex);
       sem_unlink(name);
       return 0;
   }
}
```

OUTPUT:

```
Value of balance is: 0
```

PROBLEM:
unsynchronized access to global variable!

PROBLEM: unsynchronized access to global variable!

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



long balance = 0;//global variable


void * inc(void * arg){
long i;
   for(i=0;i<10000000;i++){

      balance++;

   }
   pthread_exit(NULL);
}

void * dec(void * arg){
long j;
   for(j=0;j<10000000;j++){

      balance--;

    }
   pthread_exit(NULL);
}
```

```
int main(){


   pthread_t t1, t2;

   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);



   printf("Value of balance is :%ld\n", balance);

return 0;
}
```

UNDESIRED OUTPUT:

Value of balance is: -2362984

Handling the Critical Section (global variable)

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



long balance = 0;//global variable


void * inc(void * arg){
long i;
   for(i=0;i<10000000;i++){

      balance++;

   }
   pthread_exit(NULL);
}

void * dec(void * arg){
long j;
   for(j=0;j<10000000;j++){

      balance--;

    }
   pthread_exit(NULL);
}
```

```
int main(){


   pthread_t t1, t2;

   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);



   printf("Value of balance is :%ld\n", balance);

return 0;
}
```

Handling the Critical Section (global variable)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
long balance = 0;//global variable
sem_t *mutex;

void * inc(void * arg){
long i;
   for(i=0;i<10000000;i++){

      balance++;

   }
   pthread_exit(NULL);
}

void * dec(void * arg){
long j;
   for(j=0;j<10000000;j++){

      balance--;

    }
   pthread_exit(NULL);
}
```

```
int main(){


   pthread_t t1, t2;

   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);



   printf("Value of balance is :%ld\n", balance);

return 0;
}
```

Handling the Critical Section (global variable)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
long balance = 0;//global variable
sem_t *mutex;

void * inc(void * arg){
long i;
   for(i=0;i<10000000;i++){

      balance++;

   }
   pthread_exit(NULL);
}

void * dec(void * arg){
long j;
   for(j=0;j<10000000;j++){

      balance--;

    }
   pthread_exit(NULL);
}
```

```
int main(){
   char* name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0666, 1);
   pthread_t t1, t2;

   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);



   printf("Value of balance is :%ld\n", balance);

return 0;
}
```

Handling the Critical Section (global variable)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
long balance = 0;//global variable
sem_t *mutex;

void * inc(void * arg){
long i;
   for(i=0;i<10000000;i++){
      sem_wait(mutex);
      balance++;
      sem_post(mutex);
   }
   pthread_exit(NULL);
}

void * dec(void * arg){
long j;
   for(j=0;j<10000000;j++){
      sem_wait(mutex);
      balance--;
      sem_post(mutex);
    }
   pthread_exit(NULL);
}
```

```
int main(){
   char* name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0666, 1);
   pthread_t t1, t2;

   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);



   printf("Value of balance is :%ld\n", balance);

return 0;
}
```

Handling the Critical Section (global variable)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
long balance = 0;//global variable
sem_t *mutex;

void * inc(void * arg){
long i;
   for(i=0;i<10000000;i++){
      sem_wait(mutex);
      balance++;
      sem_post(mutex);
   }
   pthread_exit(NULL);
}

void * dec(void * arg){
long j;
   for(j=0;j<10000000;j++){
      sem_wait(mutex);
      balance--;
      sem_post(mutex);
    }
   pthread_exit(NULL);
}
```

```
int main(){
   char* name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0666, 1);
   pthread_t t1, t2;

   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);

   sem_close(mutex);
   sem_unlink(name);
   printf("Value of balance is :%ld\n", balance);

return 0;
}
```

Handling the Critical Section (global variable)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
long balance = 0;//global variable
sem_t *mutex;

void * inc(void * arg){
long i;
   for(i=0;i<10000000;i++){
      sem_wait(mutex);
      balance++;
      sem_post(mutex);
   }
   pthread_exit(NULL);
}

void * dec(void * arg){
long j;
   for(j=0;j<10000000;j++){
      sem_wait(mutex);
      balance--;
      sem_post(mutex);
    }
   pthread_exit(NULL);
}
```

```
int main(){
   char* name = "/sem1";
   mutex = sem_open(name, O_CREAT, 0666, 1);
   pthread_t t1, t2;

   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);

   sem_close(mutex);
   sem_unlink(name);
   printf("Value of balance is :%ld\n", balance);

return 0;
}
```

OUTPUT:

```
Value of balance is: 0
```

PROBLEM:
threads executed out of order!

PROBLEM: threads executed out of order!

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

void * f1(void * parm){

   fprintf(stderr, " is fun.");
}
void * f2(void * parm){

   fprintf(stderr, " Operating Systems");

}
void * f3(void * parm){

   fprintf(stderr," Learning ");

}
```

```
int main() {

   pthread_t t1, t2,t3;
   pthread_create(&t1, NULL, f1, NULL);
   pthread_create(&t2, NULL, f2, NULL);
   pthread_create(&t3, NULL, f3, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);

   printf("\n");
return 0;
}
```

UNDESIRED OUTPUT:

is fun. Operating Systems Learning
Operating Systems is fun. Learning

Handling sequential execution of threads.

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

void * f1(void * parm){

   fprintf(stderr, " is fun.");
}
void * f2(void * parm){

   fprintf(stderr, " Operating Systems");

}
void * f3(void * parm){

   fprintf(stderr," Learning ");


}
```

```
int main() {

   pthread_t t1, t2,t3;
   pthread_create(&t1, NULL, f1, NULL);
   pthread_create(&t2, NULL, f2, NULL);
   pthread_create(&t3, NULL, f3, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);

   printf("\n");
return 0;
}
```

Handling sequential execution of threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>


sem_t *semA, *semB;


void * f1(void * parm){

   fprintf(stderr, " is fun.");
}
void * f2(void * parm){

   fprintf(stderr, " Operating Systems");

}
void * f3(void * parm){

   fprintf(stderr," Learning ");


}
```

```
int main() {


   pthread_t t1, t2,t3;
   pthread_create(&t1, NULL, f1, NULL);
   pthread_create(&t2, NULL, f2, NULL);
   pthread_create(&t3, NULL, f3, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);



   printf("\n");
return 0;
}
```

Handling sequential execution of threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>


sem_t *semA, *semB;


void * f1(void * parm){

   fprintf(stderr, " is fun.");
}
void * f2(void * parm){

   fprintf(stderr, " Operating Systems");

}
void * f3(void * parm){

   fprintf(stderr," Learning ");

}
```

```
int main() {
   char* name1 = "/sem1";
   semA = sem_open(name1, O_CREAT, 0666, 0);
   char* name2 = "/sem2";
   semB = sem_open(name2, O_CREAT, 0666, 0);

   pthread_t t1, t2,t3;
   pthread_create(&t1, NULL, f1, NULL);
   pthread_create(&t2, NULL, f2, NULL);
   pthread_create(&t3, NULL, f3, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);




   printf("\n");
return 0;
}
```

Handling sequential execution of threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>


sem_t *semA, *semB;


void * f1(void * parm){
   sem_wait(semB);
   fprintf(stderr, " is fun.");
}
void * f2(void * parm){
   sem_wait(semA);
   fprintf(stderr, " Operating Systems");
   sem_post(semB);
}
void * f3(void * parm){

   fprintf(stderr," Learning ");
   sem_post(semA);
}
```

```
int main() {
   char* name1 = "/sem1";
   semA = sem_open(name1, O_CREAT, 0666, 0);
   char* name2 = "/sem2";
   semB = sem_open(name2, O_CREAT, 0666, 0);

   pthread_t t1, t2,t3;
   pthread_create(&t1, NULL, f1, NULL);
   pthread_create(&t2, NULL, f2, NULL);
   pthread_create(&t3, NULL, f3, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);




   printf("\n");
return 0;
}
```

Handling sequential execution of threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>


sem_t *semA, *semB;


void * f1(void * parm){
   sem_wait(semB);
   fprintf(stderr, " is fun.");
}
void * f2(void * parm){
   sem_wait(semA);
   fprintf(stderr, " Operating Systems");
   sem_post(semB);
}
void * f3(void * parm){

   fprintf(stderr," Learning ");
   sem_post(semA);
}
```

```
int main() {
   char* name1 = "/sem1";
   semA = sem_open(name1, O_CREAT, 0666, 0);
   char* name2 = "/sem2";
   semB = sem_open(name2, O_CREAT, 0666, 0);

   pthread_t t1, t2,t3;
   pthread_create(&t1, NULL, f1, NULL);
   pthread_create(&t2, NULL, f2, NULL);
   pthread_create(&t3, NULL, f3, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);

   sem_close(semA);
   sem_close(semB);
   sem_unlink(name1);
   sem_unlink(name2);

   printf("\n");
return 0;
}
```

Handling sequential execution of threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>


sem_t *semA, *semB;


void * f1(void * parm){
   sem_wait(semB);
   fprintf(stderr, " is fun.");
}
void * f2(void * parm){
   sem_wait(semA);
   fprintf(stderr, " Operating Systems");
   sem_post(semB);
}
void * f3(void * parm){

   fprintf(stderr," Learning ");
   sem_post(semA);
}
```

```
int main() {
   char* name1 = "/sem1";
   semA = sem_open(name1, O_CREAT, 0666, 0);
   char* name2 = "/sem2";
   semB = sem_open(name2, O_CREAT, 0666, 0);

   pthread_t t1, t2,t3;
   pthread_create(&t1, NULL, f1, NULL);
   pthread_create(&t2, NULL, f2, NULL);
   pthread_create(&t3, NULL, f3, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);

   sem_close(semA);
   sem_close(semB);
   sem_unlink(name1);
   sem_unlink(name2);

   printf("\n");
return 0;
}
```

OUTPUT:

Learning Operating Systems is fun.

Summary

Software tools to restrict access to global variables and to provide synchronization between threads:

- Mutexes:
  - provide exclusive access to code segments/variables with the use of locks (mutual exclusion)
- Semaphores:
  - provide exclusive access to code segments/variables with the use of locks (mutual exclusion)
  - provide synchronization and communication between threads that share a mutex

Thank you!