HILOS

Hilos

Hilos

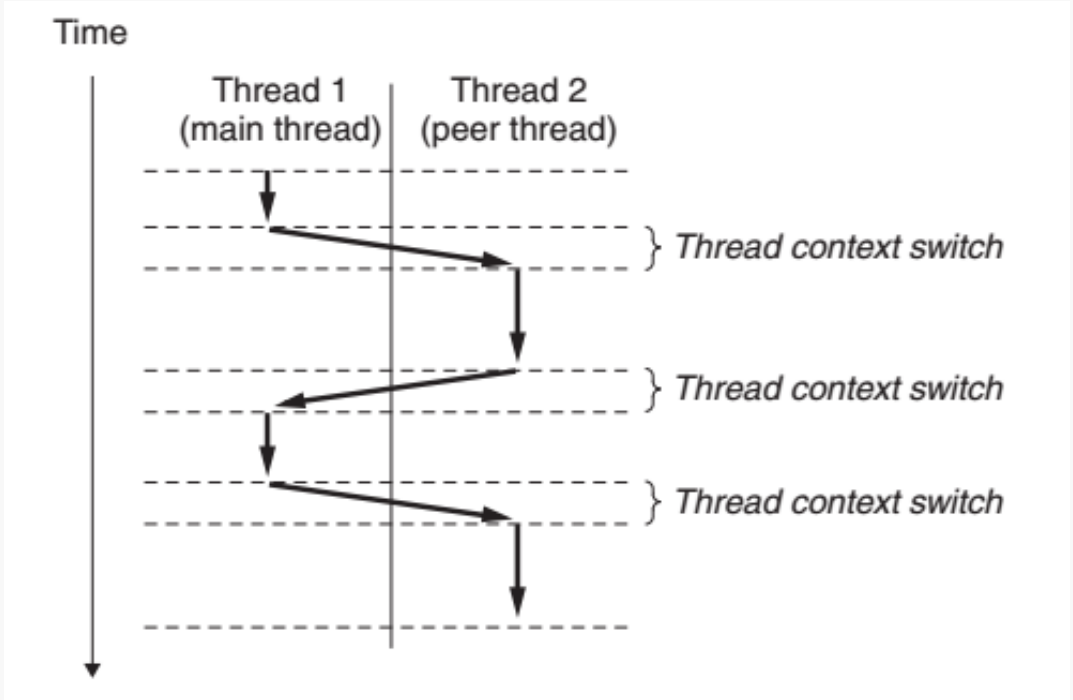
Un hilo es un flujo lógico que corre en contexto de un proceso.

Generalmente hay un hilo por proceso.

Comparten el espacio de direcciones

Hilos

Ejecución



Hilos

¿Por qué hilos?

- Con múltiples hilos, un proceso puede hacer múltiples cosas a la vez.
- Beneficios:
    1. ***Simplifica*** *el código para manejo de eventos asícrono (1 hilo por tipo de evento).*
    2. ***Compartir recursos es más fácil*** *con hilos que con procesos. Los hilos comparten el mismo espacio de direcciones.*
    3. ***Mejor rendimiento*** *(dependiendo del tipo de programa).*
    4. ***Mejor tiempo de respuesta*** *en programas interactivos.*
    5. *Programas multihilos pueden correr sobre 1 CPU (o core), pero* ***tendrán mejor rendimiento en CPUs multi-core.***

Hilos

Hilos POSIX

- Todo hilo tiene una identificación, el **Thread ID.**
- Además, cada hilo tiene sus propios/as:
  1. *Valores de registros*
  2. *Stack*
  3. *Prioridad de planificación*
  4. *Máscara de señales*
  5. *Variable* `errno`
  6. *Datos específicos del hilo*

Hilos

- **Interface de hilos es la librería POSIX threads (`pthreads`)**
- Cada hilo tiene su identificación, pero este ID solo tiene significado dentro del mismo proceso.
- Para compilar, necesitamos la bandera **`-pthread`**
- ID es representado en el tipo de dato `pthread_t`.

```
#include <pthread.h>

int pthread_equal( pthread_t tid1, pthread_t tid2);
```

- Esta función la usamos para comparar dos thread IDs. No podemos comparar dos variables pthread_t directamente (portabilidad).

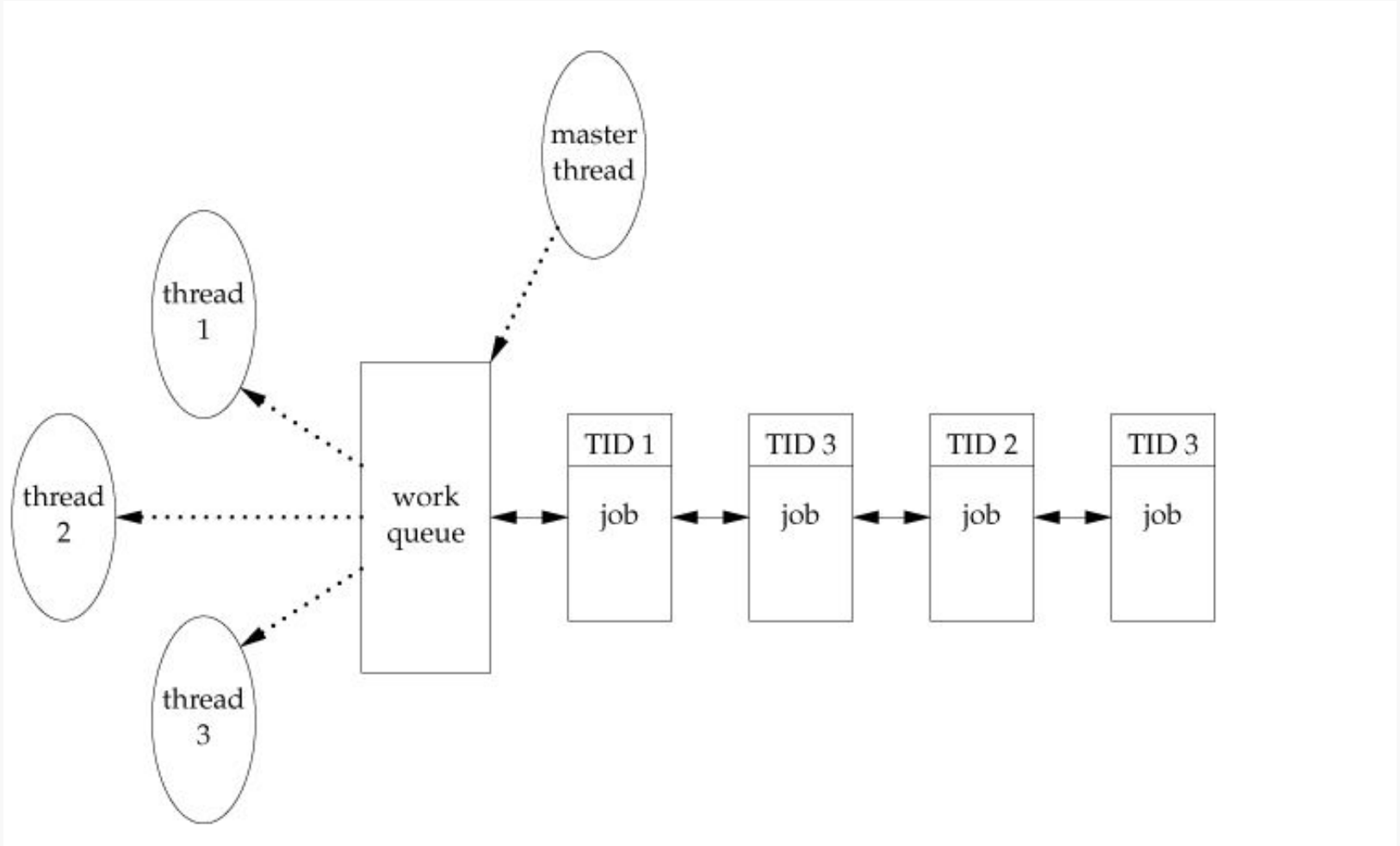
Hilos

- Para obtener el ID del hilo, usamos:

```
#include <pthread.h>

int pthread_self(void);
```

- Util cuando un proceso tiene un hilo master, que maneja otros hilos esclavos, y desea asignar trabajos a hilos específicos.


master thread
thread 1
thread 2
thread 3
work queue
TID 1
job
TID 3
job
TID 2
job
TID 3
job

Hilos

Creación de hilos

Usamos la función pthread_create:

```
#include < pthread.h >

int pthread_create( pthread_t *restrict tidp,
                    const pthread_attr_t *restrict attr,
                    void *(* start_rtn)( void *), void *restrict arg);
```

- `tdip` es un puntero a tipo de datos pthread_t (ID de hilo).
- `attr` permite configurar atributos del hilo. Por ahora le damos `NULL`.
- `start_tn` es la dirección de una función. Esta toma un solo argumento, del tipo puntero a `void`. La usamos para pasar informacion al hilo.

```c
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void printids( const char *s) {
        pid_t pid; pthread_t tid;
        pid = getpid();

        tid = pthread_self();
        printf("% s pid %lu tid %lu (0x% lx)\ n", s, (unsigned long) pid, (unsigned
                long) tid, (unsigned long) tid);
}

void * thr_fn( void *arg) {
        printids(" new thread: ");
        return(( void *) 0);
}

int main( void) {
        int err;
        err = pthread_create(&ntid, NULL, thr_fn, NULL);
        if (err != 0)
                err_exit( err, "can' t create thread");
        printids(" main thread:");
        sleep( 1);
        exit( 0);
 }
```

Hilos

Detalles del a tener en cuenta:

1. Se usa sleep, para que el hilo principal no termine antes que los otros (**aunque aún así no es garantizado**).

2. Obtenemos el ID del hilo llamando a `pthread_self`
   - *No podemos usar ntid*
   - *Hilo principal puede no haber terminado pthread_create cuando el nuevo hilo empieza a correr.*

Hilos

- **Terminando hilos**
- Si cualquier hilo llama `exit`, `_exit` o `_Exit`, el proceso termina.
- Un hilo puede terminar de tres formas:
  1. *Cuando **retorna** de su rutina (función).*
  2. *Hilo es **cancelado** por otro hilo en el mismo proceso*
  3. *Hilo llama **`pthread_exit`***

Terminando Hilos

```
#include < pthread.h >

void pthread_exit( void *rval_ptr);
```

- `rval_ptr` es valor de retorno. Esta disponible para los otros hilos llamando `pthread_join`.

Hilos

Función pthread_join

```
#include < pthread.h >

int pthread_join( pthread_t thread, void ** rval_ptr);
```

- Esta función hace que el hilo que llame, se bloquee hasta que el hilo `thread` termine.
- Al llamar esta función, el hilo `thread` es puesto en estado *detached*.
- Si no nos interesa el valor de retorno del hilo, `rval_ptr` puede ser `NULL`.

```c
#include “apue.h”
#include <pthread.h>

void * thr_fn1( void *arg) {
        printf(" thread 1 returning\ n");
        return(( void *) 1);
}
void * thr_fn2( void *arg) {
        printf(" thread 2 exiting\ n");
        pthread_exit(( void *) 2);
}

int main( void) {
        int err; pthread_t tid1, tid2;
        void *tret; err = pthread_create(& tid1, NULL, thr_fn1, NULL);

        if (err != 0)
                err_exit( err, "can' t create thread 1");
        err = pthread_create(& tid2, NULL, thr_fn2, NULL);
        if (err != 0)
                err_exit( err, "can' t create thread 2");

        err = pthread_join( tid1, &tret);
        if (err != 0)
                err_exit( err, "can' t join with thread 1");
        printf(" thread 1 exit code %ld\ n", (long) tret);
        err = pthread_join( tid2, &tret);
                if (err != 0)
        err_exit( err, "can' t join with thread 2");
        printf(" thread 2 exit code %ld\ n", (long) tret);
        exit(0);
}
```

- Puntero mandado a la función del hilo puede ser una estructura.
- Si un hilo crea una estructura (que se almacena **en el stack del hilo**), y el hilo se lo pasa a otro hilo usando pthread_create o para devolver un valor a pthread_exit, puede darse el caso que esa memoria haya sido liberada para cuando estas funciones hagan uso de ella.

- Veamos un ejemplo.

```c
#include "apue.h"
#include < pthread.h >

struct foo { int a, b, c, d; };

void printfoo( const char *s, const struct foo *fp) {
        printf("% s", s);
        printf(" structure at 0x% lx\ n", (unsigned long) fp);
        printf(" foo.a = %d\ n", fp-> a);
        printf(" foo.b = %d\ n", fp-> b);
        printf(" foo.c = %d\ n", fp-> c);
        printf(" foo.d = %d\ n", fp-> d);
}

void * thr_fn1( void *arg) {
        struct foo foo = {1, 2, 3, 4};
        printfoo(" thread 1:\ n", &foo);
        pthread_exit(( void *)&foo);
}

void * thr_fn2( void *arg) {
        printf(" thread 2: ID is %lu\ n", (unsigned long) pthread_self());
        pthread_exit(( void *) 0);
}
```

```
int main( void) {
        int err;
        pthread_t tid1, tid2;
        struct foo *fp;

        err = pthread_create(&tid1, NULL, thr_fn1, NULL);

        if (err != 0)
                err_exit( err, "can' t create thread 1");
        err = pthread_join( tid1, (void *)&fp);

        if (err != 0)
                err_exit( err, "can' t join with thread 1");

        sleep( 1);
        printf(" parent starting second thread\n");
        err = pthread_create(&tid2, NULL, thr_fn2, NULL);
        if (err != 0)
                err_exit( err, "can' t create thread 2");
        sleep(1);
        printfoo(" parent:\n", fp);
        exit( 0);
}
```

Al ejecutarlo tenemos:

```
$ ./ a.out
thread 1:
   structure at 0x7f2c83682ed0
   foo.a = 1
   foo.b = 2
   foo.c = 3
   foo.d = 4
parent starting second thread
thread 2: ID is 139829159933696
parent: structure at 0x7f2c83682ed0
   foo.a = -2090321472
   foo.b = 32556
   foo.c = 1
   foo.d = 0
```

El hilo dos sobre-escribió el stack del hilo uno (depende del SO). Para resolver, usamos variable global, o asignamos memoria con malloc.

Hilos

Función pthread_cancel

```
#include < pthread.h >

int pthread_cancel( pthread_t tid);
```

- Al llamar esta función, el hilo con id **`tid`** se comportará como si hubiera llamado pthread_exit con argumento PTHREAD_CANCELED.
  - *Hilo afectado puede controlar o ignorar como es cancelada.*

Hilos

Funcion pthread_cleanup_push

```
#include < pthread.h >

void pthread_cleanup_push( void (* rtn)( void *), void *arg);

void pthread_cleanup_pop( int execute);
```

- Un hilo puede hacer que ciertas funciones sean llamadas al salir (del hilo), como lo hacemos con `atexit`

```c
#include "apue.h"
#include < pthread.h >

void cleanup( void *arg) {
        printf(" cleanup: %s\n", (char *) arg);
}

void * thr_fn1( void *arg) {
        printf(" thread 1 start\n");
        pthread_cleanup_push( cleanup, "thread 1 first handler");
        pthread_cleanup_push( cleanup, "thread 1 second handler");
        printf(" thread 1 push complete\n");
        if (arg)
                return(( void *) 1);
        pthread_cleanup_pop( 0);
        pthread_cleanup_pop( 0);
        return(( void *) 1);
}
void * thr_fn2( void *arg) {
        printf(" thread 2 start\n");
        pthread_cleanup_push( cleanup, "thread 2 first handler");
        pthread_cleanup_push( cleanup, "thread 2 second handler");
        printf(" thread 2 push complete\n");
        if (arg)
                pthread_exit(( void *) 2);
        pthread_cleanup_pop( 0);
        pthread_cleanup_pop( 0);
        pthread_exit(( void *) 2);
}
.
```

```
int main( void) {
        int err; pthread_t tid1, tid2;
        void *tret;

        err = pthread_create(&tid1, NULL, thr_fn1, (void *) 1);
         if (err != 0)
                err_exit( err, "can't create thread 1");
        err = pthread_create(&tid2, NULL, thr_fn2, (void *) 1);
        if (err != 0)
                err_exit( err, "can't create thread 2");
        err = pthread_join( tid1, &tret);
        if (err != 0)
                err_exit( err, "can't join with thread 1");

        printf(" thread 1 exit code %ld\n", (long) tret);
        err = pthread_join( tid2, &tret);
        if (err != 0)
                err_exit( err, "can't join with thread 2");
        printf(" thread 2 exit code %ld\n", (long) tret);
        exit(0);
}
```

```
$ ./ a.out
thread 1 start
thread 1 push complete
thread 2 start
thread 2 push complete
cleanup: thread 2 second handler
cleanup: thread 2 first handler
thread 1 exit code 1
thread 2 exit code 2
```

- Las funciones son llamadas en orden inverso que son registradas.
- Si hilo termina por que llegamos al **fin de su rutina**, estos *handlers* **no** son llamados.

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

- Status de terminación del hilo es retenido otro hilo hasta llame a pthread_join sobre nosotros.

Hilos

Función pthread_detach

```
#include < pthread.h >

int pthread_detach( pthread_t tid);
```

Si hilo es *detached*, el espacio de memoria del hilo es retomado por el SO automáticamente, sin tener que llamar `pthread_join`.

- **Sincronización de Hilos**

- Cuando múltiples hilos comparten información, **DEBEMOS** asegurarnos que tengan una vista **CONSISTENTE** de la información.

- Cuando un hilo modifica una variable, los otros hilos **pueden potencialmente** ver inconsistencias.

- Más probable en arquitecturas donde las escrituras de memoria toman más de un ciclo de reloj.
  - *Se generan condiciones de carrera*

SINCRONIZACIÓN DE HILOS

Sincronización de Hilos

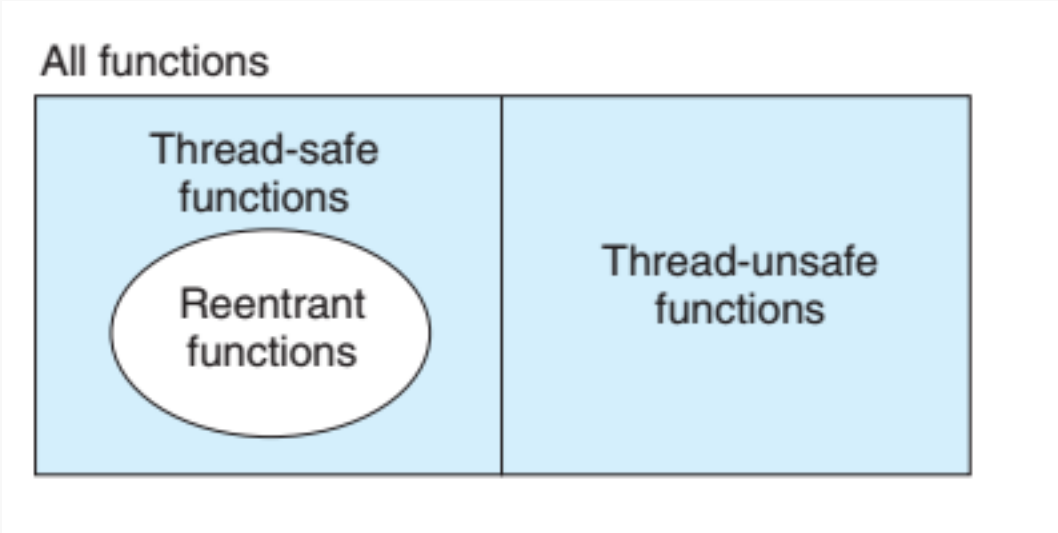
Thread-Safety

Cuatro clases de funciones

1. Funciones no protegen variables compartidas
2. Funciones que mantienen estado entre instancias
3. Funciones retornan puntero a variables estatica
4. Funciones que llaman funciones inseguras para hilos

Sincronizacion de Hilos

Re-entrancia



Funciones que NO hacen referencia a ninguna informacion compartida.

Sincronizacion de Hilos

Thread-unsafe function	Thread-unsafe class	Unix thread-safe version
`rand`	2	`rand_r`
`strtok`	2	`strtok_r`
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	(none)
`localtime`	3	`localtime_r`

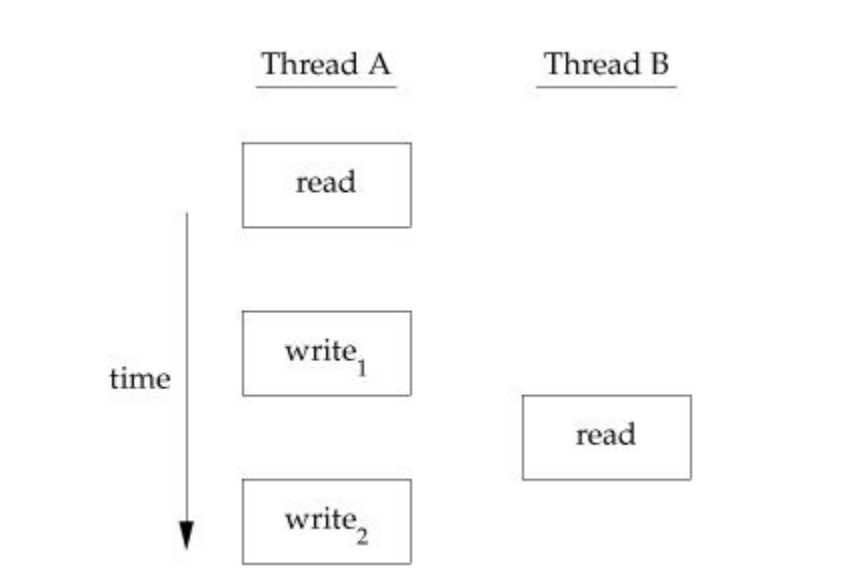
Sincronizacion de Hilos

Condiciones de Carrera

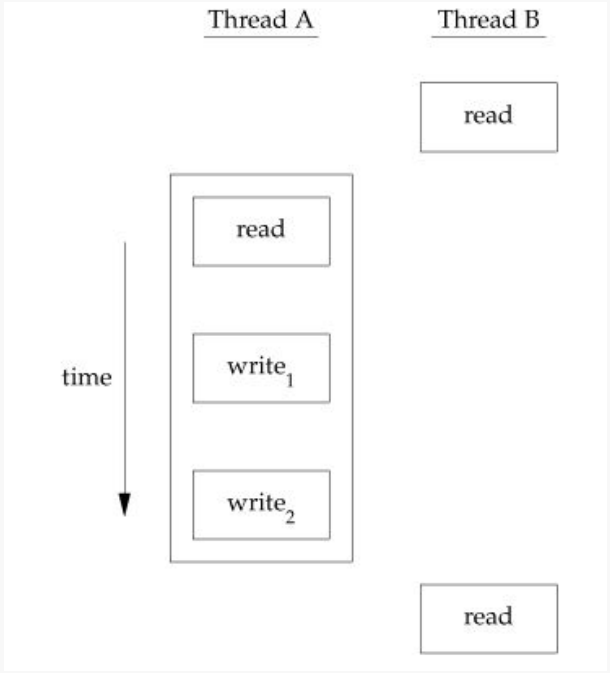
Ocurren cuando la ejecucion correcta de un programa depende de que uno u otro hilo llegue a un punto X antes que otro llegue a un punto Y.

Sincronización de hilos

- Hilo A escribe una variable que lee el hilo B.
- La escritura toma 2 ciclos
- Hilo B trata de leer entre el ciclo 1 y 2 de escritura de memoria, leerá valores inconsistentes.



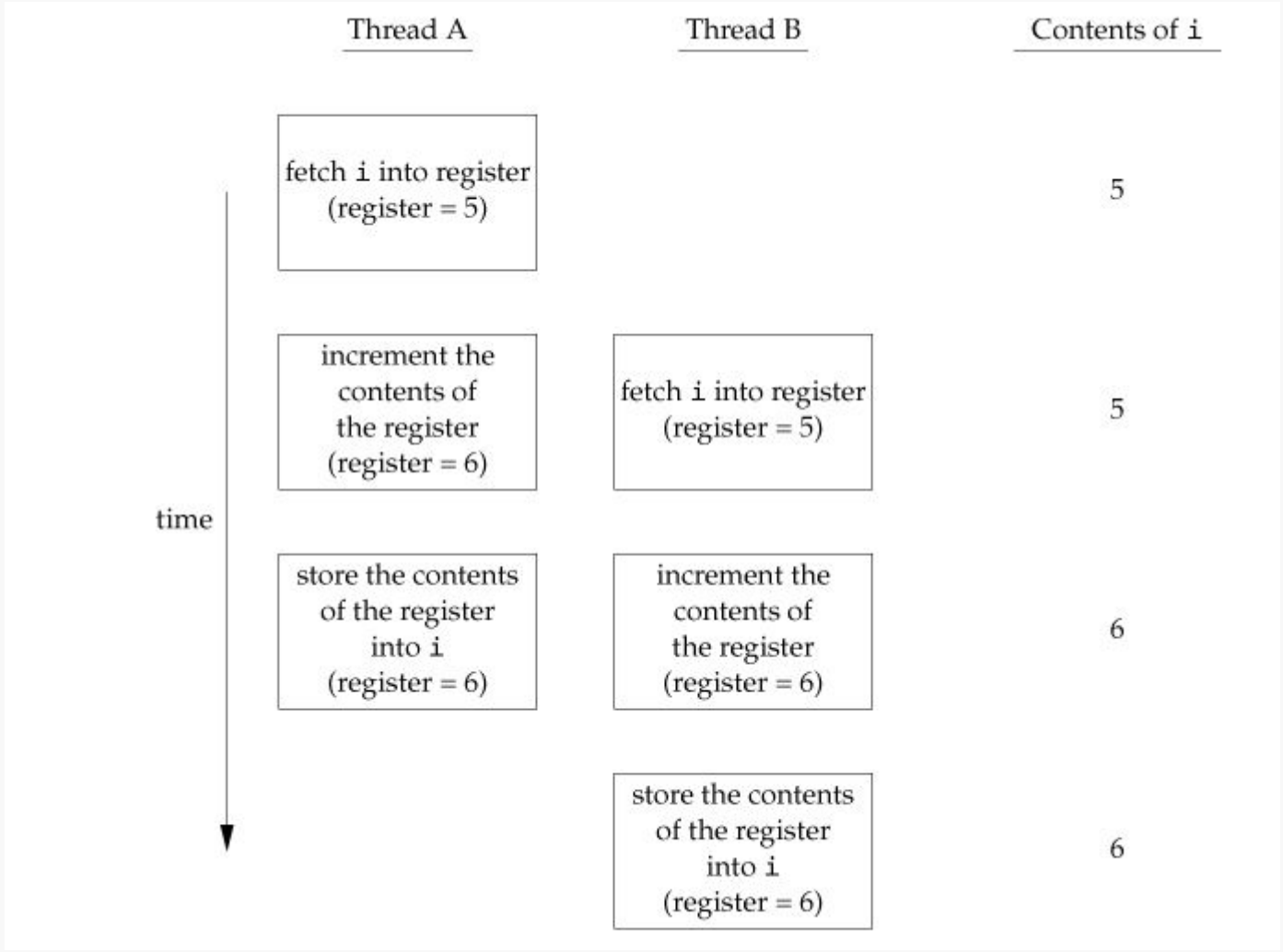
Sincronizacion de Hilos



- Para resolver debemos usar un candado (*lock)*, que permita que un solo hilo a la vez pueda acceder a la variable.

Sincronizacion de Hilos

- Otra situación que se presenta es que múltiples hilos quieren modificar una variable a la vez.
- Modificar una variable involucra:
  - *Leer la variable y cargarla en un registro del CPU*
  - *Modificar el registro*
  - *Escribir el nuevo valor de registro en la memoria.*
- Si dos hilos tratan de actualizar el valor al mismo tiempo, el resultado será inconsistente.


Thread A
Thread B
Contents of i
fetch i into register (register = 5)
5
increment the contents of the register (register = 6)
fetch i into register (register = 5)
5
time
store the contents of the register into i (register = 6)
increment the contents of the register (register = 6)
6
store the contents of the register into i (register = 6)
6

Sincronizacion de Hilos

Semaforos

Dos operaciones:

P(s): si s < 0, P decrementa s y retorna; caso contrario hilo se suspende.

V(s): incrementa s en 1. Si hay hilo dormido esperando por s, lo despierta.

Sincronizacion de Hilos

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

Returns: 0 if OK, −1 on error

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

Returns: nothing

Sincronizacion de Hilos

Mutexes

- Un mutex es un candado (*lock)* que lo cerramos antes de ejecutar código de **región crítica,** y lo abrimos cuando salimos de dicha región.
- *El mutex es el caso especial de un semaforo, cuyo valor incial es 1 (semaforo binario)*

- Mutex se presenta con tipo `pthread_mutex_t`
- Lo inicializamos con asignándole `PTHREAD _MUTEX_INITIALIZER` o llamando a `pthread_mutex_init`:

```
pthread_mutex_t mi_mutex = PTHREAD_MUTEX_INITIALIZER;
```

o

```
pthread_mutex_t mi_mutex;
pthread_mutex_init(&mi_mutex, NULL);
```

o

```
pthread_mutex_t *mi_mutex =     malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(mi_mutex, NULL);
```

- **Si lo inicializamos usando malloc, debemos llamar `pthread_mutex_destroy` antes de llamar `free()` para liberar mi_mutex.**

Sincronizacion de Hilos

```
#include <pthread.h>

int pthread_mutex_init( pthread_mutex_t *restrict mutex, const
                        pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy( pthread_mutex_t *mutex);
```

- `mutex` es el mutex a inicializar o destruir.
- Los atributos `attr` serán discutidos después

Sincronizacion de hilos

Abriendo y cerrando el mutex

```
#include < pthread.h >

int pthread_mutex_lock( pthread_mutex_t *mutex);

int pthread_mutex_trylock( pthread_mutex_t *mutex);

int pthread_mutex_unlock( pthread_mutex_t *mutex);
```

- `pthread_mutex_lock` trata de cerrar el *lock* mutex. Si esta cerrado, el hilo se bloquea.
- `pthread_mutex_unlock` abre el candado mutex

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

//Declaramos una estructura
int variable_compartida = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //inicializamos el mutex estaticamente

void * funcion_hilo1(void *arg){

        int i = 0;
        for (i = 0; i < 100000000; i++){
                pthread_mutex_lock(&mutex);
                variable_compartida++;
                pthread_mutex_unlock(&mutex);
        }
        return (void *)0;
}

void * funcion_hilo2(void *arg){

        int i = 0;

        for (i = 0; i < 200000000; i++){
                pthread_mutex_lock(&mutex);
                variable_compartida--;
                pthread_mutex_unlock(&mutex);
        }
        return (void *)0;
}
```

```c
int main(int argc, char **argv){

        pthread_t id1, id2;
        int status;

        status = pthread_create(&id1, NULL, funcion_hilo1, NULL);
        if (status < 0){
                fprintf(stderr, "Error al crear el hilo 1\n");
                exit(-1);
        }
        status = pthread_create(&id2, NULL, funcion_hilo2, NULL);

        if (status < 0){
                fprintf(stderr, "Error al crear el hilo 2\n");
                exit(-1);
        }

        void * valor_retorno = NULL;

        printf("Hilo principal esta esperando a que terminen los otros hilos\n");
        int status1 = pthread_join(id1, NULL);
        int status2 = pthread_join(id2, &valor_retorno);
        if (status1 < 0){
                fprintf(stderr, "Error al esperar por el hilo 1\n");
                exit(-1);
        }

        if (status2 < 0){
                fprintf(stderr, "Error al esperar por el hilo 2\n");
                exit(-1);
        }

        printf("valor final variable compartida %d\n", variable_compartida);
        printf("Hilos terminaron normalmente\n");
        exit(0);
}
```

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

struct foo {
        int f_count;
        pthread_mutex_t f_lock;
        int f_id; /* ... more stuff here ... */
 };

struct foo * foo_alloc( int id) /* allocate the object */
{
        struct foo *fp;
        if (( fp = malloc( sizeof( struct foo))) != NULL) {
                fp->f_count = 1;
                fp->f_id = id;
                if (pthread_mutex_init(& fp-> f_lock, NULL) != 0) {
                        free( fp); return( NULL);
                } /* ... continue initialization ... */
        }
        return(fp);
 }

void foo_hold( struct foo *fp) /* add a reference to the object */
{
        pthread_mutex_lock(&fp-> f_lock);
        fp->f_count++;
        pthread_mutex_unlock(&fp->lock);
}
```

```
void foo_rele(struct foo *fp) /* release a reference to the object */
{
        pthread_mutex_lock(&fp-> f_lock);
        if (--fp->f_count == 0) { /* last reference */
                pthread_mutex_unlock(& fp->f_lock);
                pthread_mutex_destroy(&fp->f_lock);
                free( fp);
        }
        else {
                pthread_mutex_unlock(& fp-> f_lock);
        }
}
```

- Con los mutexes, podemos implementar controles de accesos a variables/estructuras, como se muestra en este ejemplo.

Sincronizacion de Hilos

Deadlock