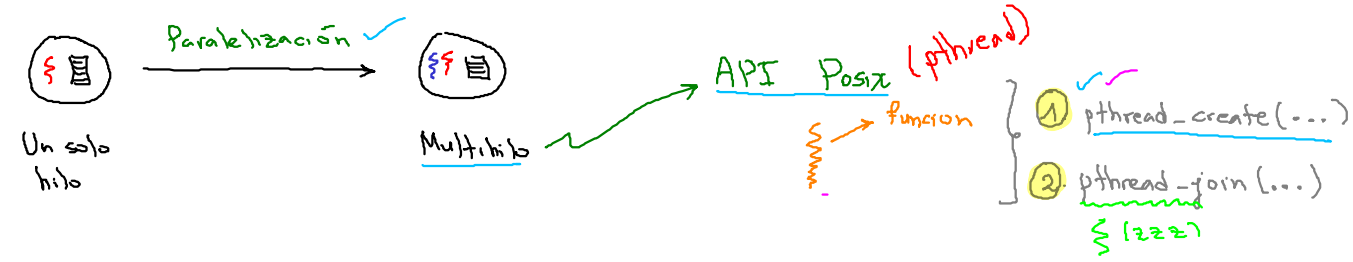
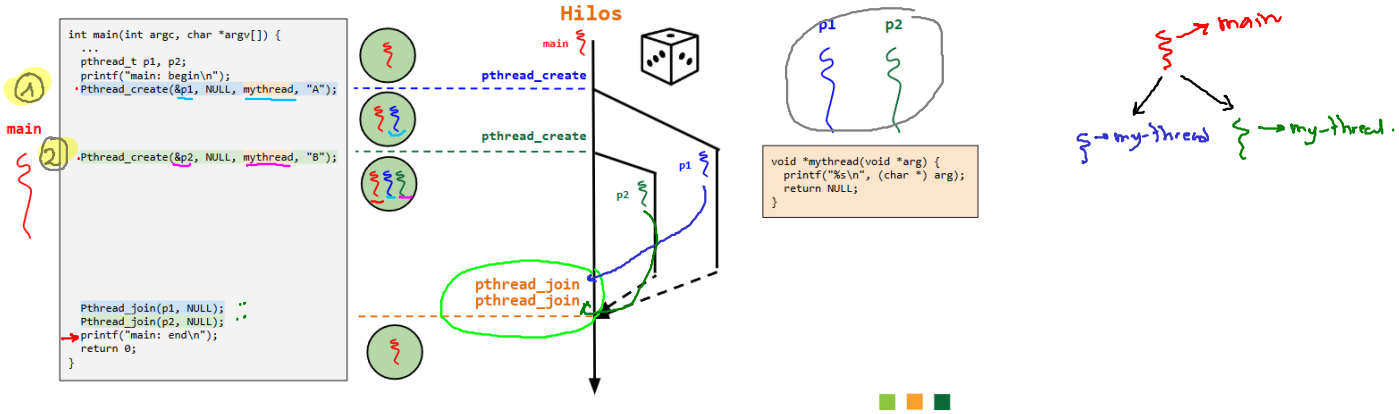
11/11/2025 - Sistemas Operativos (UdeC)

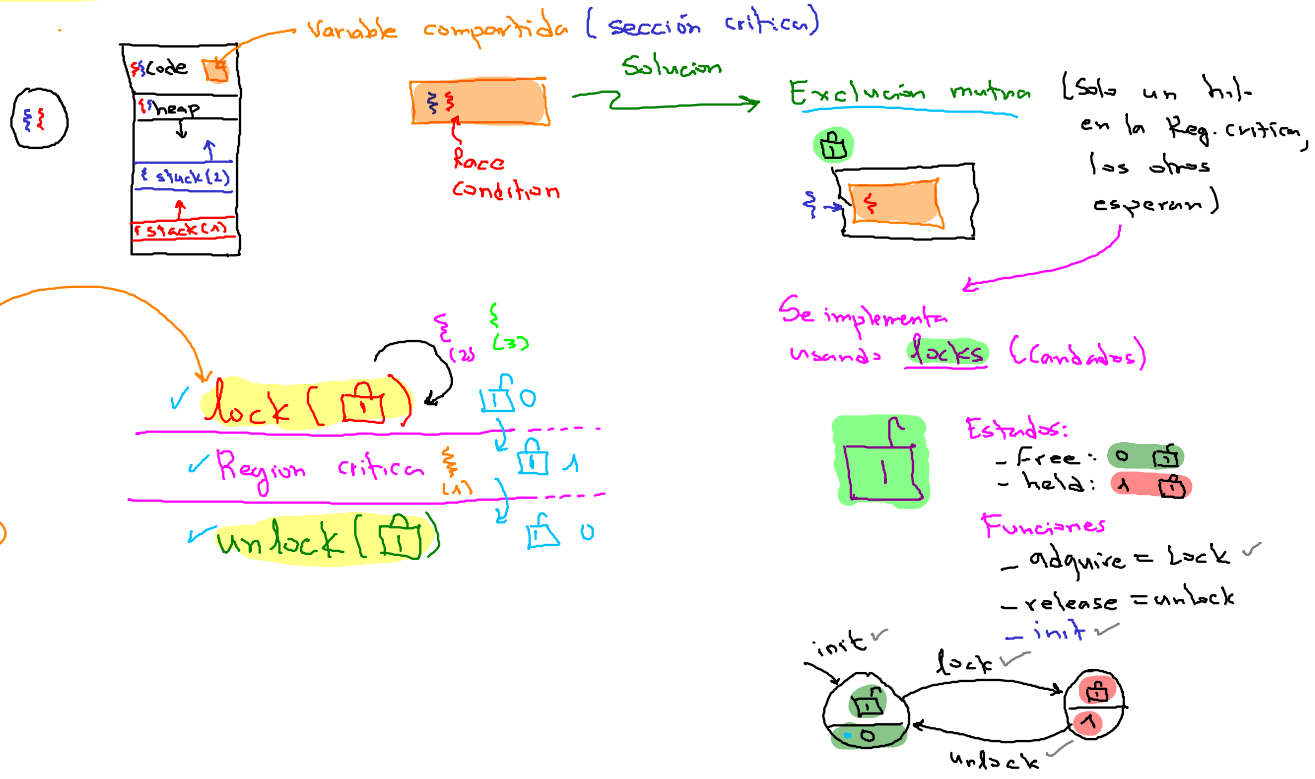
1. Repaso.





Cuando se paraleliza nos enfrentamos a nuevos desafíos (problemas)

1) Race condition



② Orden Aleatorio (Problemas de sincronización)

→ pthread_join

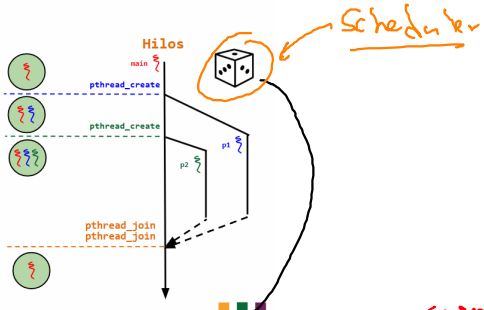
Ejemplo - Orden de ejecución

- **Pregunta**: ¿Cual hilo se ejecuta primero?
- **Respuesta**:
  - El orden es indeterminado (aleatorio).
  - Depende del scheduler.

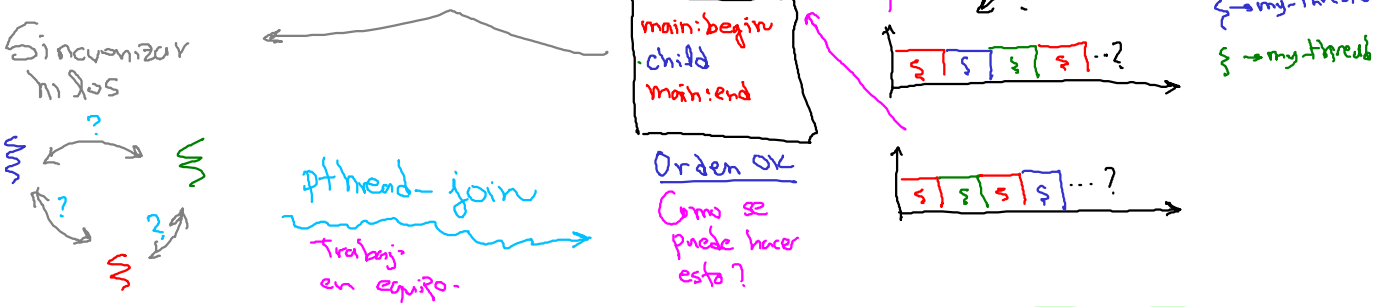
La salida es no determinista (Aleatoria)

Ejemplo - Orden de ejecución

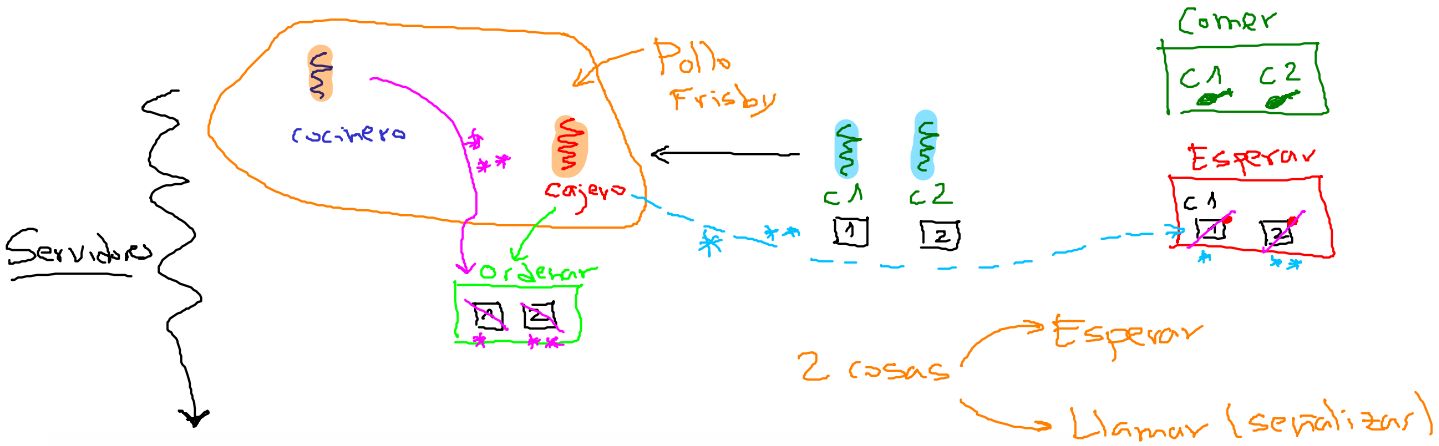
```
13 int main(int argc, char *argv[]) {
14     if (argc != 1) {
15     fprintf(stderr, "usage: main\n");
16     exit(1);
17     }
18
19     pthread_t p1, p2;
20     printf("main: begin\n");
21     Pthread_create(&p1, NULL, mythread, "A");
22     Pthread_create(&p2, NULL, mythread, "B");
23     // join waits for the threads to finish
24     Pthread_join(p1, NULL);
25     Pthread_join(p2, NULL);
26     printf("main: end\n");
27     return 0;
28 }
```



Scheduler

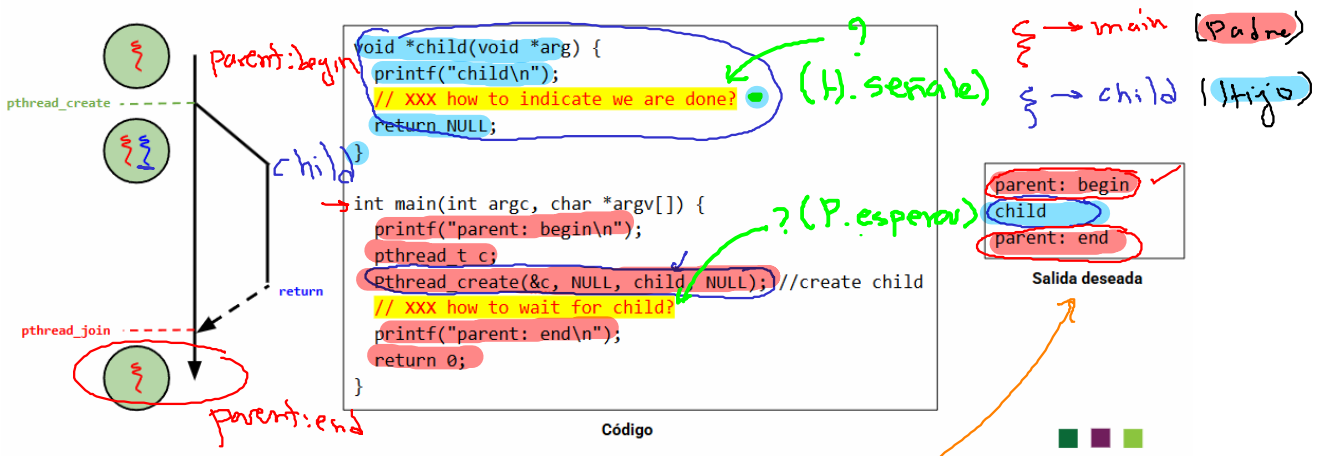


2. Como lograr implementar sincronización → VC (variables de condición)



¿Cómo implementar el orden de ejecución deseado?

- **¿Cómo implementar el join?** (Para lograr el orden de ejecución deseado).



Como lograr que esta siempre sea la salida?

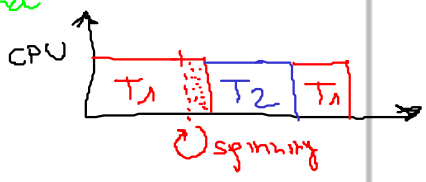
```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

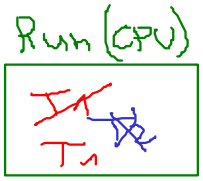
Flag { 0: Hijo no ha acabado
1: Hijo ha acabado

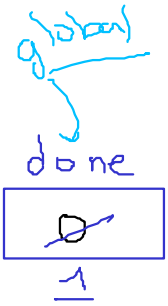
H: Llama

P: Espera

spin-lock








```
parent: begin
child
parent: end
```

parent: Begin
child
parent: End

Spin-lock (Indeseable) – Como evitar el spinlock

Primitivas:
1 – lock (Mutual Exclusion)
2 – CV { condition Variable (Sincroni

Variables de condición (Condition Variables - CV)

Estado – flag

Idea clave: ¿Cómo esperar por una condición?

- **Espera de una condición:** En programas multihilo, es a menudo útil que un hilo espere hasta que alguna condición se vuelva verdadera antes de proseguir.
- **Solución ineficiente – Spinning:** Una solución es esperar validando continuamente hasta que la condición se vuelva verdadera (**spinning**) sin embargo esto es muy ineficiente debido al gasto de CPU.

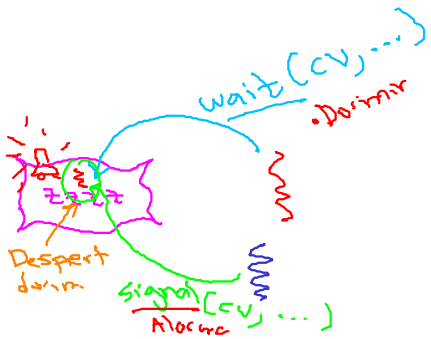
¿Cómo implementar un hilo que espere por una condición?

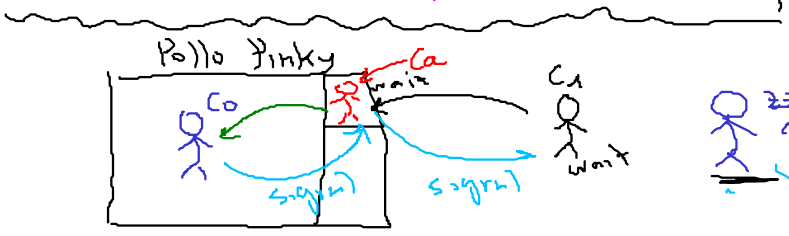
→ Primitivas

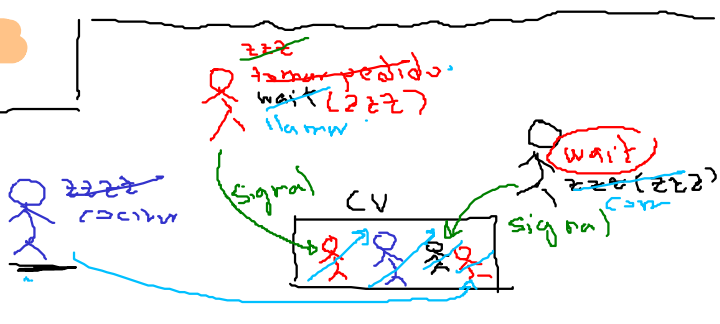
1 lock 1 — init lock unlock

(new) 2 Variables de condición

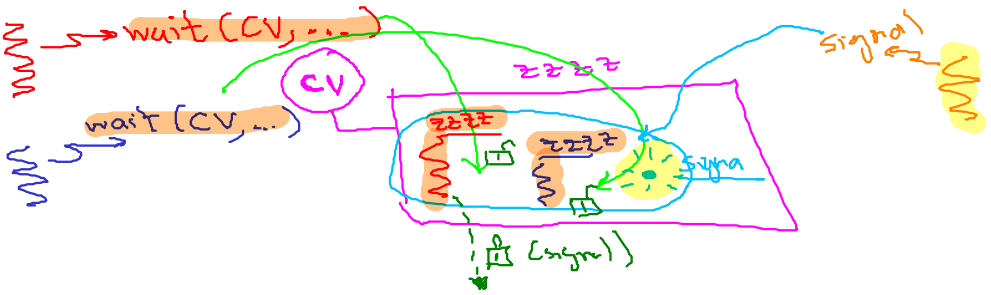
CV | wait(CV, ...) | signal(CV, ...)



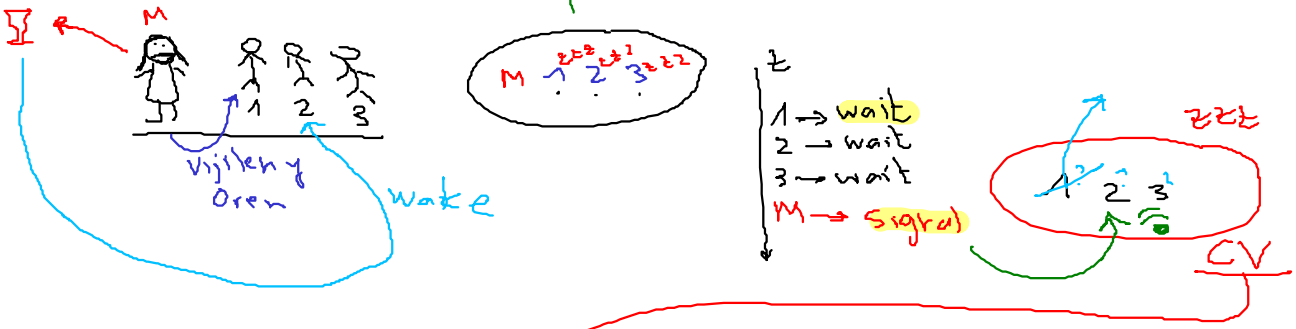




3. Variables de condición y funciones wait (zzz) y signal (despertar)



→ Semana Santa (Desprendimiento → Oración en el huerto



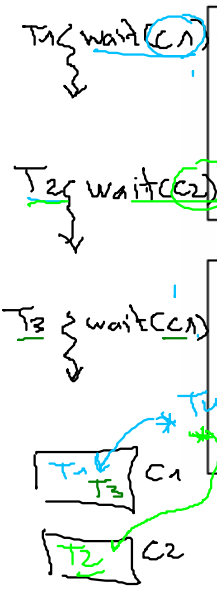
Operaciones de las variables de condición

wait(cond_t *cv, mutex_t *mutex)

- Un hilo se pone en una cola (a dormir) a la espera de que un estado deseado suceda.
- Recibe un mutex como parámetro.
- Cuando se llama el wait el lock es liberado y se pone el hilo a dormir.

signal(cond_t *cv)

- Despierta un único hilo de los que se encuentran en la cola de espera (**if >= 1** el hilo está esperando).
- Si no hay hilos esperando, solo retorna sin hacer nada.
- Cuando el hilo se despierte se debe adquirir de nuevo el lock.



Pthreads interface – Definición y funciones

Los Mutexes and CVs son soportados en Pthreads

1. Declaración de una variable de condición (CV) y el mutex:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

Pthreads interface – Definición y funciones

Los Mutexes and CVs son soportados en Pthreads

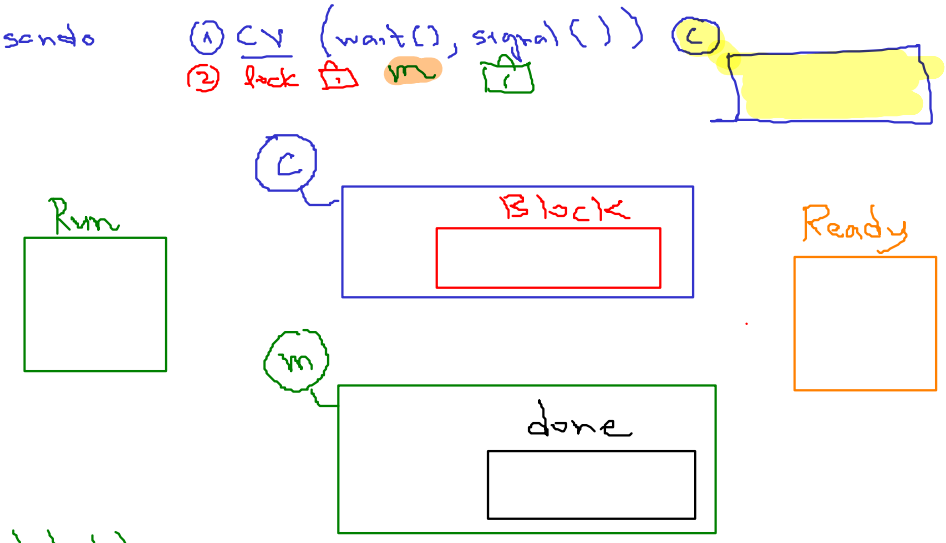
2. Operaciones de una variable de condición (CV):
   - **wait()**: Esta llamada es ejecutada cuando un hilo desea ponerse a sí mismo a dormir.

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

   - **Signal()**: Esta llamada es realizada cuando un hilo ha cambiado algo en el programa y desea despertar un hilo en espera (que se encuentra durmiendo) cuando se da esta condición.

```
pthread_cond_signal(pthread_cond_t *c);
```

4. Reimplementación usando ① CV (wait(), signal()) ② lock



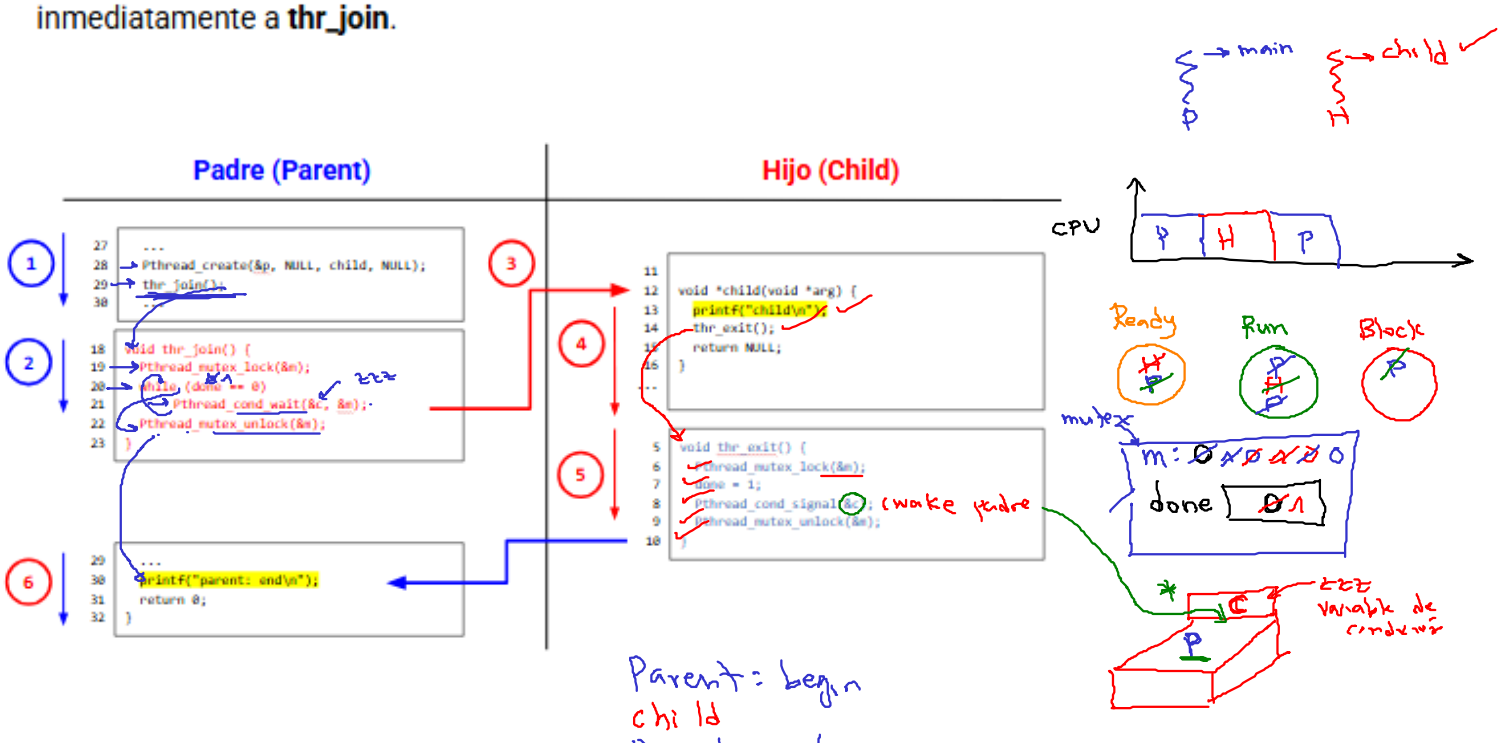

```c
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
  Pthread_mutex_lock(&m);
  done = 1;
  Pthread_cond_signal(&c);
  Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
  printf("child\n");
  thr_exit();
  return NULL;
}
```

```c
void thr_join() {
  Pthread_mutex_lock(&m);
  while (done == 0)
    Pthread_cond_wait(&c, &m);
  Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  pthread_t p;
  Pthread_create(&p, NULL, child, NULL);
  thr_join();
  printf("parent: end\n");
  return 0;
}
```

Código (link)

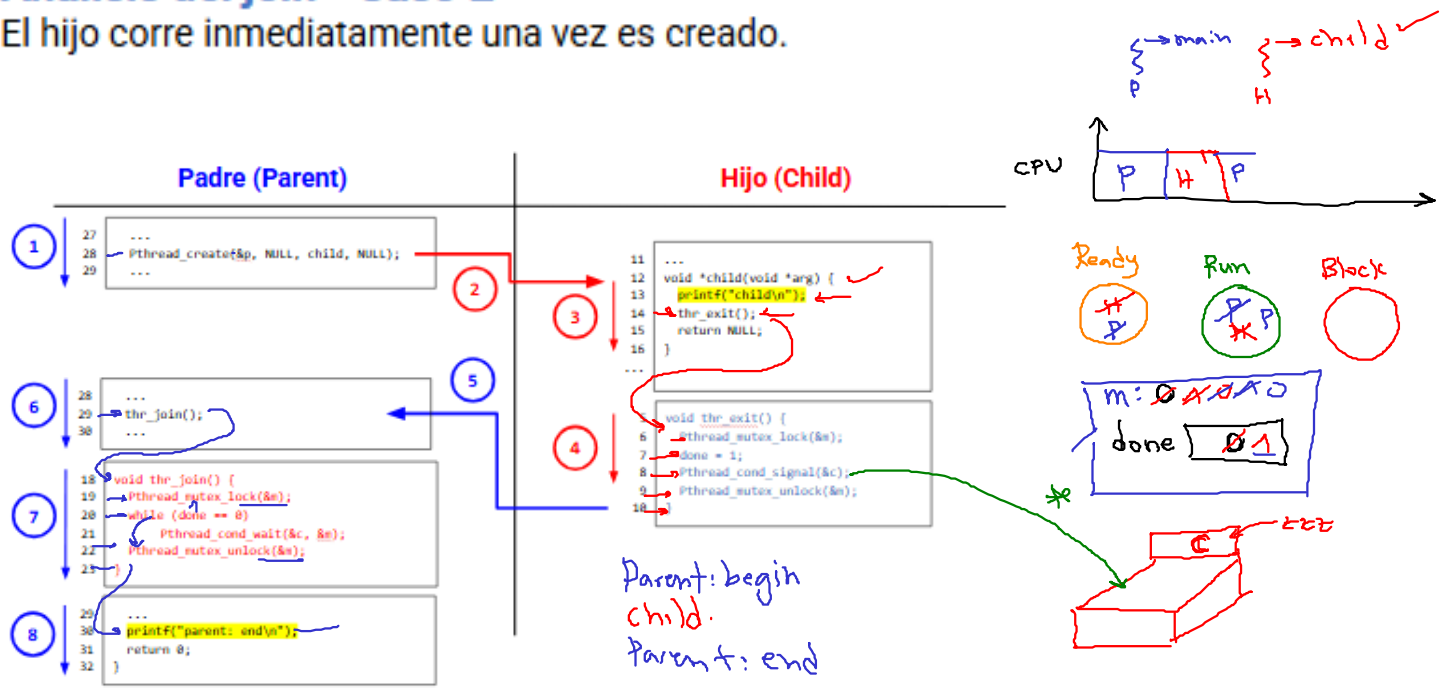
Análisis del join - Caso 1

El padre espera por el hijo; pero continúa corriendo así mismo de modo que llama inmediatamente a **thr_join**.



Análisis del join - Caso 2

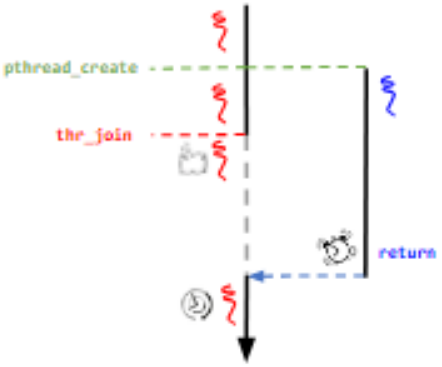
El hijo corre inmediatamente una vez es creado.

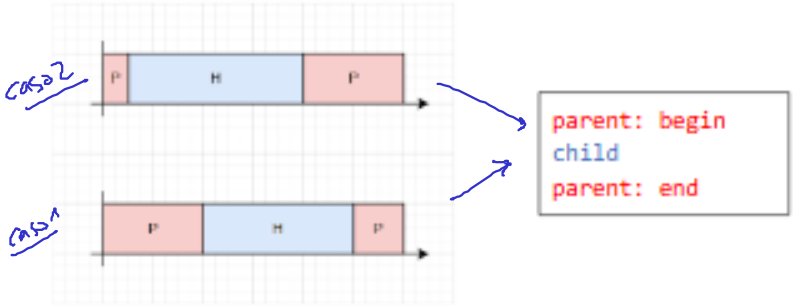



```c
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6    Pthread_mutex_lock(&m);
7    done = 1;
8    Pthread_cond_signal(&c);
9    Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13   printf("child\n");
14   thr_exit();
15   return NULL;
16 }
17
18 void thr_join() {
19   Pthread_mutex_lock(&m);
20   while (done == 0)
21     Pthread_cond_wait(&c, &m);
22   Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26   printf("parent: begin\n");
27   pthread_t p;
28   Pthread_create(&p, NULL, child, NULL);
29   thr_join();
30   printf("parent: end\n");
31   return 0;
32 }
```

Código (link)

Conclusión





Como podemos evitar usar el while???

Solución:

Reimplementación

① thr-exit
② thr-join

~~while()~~

Función thr_exit

```
void thr_exit() {
  Pthread_mutex_lock(&m);
  done = 1;
  Pthread_cond_signal(&c);
  Pthread_mutex_unlock(&m);
}
```

Función thr_join

```
void thr_join() {
  Pthread_mutex_lock(&m);
  while (done == 0)
    Pthread_cond_wait(&c, &m);
  Pthread_mutex_unlock(&m);
}
```

Intentemos reimplementar **thr_join** y thr_exit

5. Reescritura de thr_exit() y thr_join() para no usar while

① ~~while~~ ~~done~~ → no sirve

Función thr_exit

```
void thr_exit() {
  Pthread_mutex_lock(&m);
  done = 1;
  Pthread_cond_signal(&c);
  Pthread_mutex_unlock(&m);
}
```

Función thr_join

```
void thr_join() {
  Pthread_mutex_lock(&m);
  while (done == 0)
    Pthread_cond_wait(&c, &m);
  Pthread_mutex_unlock(&m);
}
```

Reimplementación

Función thr_exit

```
void thr_exit() {
  pthread_mutex_lock(&m);
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

Función thr_join

```
void thr_join() {
  pthread_mutex_lock(&m);
  pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}
```

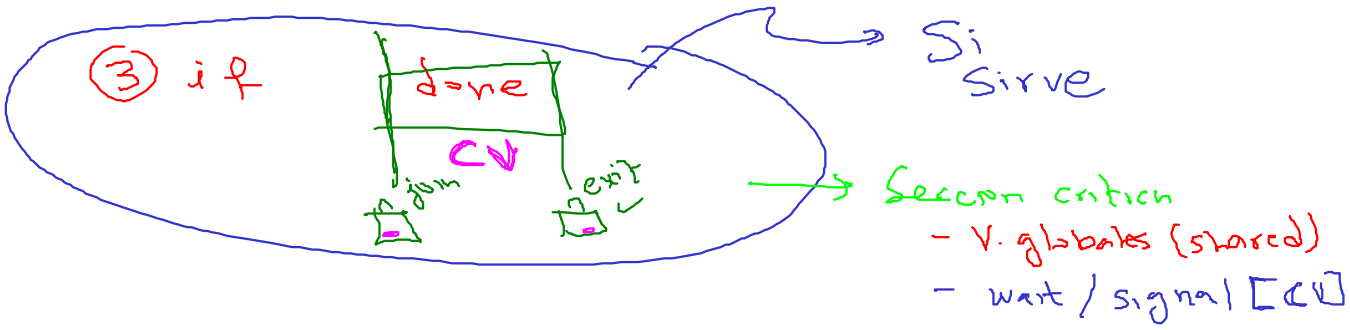
- **Regla de oro**: Además de usar CV es necesario **mantener el estado**.
- Los **CV** se utilizan para señalar hilos cuando cambia el estado.

Función thr_exit

```
void thr_exit() {
  done = 1;
  pthread_cond_signal(&c);
}
```

Función thr_join

```
void thr_join() {
  pthread_mutex_lock(&m)
  if (done == 0) {
    pthread_cond_wait(&c);
  }
  pthread_mutex_unlock(&m);
}
```



Reimplementación del join 3 (link) - Regla de oro: El mutex es importante

- **Regla de oro**: Mantenga el **lock** siempre que llame el **`signal`** o el **`wait`**.
- Se usa el **`mutex`** para asegurarse que no se da una condición de competencia al interactuar con el estado (**done**) y con las llamadas **`wait`** y **`signal`**.

Función thr_exit

```
void thr_exit() {
  pthread_mutex_lock(&m);
  done = 1;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

Función thr_join

```
void thr_join() {
  pthread_mutex_lock(&m)
  if (done == 0) {
    pthread_cond_wait(&c);
  }
  pthread_mutex_unlock(&m);
}
```