Exemple comprenant : threads, mutex, variable de condition, variable spécifique et signaux :

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

void handlerSIGALRM(int sig);
void handlerSIGUSR1(int sig);
void* fctThread1(void* p);
void* fctThread2(void* p);
void fctFinThread(void* p);
void destructeurVarSpec(void* p);

pthread_key_t cle;
int nbThreadsTermines = 0;
pthread_mutex_t mutexNbThreads;
pthread_cond_t condNbThreads;
struct sigaction sigAct;

// ----------------------------------------------------------------------------

int main()
{
	sigset_t mask;
	pthread_t hThread1, hThread2;
	struct timespec temps = { 1, 0 };

	printf("Début du thread principal (%u)\n", (unsigned int)pthread_self());
	fflush(stdout);

	// armer SIGUSR1 et SIGALRM et masquer ces signaux

	sigemptyset(&sigAct.sa_mask);
	sigAct.sa_handler = handlerSIGUSR1;
	sigAct.sa_flags = 0;
	sigaction(SIGUSR1, &sigAct, NULL);

	sigemptyset(&sigAct.sa_mask);
	sigAct.sa_handler = handlerSIGALRM;
	sigAct.sa_flags = 0;
	sigaction(SIGALRM, &sigAct, NULL);

	sigemptyset(&mask);
	sigaddset(&mask, SIGUSR1);
	sigaddset(&mask, SIGALRM);
	sigprocmask(SIG_BLOCK, &mask, NULL);

	// créer les mutexes et la variable spécifique

	pthread_mutex_init(&mutexNbThreads, NULL);
	pthread_cond_init(&condNbThreads, NULL);
	pthread_key_create(&cle, destructeurVarSpec);

	// créer les threads secondaires

	pthread_create(&hThread1, NULL, fctThread1, (void*)1);
	pthread_create(&hThread2, NULL, fctThread2, (void*)2);

	nanosleep(&temps, NULL);
```

```
	// générer les signaux SIGALRM et SIGUSR1

	alarm(3);
	pthread_kill(hThread2, SIGUSR1);

	// attendre la terminaison des 2 threads

	pthread_mutex_lock(&mutexNbThreads);

	while (nbThreadsTermines != 2)
		pthread_cond_wait(&condNbThreads, &mutexNbThreads);

	pthread_mutex_unlock(&mutexNbThreads);

	printf("Fin du thread principal (%u)\n", (unsigned int)pthread_self());
	fflush(stdout);

	pthread_exit(0);

	return 0;
}

// ----------------------------------------------------------------------------

void* fctThread1(void* p)
{
	sigset_t mask;
	int param = (int)p;

	sigemptyset(&mask);
	sigaddset(&mask, SIGALRM);
	sigprocmask(SIG_UNBLOCK, &mask, NULL);

	pthread_cleanup_push(fctFinThread, NULL);

	pthread_setspecific(cle, (const void*)param);

	printf("Début du thread %d (%u)\n", param, (unsigned int)pthread_self());
	fflush(stdout);

	pause();

	printf("Fin du thread %d (%u)\n", param, (unsigned int)pthread_self());
	fflush(stdout);

	pthread_cleanup_pop(1);

	pthread_exit(0);
}

// ----------------------------------------------------------------------------

void* fctThread2(void* p)
{
	sigset_t mask;
	int param = (int)p;

	sigemptyset(&mask);
	sigaddset(&mask, SIGUSR1);
	sigprocmask(SIG_UNBLOCK, &mask, NULL);

	pthread_cleanup_push(fctFinThread, NULL);

	pthread_setspecific(cle, (const void*)param);
```

```
        printf("Début du thread %d (%u)\n", param, (unsigned int)pthread_self());
        fflush(stdout);

        pause();

        printf("Fin du thread %d (%u)\n", param, (unsigned int)pthread_self());
        fflush(stdout);

        pthread_cleanup_pop(1);

        pthread_exit(0);
}

// ------------------------------------------------------------------------

void handlerSIGALRM(int sig)
{
        int varspec = (int)pthread_getspecific(cle);

        printf("SIGALRM pour le thread %d (%u)\n", varspec, (unsigned int)pthread_self());
        fflush(stdout);
}

// ------------------------------------------------------------------------

void handlerSIGUSR1(int sig)
{
        int varspec = (int)pthread_getspecific(cle);

        printf("SIGUSR1 pour le thread %d (%u)\n", varspec, (unsigned int)pthread_self());
        fflush(stdout);
}

// ------------------------------------------------------------------------

void fctFinThread(void* p)
{
        int varspec = (int)pthread_getspecific(cle);

        printf("Fonction de fin pour le thread %d (%u)\n", varspec,
                (unsigned int)pthread_self());
        fflush(stdout);
}

// ------------------------------------------------------------------------

void destructeurVarSpec(void* p)
{
        printf("Destructeur de la variable specifique pour le thread (%u)\n",
                (unsigned int)pthread_self());
        fflush(stdout);

        pthread_mutex_lock(&mutexNbThreads);

        nbThreadsTermines++;

        pthread_mutex_unlock(&mutexNbThreads);

        pthread_cond_signal(&condNbThreads);  // réveiller le thread principal
}
```

Compilation : `g++ prog.c -o prog -lpthread -fpermissive`

Résultat de l'exécution du programme :

```
Début du thread principal (412120896)
Début du thread 2 (403724032)
Début du thread 1 (412116736)
SIGUSR1 pour le thread 2 (403724032)
Fin du thread 2 (403724032)
Fonction de fin pour le thread 2 (403724032)
Destructeur de la variable specifique pour le thread (403724032)
SIGALRM pour le thread 1 (412116736)
Fin du thread 1 (412116736)
Fonction de fin pour le thread 1 (412116736)
Destructeur de la variable specifique pour le thread (412116736)
Fin du thread principal (412120896)
```

Informations sur le code :

- Au niveau des **signaux** :
  - Quand on masque (bloque) des signaux dans le thread principal AVANT de créer les threads secondaires, alors les signaux sont aussi masqués dans les threads secondaires. À partir de là, chaque thread secondaire peut débloquer ou masquer les signaux qu'il souhaite, car chaque thread possède son propre masque de signal.

    Dans cet exemple, le thread principal masque les signaux SIGUSR1 et SIGALRM, puis uniquement le thread 1 débloque le signal SIGALRM et uniquement le thread 2 débloque le signal SIGUSR1. Ainsi, seul le thread 1 pourra recevoir le signal SIGALRM et seul le thread 2 pourra recevoir le signal SIGUSR1.

  - L'armement d'un signal (= attacher un handler de signal à un signal) est global pour tous les threads, car le handler attaché à un signal est le même pour tous les threads du processus.

  - La fonction `alarm(n)` déclenche le signal SIGALRM après n secondes.

  - La fonction `pthread_kill()` permet d'envoyer un signal à un thread spécifique.

- Une **section critique** est une portion de code dans laquelle il n'y aura jamais plus d'un thread à la fois. Il faut utiliser des sections critiques quand il y a accès à des ressources partagées par plusieurs threads.

  Un mutex permet de créer une section critique au sein de laquelle on manipule une variable globale qui est partagée entre plusieurs threads. **Tout accès en lecture ou en écriture à une variable partagée entre plusieurs threads nécessite l'usage d'une section critique (l'accès à chaque variable partagée est protégé avec un mutex spécifique).** Avant l'accès en lecture ou en écriture à la variable partagée, on débute la section critique avec la fonction `pthread_mutex_lock()` et on termine la section critique avec la fonction `pthread_mutex_unlock()`.

- Avec les macros `pthread_cleanup_push()` et `pthread_cleanup_pop()`, il est possible de spécifier, pour chaque thread, une fonction à exécuter au moment où ce thread se termine.

- Une **variable de condition** permet de créer de la synchronisation entre threads (voir la page 80 dans les notes du cours théorique).

  Dans ce programme, le thread principal utilise une variable de condition pour attendre la fin de l'exécution des 2 threads.

  1. Au départ, la variable globale nbThreadsTermines contient la valeur 0.

  2. Avec le code suivant, le thread principal attend tant que la variable globale nbThreadsTermines n'est pas à 2 indiquant que les 2 threads secondaires sont terminés.

```
pthread_mutex_lock(&mutexNbThreads);

while(nbThreadsTermines != 2)
      pthread_cond_wait(&condNbThreads, &mutexNbThreads);

pthread_mutex_unlock(&mutexNbThreads);
```

  3. À la fin de l'exécution de chacun des 2 threads secondaires, la variable globale nbThreadsTermines est incrémentée, puis le thread principal est réveillé avec `pthread_cond_signal()` :

```
pthread_mutex_lock(&mutexNbThreads);

nbThreadsTermines++;

pthread_mutex_unlock(&mutexNbThreads);

pthread_cond_signal(&condNbThreads);
```

- Une **variable spécifique** est une variable accessible par toutes les fonctions utilisées par un thread (en ce sens, elle est un peu globale) mais qui reste propre à chaque thread (en ce sens, elle est plutôt locale).

  Une variable spécifique permet, par exemple, de partager une donnée entre la fonction du thread et un handler de signal. Dans cet exemple, la fonction `pthread_key_create()` est appelée 1 seule fois lors de la création des ressources dans le thread principal. Le 1$^{er}$ paramètre transmis à `pthread_key_create()` permet d'obtenir une clé qui donnera accès à la variable spécifique. Le deuxième paramètre transmis à `pthread_key_create()` permet de définir une fonction de terminaison dont le rôle est de libérer les ressources associées à la clé. Cette fonction de terminaison est exécuté au moment de la terminaison de chaque thread qui a utilisé la variable spécifique avec la fonction `pthread_setspecific()`. Dans le handler du signal, on récupère la valeur de la variable spécifique avec la fonction `pthread_getspecific()`.

Exercice :

1. Ajouter un 3$^{e}$ thread secondaire et un 4$^{e}$ thread secondaire.

2. Ces 2 nouveaux threads secondaires doivent fonctionner exactement comme les 2 autres threads secondaires : usage d'une variable spécifique, indiquer leur terminaison via la variable nbThreadsTermines et utiliser une fonction de terminaison (pthread_cleanup_push, pthread_cleanup_pop).

3. Par contre, le 3$^{e}$ thread secondaire doit se terminer à la réception du signal SIGUSR2 et le 4$^{e}$ thread secondaire doit se terminer à la réception du signal SIGINT (signal déclenché avec CTRL+C dans le terminal).