

# Threading

## Explicación de práctica 3

Sistemas Operativos

Facultad de Informática  
Universidad Nacional de La Plata

2025



Contenido:



Repositorio con código para la práctica



<https://gitlab.com/unlp-so/codigo-para-practicas>

# Procesos



# Procesos: Conceptos

- Son administrados por el scheduler del SO.
- Si hay múltiples procesadores, pueden ejecutarse en paralelo:
  - Múltiples chips físicos
  - Múltiples cores en el mismo chip
  - Hyperthreading
  - Combinaciones de los anteriores
- Por defecto no comparten memoria R/W.
  - Al crear un nuevo proceso con ``fork()``, se comparten las páginas de memoria mediante Copy-On-Write (COW).
  - Copy-On-Write: si se intenta escribir, se duplica la página.
- Típicamente se crean con ``fork()`` (syscall `clone()` o syscall `fork()`)
- PID: Process Identifier



# Procesos: Comunicación (IPC)

- Pipes y FIFOs (named pipes)
- Sockets:
  - UNIX
  - TCP
  - UDP
- Memoria compartida (``shm_open`` / ``mmap``)
- Archivos



# Procesos: Sincronización

- Memoria compartida:
  - Semáforos POSIX (`sem open`)
  - Mutex de PThreads
- Pipes, FIFOs, Sockets y archivos:
  - E/S sincrónica (bloqueo en lectura/escritura)
- Archivos (<https://gavv.net/articles/file-locks/>):
  - flock() Advisory lock (no es obligatorio)
  - lockf() Lock POSIX por sección de archivo
  - fcntl() Advisory locks (estable) y mandatory locks (buggy)



# Procesos: Funciones básicas

- `fork()` crea un nuevo proceso (con el mismo código, recordar que se comparten las páginas hasta alguna escritura -COW-)
  - Retorna 0 en el hijo
  - Retorna el PID del hijo en el padre (útil para esperar al hijo con `waitpid()`)
- `exec*()` ejecuta un nuevo programa en el proceso actual (reemplaza las páginas del proceso)
- `waitpid(pid, NULL, 0)` espera la finalización de un hijo
- `wait(NULL)` espera la finalización de cualquier hijo
- `getpid()` retorna el PID del proceso actual

Referencias: ver manpages (ej: `man fork`)





# Procesos: fork

```
// includes ...
int main() {
    int child_pid = fork();
    if (child_pid == 0) {
        // Child process
        printf("Child process: PID = %d, my parent is: %d\n",
               getpid(), getppid());
    } else {
        // Parent process
        printf("Parent process: PID = %d, my parent is: %d\n",
               getpid(), getppid());
        waitpid(child_pid, NULL, 0); // Wait child to finish
    }
}
```



# Procesos: Zombies

- Un child al que no se le hizo wait() se convierte en “zombie” al terminar
- El Kernel mantiene información de ese proceso
- Los procesos zombie consumen una entrada en la tabla de procesos del Kernel
- Si un proceso padre termina, sus hijos “zombies” (si los hay) son adoptados por el proceso “init” por defecto.



# Procesos: Zombies

```
int main() {
    if (fork() < 0) {
        perror("fork failed");
        return 1;
    }
    // Child process
    if (getpid() == 0) {
        // Child process
        // ... (code for child process) ...
    }
    // Parent process
    // ... (code for parent process) ...
}
```

```
$ ps ax | grep zombie
3249316 pts/1    S+      0:00  ./zombie
3249317 pts/1    Z+      0:00  [zombie] <defunct>
3249414 pts/4    S+      0:00  grep zombie
```

# Procesos: fork + exec

```
int main() {  
    int child_pid = fork();  
    if (child_pid == 0) { // Child process  
        printf("Child process: PID = %d, my parent is: %d\n",  
               getpid(), getppid());  
        execl("/bin/ls", "ls", "--color=always", NULL);  
    } else { // Parent process  
        printf("Parent process: PID = %d, my parent is: %d\n",  
               getpid(), getppid());  
        waitpid(child_pid, NULL, 0); // Wait child process  
        printf("Parent process: Child finished\n");  
    }  
}
```

```
 $ ./forkexec
```

```
Parent process: PID = 3254289, my parent is: 3248795
```

```
Child process: PID = 3254290, my parent is: 3254289
```

```
fork fork.c forkexec forkexec.c Makefile zombie zombie.c
```

```
Parent process: Child finished
```

# Kernel Level Threads

Native POSIX Threads Library (Linux)



# Native POSIX Threads Library - NPTL (KLT): Conceptos

- Más conocidos como la implementación actual de PThreads en Linux.
- Son administrados por el scheduler del sistema operativo.
- Pueden ejecutarse en paralelo en múltiples procesadores:
  - Múltiples chips físicos
  - Múltiples cores en el mismo chip
  - Hyperthreading
  - Combinaciones de los anteriores
- Comparten el espacio de direcciones (código, datos, heap), pero cada hilo tiene su propio stack.
- Se crean típicamente con ``pthread_create``.
- TID: Thread Identifier



# PThreads NPTL (KLT): Comunicación y Sincronización

- Comunicación:
  - Memoria compartida (variables globales o punteros compartidos)
- Sincronización:
  - Mutex (``pthread_mutex_t``)
  - Condicionales (``pthread_cond_t``)
  - Barreras (``pthread_barrier_t``)
  - Semáforos (``sem_init``)



# PThreads NPTL (KLT): Funciones básicas

- `pthread_create(&t, NULL, func, NULL)` crea un nuevo thread “t”, invoca a `func()` dentro de ese thread (syscall `clone3()`)
- `pthread_join(t, NULL)` espera la finalización del thread “t”.
  - Si no se hace join el thread queda zombie
  - Los recursos de un thread zombie se liberan al terminar el proceso que lo creó.
  - También se puede hacer `pthread_detach()` para que no necesite join
- `pthread_self()` retorna el id del thread actual (a nivel PThreads).
- `gettid()` retorna el id del thread actual (a nivel SO) es distinto al anterior.

Referencias: ver manpages (ej: `man pthreads`)





# PThreads NPTL (KLT): pthread\_create

```
void *task(void *args) {
    printf("Child thread: PID = %u, THREAD_ID = %d\n",
        getpid(), gettid());
    return NULL;
}
int main() {
    pthread_t thread;
    printf("Parent thread: PID = %u, THREAD_ID = %d\n",
        getpid(), gettid());
    // Create new kernel level thread
    if (pthread_create(&thread, NULL, task, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }
    // Wait for the child thread to finish
    if (pthread_join(thread, NULL) != 0) {
        perror("pthread_join");
        return 1;
    }
    return 0;
}
```

```
 $ ./pthread
```

```
Parent thread: PID = 3257449, THREAD_ID = 3257449
Child thread: PID = 3257449, THREAD_ID = 3257450
```

# User Level Threads



# Threads ULT (User-Level Threads): Conceptos

- Son administrados por un scheduler en espacio de usuario.
- Ejemplos:
  - GNU Pth
  - greenlets (Python) / asyncio por defecto (Python)
  - coroutines
- No se ejecutan en paralelo sobre múltiples núcleos.
- Scheduling:
  - Non-preemptive: Cooperativo (cada hilo debe ceder voluntariamente el control). Lo más común.
  - Preemptive: Se usa un quantum. Es posible pero no hay implementaciones populares (<https://dl.acm.org/doi/10.1145/3437801.3441610>)
- Comparten memoria R/W.
- Una llamada bloqueante (como ``read()``) bloquea a todos los ULT.



# Threads ULT (User-Level Threads): Com. y sincro.

- Comunicación:
  - Memoria compartida (variables globales o punteros compartidos)
- Sincronización:
  - Mutexes, semáforos y otros mecanismos provistos por la biblioteca de ULT



# Threads ULT (User-Level Threads): Ventajas / Desventajas

- Ventajas:
  - Muy bajo costo de creación y cambio de contexto
  - Independientes del sistema operativo
- Desventajas:
  - No hay paralelismo real (salvo modelos híbridos M:N)
  - Las llamadas bloqueantes afectan a todos los ULT



# Crear un hilo ULT usando GNU PTh

- `pth_init()` inicializa la biblioteca
- `t = pth_spawn(PTH_ATTR_DEFAULT, func, NULL)` crea un nuevo ULT “t”
- `pth_join(t, NULL)` espera la finalización del thread “t”.
- `pth_self()` retorna el id del thread actual.

Referencias: ver manpages (man pth)



# ULT usando GNU PTh: pth\_spawn

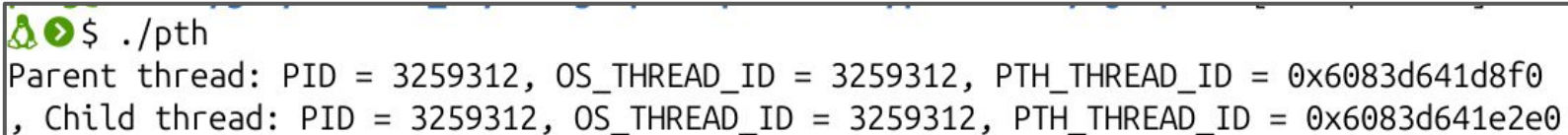
```
void *task(void *args) {
    printf("Child thread: PID = %d, OS_THREAD_ID = %d, PTH_THREAD_ID = %p\n", ",
        getpid(), gettid(), pth_self());
    return NULL;
}
int main() {
    pth_t thread;
    // Initialize the GNU Pth library
    if (pth_init() == 0) { perror("pth_init"); return 1; }

    printf("Parent thread: PID = %d, OS_THREAD_ID = %d, PTH_THREAD_ID = %p\n", ",
        getpid(), gettid(), pth_self());

    // Create a new thread
    thread = pth_spawn(PTH_ATTR_DEFAULT, task, NULL);
    if (thread == NULL) { perror("pth_spawn"); return 1; }
    // Wait for the child thread to finish

    if (pth_join(thread, NULL) == 0) { perror("pth_join"); return 1; }
    // Finalize the GNU Pth library

    pth_kill();
    return 0;
}
```



```
$ ./pth
Parent thread: PID = 3259312, OS_THREAD_ID = 3259312, PTH_THREAD_ID = 0x6083d641d8f0
, Child thread: PID = 3259312, OS_THREAD_ID = 3259312, PTH_THREAD_ID = 0x6083d641e2e0
```

# Algunas implementaciones de threading





# Procesos y KLTs en lenguajes interpretados

- Fork o clone del intérprete completo.
- Necesidad de sincronización de estructuras de datos del intérprete (locks).
- Conteo de referencias para garbage collector corre todo el tiempo (requiere lock).
- GIL (Global Interpreter Lock): CPython y MRI (implementaciones oficiales de Python y Ruby)
  - Ejecuta un hilo por vez.
  - Simplifica la implementación del intérprete y de algunas bibliotecas nativas.
  - Aceptable para tareas IO-Bound.
  - Poco conveniente para CPU-Bound.



# Implementaciones de threading

- C
  - GNU/Linux Native POSIX Threads Library: KLT 1:1
  - GNU Pth: ULTs non-preemptive
  - Sandialabs qthreads: ULTs M:N (M ULTs en N KLTs)  
<https://github.com/sandialabs/qthreads>
- Python / Ruby
  - Módulo de threading nativo: KLT (limitado por GIL en CPython / MRI)
  - Bibliotecas de greenlets (Fibers en Ruby): ULT
  - Async: interfaz de alto nivel para concurrencia, por defecto ULT
  - CPython se está trabajando en eliminar el GIL o hacerlo opcional.
  - Paralelismo: Procesos
- Go
  - Goroutines ULTs M:N (M ULTs en N KLTs) <https://go.dev/talks/2012/concurrency.slide>



# Práctica guiada



# Práctica guiada

- Se provee un repositorio con ejemplos en C y Python.
- El repositorio cuenta con una serie de Makefiles para simplificar la compilación y ejecución de los ejemplos.
- Los ejemplos plantean distintos escenarios usando Subprocesos, User Level Threads y Kernel Level Threads.
- Los ejercicios implican:
  - Leer el código (al menos superficialmente siguiendo los comentarios del mismo)
  - Ejecutarlo y monitorearlo con htop o strace
  - Razonar porqué el ejemplo se comporta de determinada forma y compararlo con la ejecución de otros ejemplos.



# Herramientas necesarias

- git: Para clonar el repo con ejemplos.
- build-essential: gcc, make, as, ld, libc6-dev, etc... Lo mínimo necesario para compilar código C.
- libpth-dev: Biblioteca que provee ULTs para C.
- strace: Herramienta para monitorear las syscalls invocadas.
- python3 + python3-venv: Intérprete de Python.
- htop: Lo usaremos para monitorear el uso de los cores de la CPU.
- podman: Lo usaremos para ejecutar un intérprete de Python con el GIL deshabilitado (dentro de un container).
- Varias ventanas de la terminal al mismo tiempo para usar htop mientras ejecutamos los ejemplos.



# Instalar dependencias y descargar ejemplos

```
# su -  
# apt update  
# apt install git  
# exit  
$ git clone https://gitlab.com/unlp-so/c...git  
$ cd codigo-para-practicas/practica3  
$ su -c ./instalar_deps.sh
```



# Probar ejemplos

```
$ cd <DIRECTORIO DEL EJEMPLO>
$ make # compila todo el código C del dir.
$ make help # muestra ayuda (algunos dir)
$ ./nombre_ejemplo # ejecutar binarios
$ # Los siguientes son para ejecutar los scripts
$ # Python
$ make run_klt_py
$ make run_ult_py
$ make run_klt_py_nogil
```



¿Preguntas?

