Heejae Kim
(adpp00@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

2023.12.05

Project #5:
Native thread support

Thread and Process

- A thread is a lightweight unit of execution within a process
- A thread shares the process's resources but operating independently in terms of program counter, registers, stack, etc

- A process is a self-contained execution environment, typically comprising its own memory space, program code, and other resources

- More details are on lecture slide and github specification

Project#5: Native Thread Support

- In this project, you have to
  - 1. Prepare the xv6 kernel for native thread support (40 points)
  - 2. Support user-level threads (50 points)
  - 3. Submit design documents (10 points)

- Due date is 11:59(PM), December 22nd (Friday)

1. Prepare for Native Thread Support

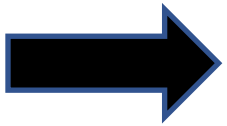
- We can view the "process" in the current xv6 kernel as the process with only one thread.
- Our goal is to make the process have more than one thread.
- Presently, struct proc in xv6 holds all the information pertinent to both a process and its thread
- Your task is to isolate the data structures required for each "thread" from those used by the "process“

1. Prepare for Native Thread Support

- Struct thread
  - Struct thread is a new data structure to store thread-specific information.
  - Any process-wide data will remain in struct proc
  - The struct proc will include a struct thread.

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  ...

  int pid;                     // Process ID
  ...

  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  ...

};
```

Original xv6 code of struct proc



```
struct thread {
  int   tid;    // thread id
  int   state;  // thread state
  ...
};

struct proc {
  int   pid;    // process id
  ...

  struct thread thr[NTH];
};
```

Modified xv6 code

1. Prepare for Native Thread Support

- Modifying struct proc in this way will break the kernel code
  - For example, the kernel must keep track of the state of each thread and the scheduler is responsible for selecting the next thread to be executed
- Your task is to modify the xv6 kernel code to ensure that it functions correctly even after struct thread has been separated from struct proc
- For this part, you can assume that the number of threads per process(NTH) is fixed to one
  - Initially, the value of NTH is set to 4 in Makefile
- Any existing program should run correctly on the new xv6 kernel, including user-level program *usertests*

2. Support user-level threads

- In this part, you have to modify your code to support
  - 2.1. SNU Threads (sthreads) APIs
  - 2.2. Trapframe Handling
  - 2.3. Interactions with Process-oriented System Calls

- For part 2, please change the value of NTH to more than one
  - NTH variable is in Makefile

2.1. SNU Threads (sthreads) APIs

- Your task is to enable user-level threads (called sthreads) by implementing following APIs

- The system call numbers for the following functions have been pre-assigned, ranging from 24 to 27
  - int sthread_self(void);
  - int sthread_create(void (*func)(), void *arg);
  - void sthread_exit(int retval);
  - int sthread_join(int tid, int *retval);

2.1. SNU Threads (sthreads) APIs

- int sthread_self(void);
  - This function returns the thread ID of the calling thread.
  - The return ID is represented as an integer type.
  - This function always succeeds

2.1. SNU Threads (sthreads) APIs

- int sthread_create(void (*func)(), void *arg);
  - Creates a new thread within the calling process
  - The thread begins execution at func() with arg provided as the sole argument to func()
  - Thread ID (t->tid) of created thread is assigned using the following formula
    - **t->tid = p->pid * 100 + n**
      - p->pid is the process ID of the process to which the thread belongs.
      - n is a monotonically increasing number that starts from 0 within that process.
  - This function returns the thread ID of the newly created process on success
  - This function returns -1 on error

2.1. SNU Threads (sthreads) APIs

- void sthread_exit(int retval);
  - This function terminates the calling thread
  - This function returns a value via retval
  - The return value retval is available to another thread in the same process that calls sthread_join()
  - If the last thread in a process executes sthread_exit(), the associated process should also terminate, freeing up all resources allocated to that process.

2.1. SNU Threads (sthreads) APIs

- int sthread_join(int tid, int *retval);
  - This function waits for the thread specified by tid to terminate
  - If that thread has already terminated, returns immediately
  - If retval is not NULL (0), then sthread_join() copies the exit status of the target thread into the location pointed to by retval
  - If multiple threads simultaneously try to join with the same thread, the results are undefined.
  - This function returns 0 on success
  - This function returns -1 if the target has already terminated or the target thread is not found

2.2. Trapframe Handling

- The current xv6 uses a fixed memory region in the virtual address space to preserve the user context across traps
- However, when a process has multiple threads, it is essential to maintain the user context of **each individual thread** across traps
- Consequently, a significant challenge in implementing multi-thread support in xv6 is to ensure each thread operates with its own dedicated trapframe

2.2. Trapframe Handling

- We solve this problem by saving the address of trapframe
  - Saves the address of the corresponding trapframe to the sscratch register whenever a thread returns to the user space
  - Slightly modified the usertrapret() function so that it passes the trapframe address to the userret()
  - At the very beginning of the userret() function, the address is saved to the sscratch register

2.2. Trapframe Handling

- When a trap occurs in the user space, the control is transferred to the uservec() function
- Previously, xv6 has initialized a0 register with the constant TRAPFRAME after backing up the previous value of the a0 register to sscratch
- Now, the trapframe address for the currently running thread is stored in the sscratch register
- Therefore, we need to swap the value of the sscratch register and the a0 register atomatically

2.2. Trapframe Handling

- If we execute the **csrrw a0, sscratch, a0** instruction, the value in sscratch is put into a0 while the old value of a0 is stored in sscratch simultaneously
- After this instruction, we can freely use the trapframe to save the user registers used by the current thread.

- All you need to do is to pass the correct trapframe address at the end of the usertrapret() function allocated for the thread currently

2.3 Interactions with Process-oriented System Calls

- fork():
  - If one of the threads invokes fork(), only the thread that made the call is duplicated in the new process,
  - That thread becomes the default thread in that process.

- exec():
  - If one of the threads executes exec(), only the thread that initiated the call will survive, becoming the default thread in that process.
  - This thread starts its execution from the entry point of the new program, while all the other threads in the process are terminated.

2.3 Interactions with Process-oriented System Calls

- exit():
  - If any thread within a process calls exit(), all the threads are terminated, and the associated process is subsequently removed from the system.
  - A process can also be terminated when its last thread calls the sthread_exit() function. In this scenario, the behavior should be identical to that of the process executing exit(-1).

- kill():
  - If any process is killed by another process via kill(), all the threads within the process are terminated, and the associated process is subsequently removed from the system.

3. Design document (10 points)

- In this project, you need to submit a report explaining your implementation (in a single PDF file)
- These must be included in your report
  - What information is maintained in the struct thread and why?
  - Are there any new variables introduced in the struct proc and struct thread? Why?
  - How are the trapframes, user stacks, and kernel stacks managed?
  - Show the pseudocode for sthread_create(), sthread_exit(), sthread_join() and how to read the value returned by sthread_exit()
  - If you modify the existing system calls such as fork(), exec(), exit(), kill(), wait(), etc., explain why
  - What was the hardest part of this project?

Skeleton Code

- You should work on the pa5 branch as follows:
  $ git clone https://github.com/snu-csl/xv6-riscv-snu
  $ cd xv6-riscv-snu
  $ git checkout pa5

- Then, you have to set your STUDENTID in the Makefile

- The skeleton code includes four user-level programs
  - source code is available in ./user/thread1.c ~ ./user/thread4.c, respectively.
  - You can use these programs to test your implementation.

Restrictions

- You can assume that the maximum value of NTH is 8
  - NTH variable is in the Makefile

- Submitting either the unmodified or slightly modified xv6 code might allow you to pass the test cases in Part 1 of this project.
  - Any attempt to do so will result in a penalty score of -10 points
  - TAs will manually review your code to ensure that your implementation is in line with the intended purpose of this project assignment.

Restrictions

- Your implementation should work on multi-core systems.
  - The number of CPUs is already set to 4 in the Makefile.
  - If your implementation works only on a single core, you may receive only half of the points.
- Do not add any other system calls.
- You only need to modify those files in the ./kernel directory.
  - Changes to other files will be ignored during grading.

Tips

- Read xv6 book
  - Chapter 2,3,4,6,7 to process management and trap handling in xv6.

Submission

- Perform the make submit command to generate a compressed tar file
- Upload this tar file + report to the submission server
- The total number of submissions will be limited to 30
- Only the version marked FINAL will be considered
- Please remove all the debugging outputs before you submit.

Sample Output 1

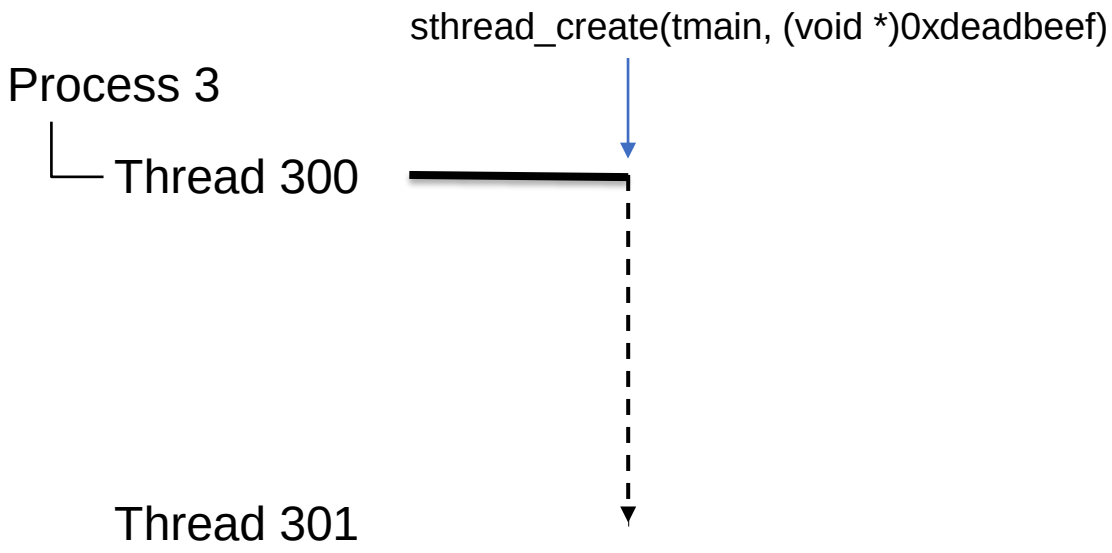
- user/thread1.c

- There will be two threads, whose tids are 300 and 301, respectively

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void tmain(void *arg)
{
  long i = (long) arg;

  sleep(1);
  printf("tid %d: got 0x%x\n", sthread_self(), i);
  sthread_exit(0x900dbeef);
}

int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

Sample Output 1

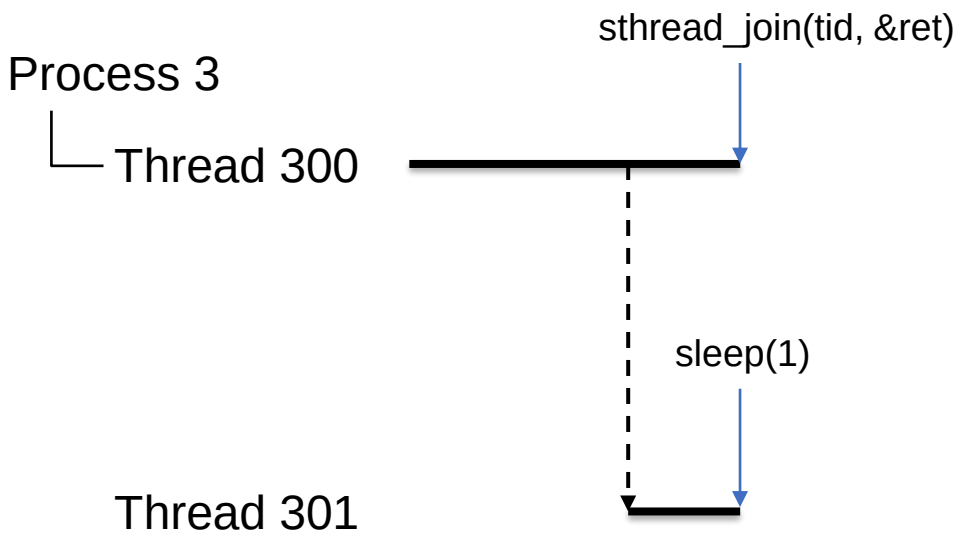
- Thread 300 calls sthread_create(tmain, (void *)0xdeadbeef)
  - Thread 300 creates new thread, thread 301
  - Thread 300 passes the value 0xdeadbeef to thread 301




```c
int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

Sample Output 1

- Thread 300 calls sthread_join(tid, &ret)
  - Thread 300 waits for the thread 301 to terminate
- Thread 301 sleeps

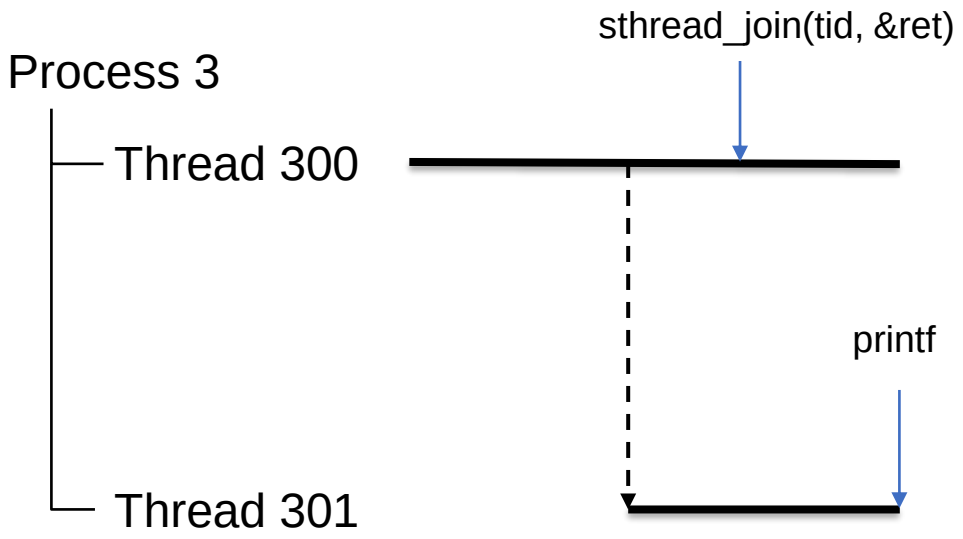



```c
int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

```c
void tmain(void *arg)
{
  long i = (long) arg;

  sleep(1);
  printf("tid %d: got 0x%x\n", sthread_self(), i);
  sthread_exit(0x900dbeef);
}
```

Sample Output 1

- Thread 301 wakes up and calls printf
  - Prints "tid 301: got 0xDEADBEEF"

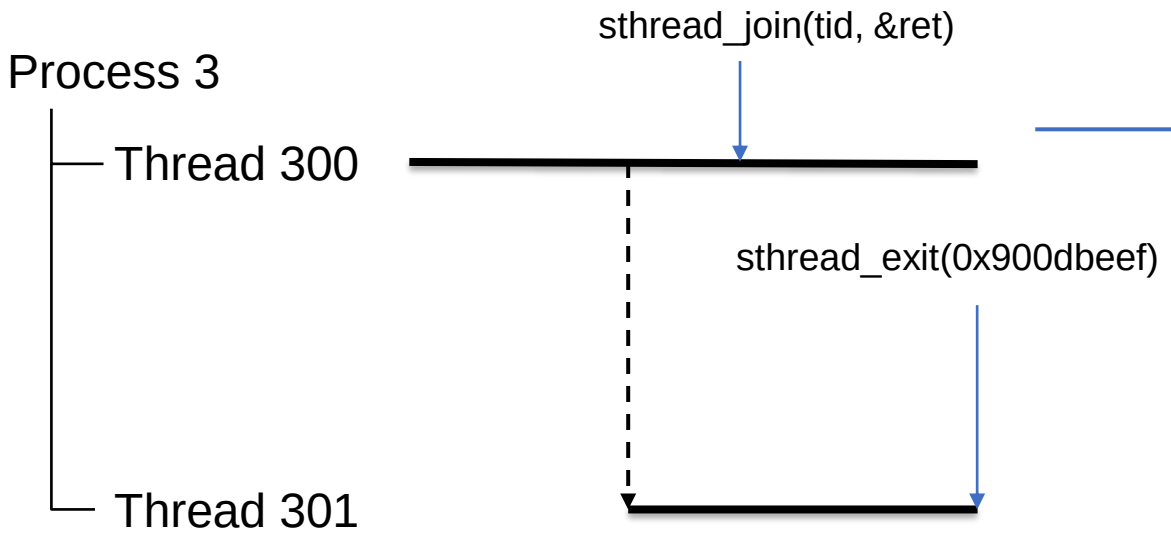



```c
int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

```c
void tmain(void *arg)
{
  long i = (long) arg;

  sleep(1);
  printf("tid %d: got 0x%x\n", sthread_self(), i);
  sthread_exit(0x900dbeef);
}
```

Sample Output 1

- Thread 301 calls sthread_exit(0x900dbeef)
  - Thread 301 exits with the value 0x900dbeef

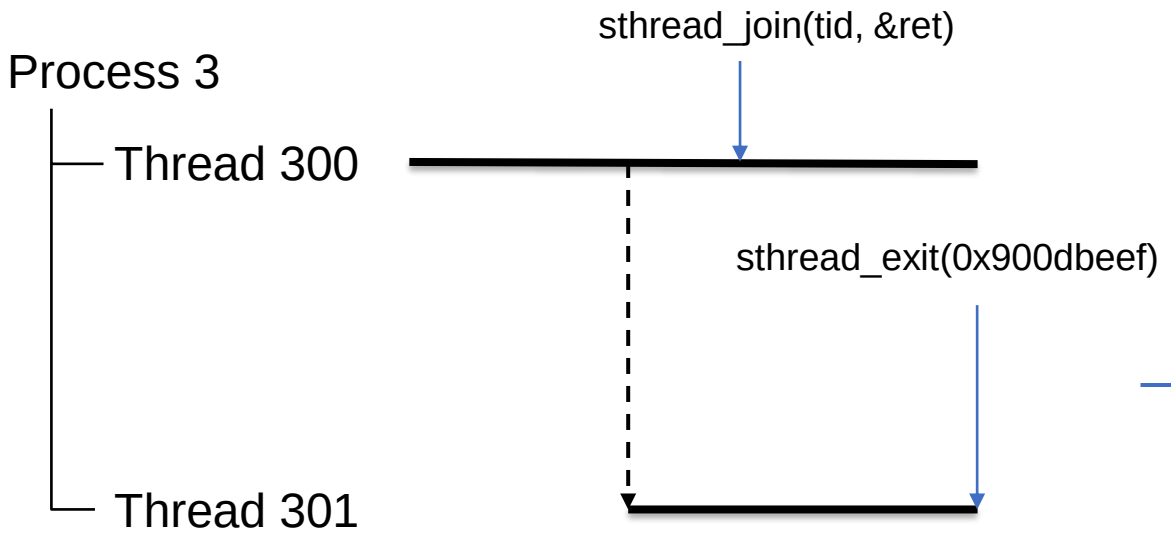



```c
int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

```c
void tmain(void *arg)
{
  long i = (long) arg;

  sleep(1);
  printf("tid %d: got 0x%x\n", sthread_self(), i);
  sthread_exit(0x900dbeef);
}
```

Sample Output 1

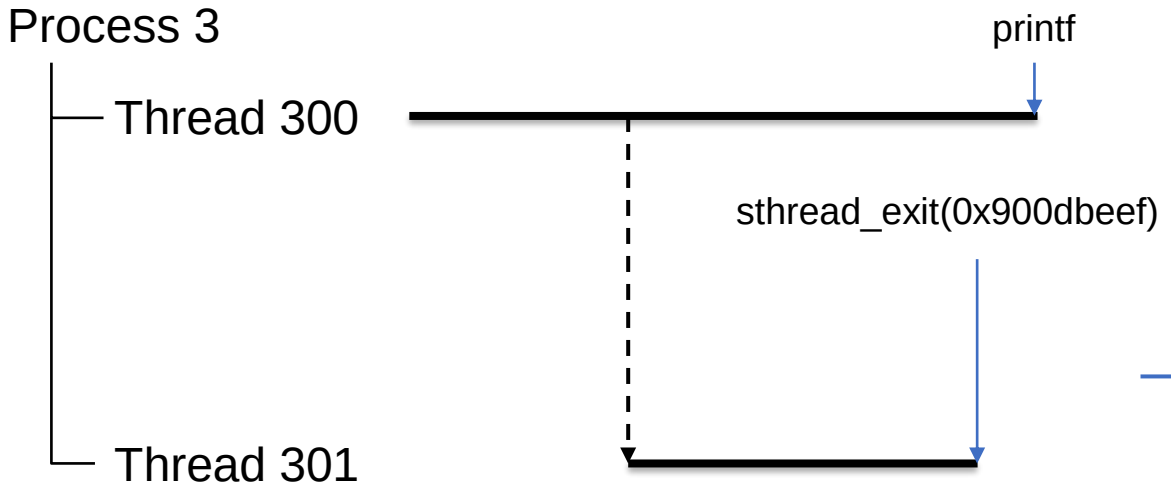
- Thread 300's sthread_join(tid, &ret) returned
  - sthread_join returns 0
  - sthread_join copies the exit status 0x900dbeef to the ret




```c
int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

Sample Output 1

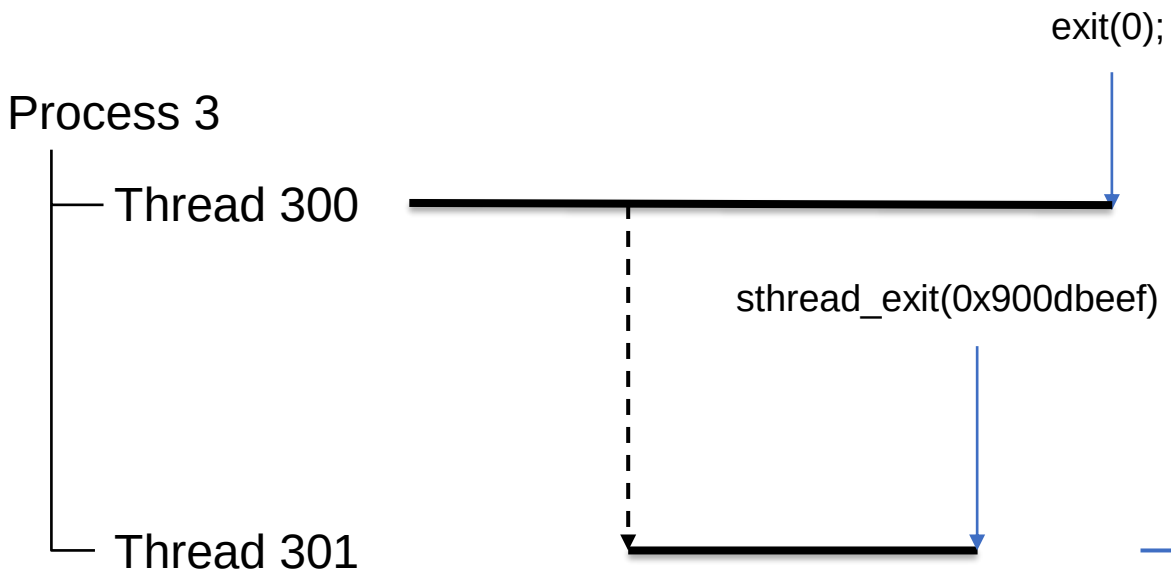
- Thread 300 calls printf
  - Prints "tid 300: got 0x900DBEEF"




```c
int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

Sample Output 1

- Thread 300 calls exit(0);
  - Process is terminated




```c
int
main(int argc, char *argv[])
{
  int tid;
  int ret;

  tid = sthread_create(tmain, (void *)0xdeadbeef);
  sthread_join(tid, &ret);
  printf("tid %d: got 0x%x\n", sthread_self(), ret);

  exit(0);
}
```

Sample Output 2

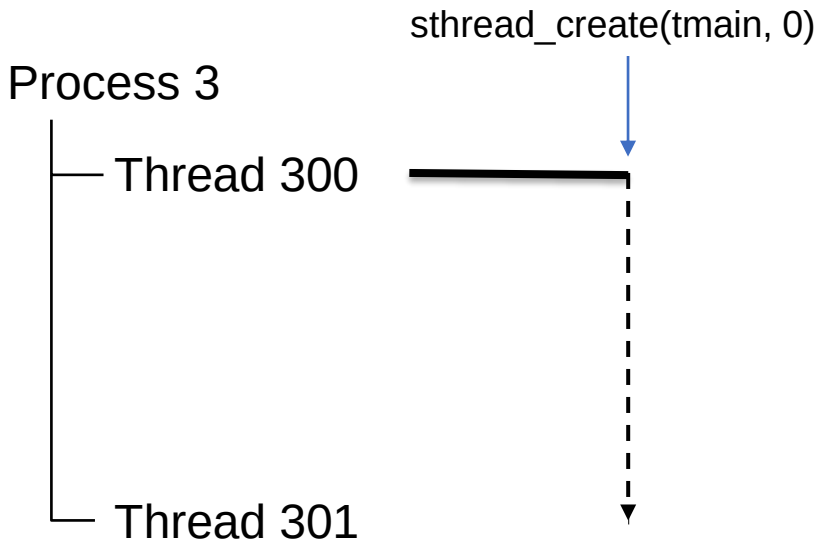
- user/thread2.c

- There will be two threads, whose tids are 300 and 301, respectively

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void tmain(void *arg)
{
  sleep(10);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}

void
main(int argc, char *argv[])
{
  sthread_create(tmain, 0);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 2

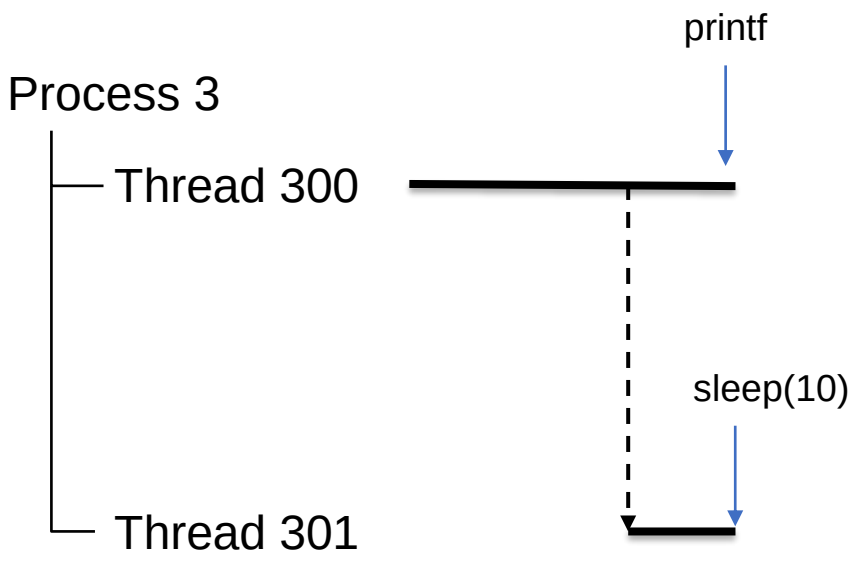
- Thread 300 calls sthread_create(tmain, 0)
  - Thread 300 creates new thread, thread 301
  - Thread 300 passes the value 0 to thread 301




```c
void
main(int argc, char *argv[])
{
  sthread_create(tmain, 0);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 2

- Thread 300 calls printf
  - Prints "Thread 300 is exiting"
- Thread 301 sleeps

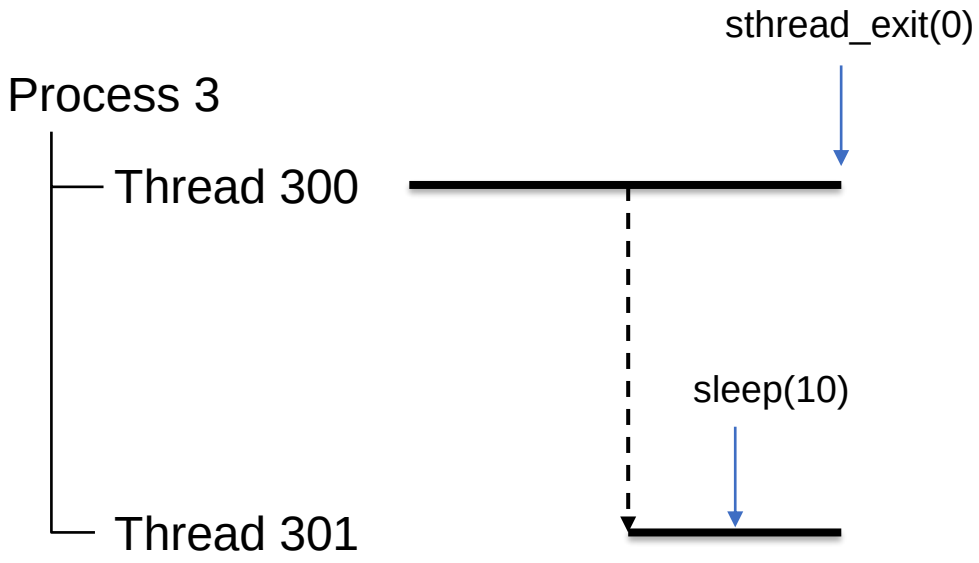



```c
void
main(int argc, char *argv[])
{
  sthread_create(tmain, 0);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

```c
void tmain(void *arg)
{
  sleep(10);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 2

- Thread 300 calls sthread_exit(0)
  - Thread 300 exits with the value 0

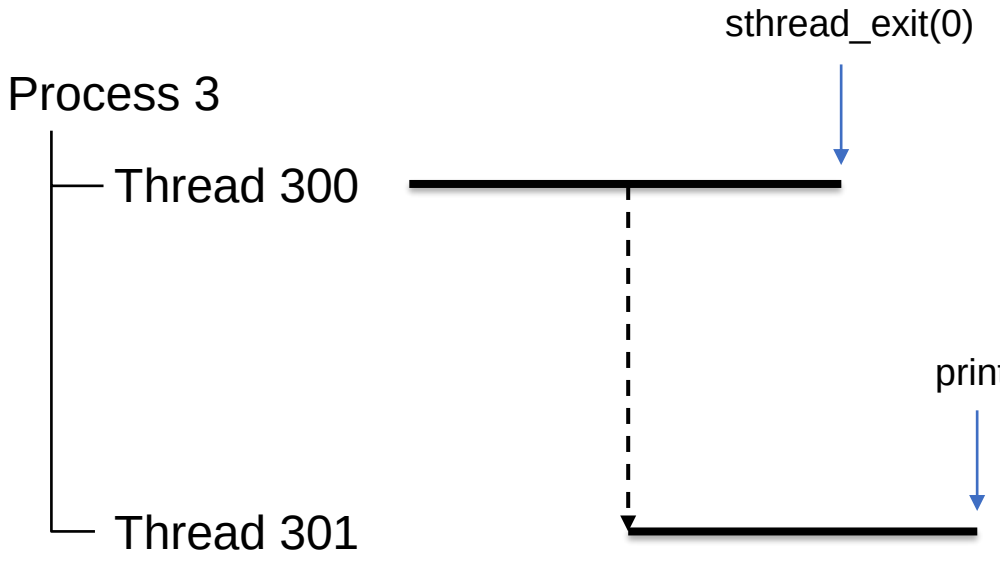



```c
void
main(int argc, char *argv[])
{
  sthread_create(tmain, 0);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

```c
void tmain(void *arg)
{
  sleep(10);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 2

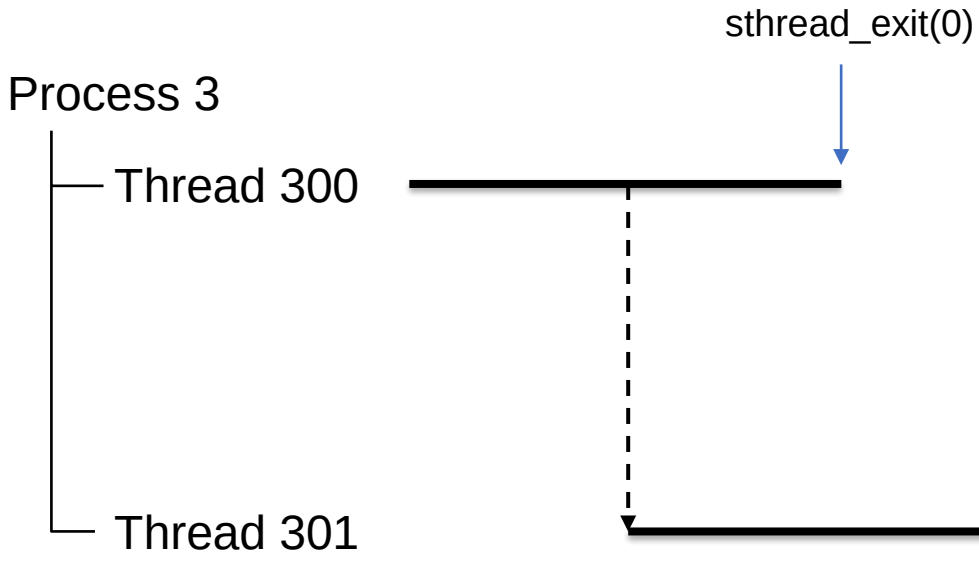
- Thread 301 wakes up, and calls printf
  - Prints "Thread 301 is exiting"

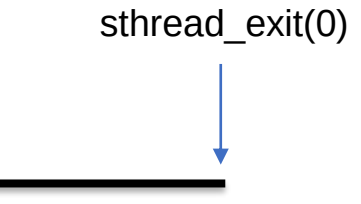



```
void tmain(void *arg)
{
  sleep(10);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 2

- Thread 301 calls sthread_exit(0)
  - Thread 301 is the last thread in a process, so process 3 is also terminated






```
void tmain(void *arg)
{
  sleep(10);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

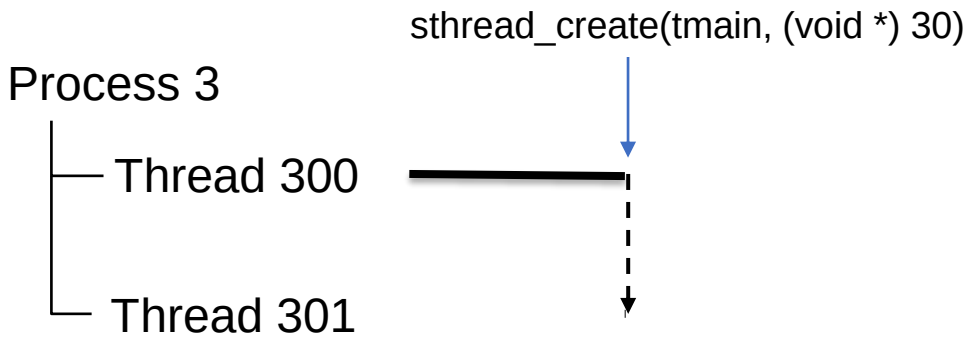
- user/thread3.c

- There will be four threads, whose tids are 300, 301, 302 and 303, respectively

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}

void
main(int argc, char *argv[])
{
  sthread_create(tmain, (void *)30);
  sthread_create(tmain, (void *)10);
  sthread_create(tmain, (void *)20);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

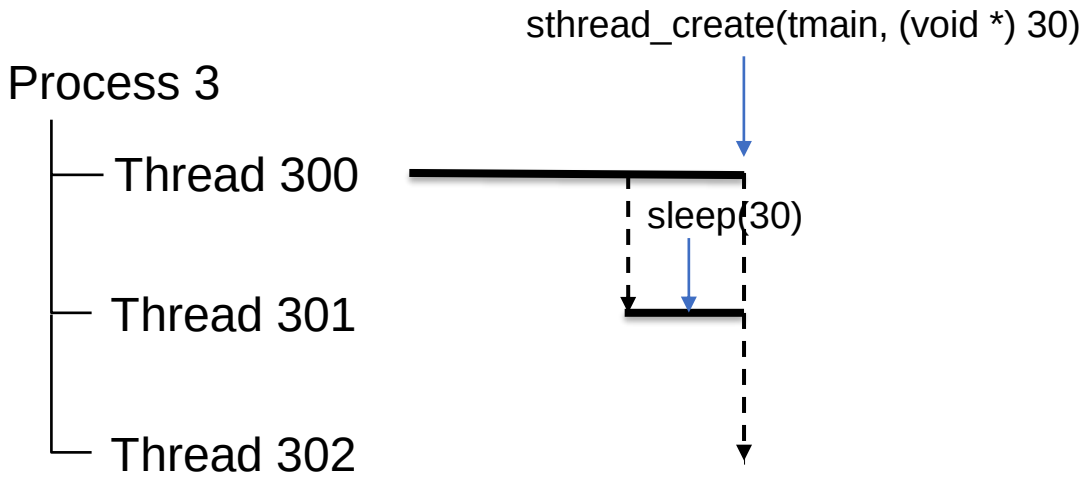
- Thread 300 calls sthread_create(tmain, 30)
  - Thread 300 creates new thread, thread 301
  - Thread 300 passes the value 30 to thread 301




```
void
main(int argc, char *argv[])
{
  sthread_create(tmain, (void *)30);
  sthread_create(tmain, (void *)10);
  sthread_create(tmain, (void *)20);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

- Thread 300 calls sthread_create(tmain, 10)
  - Thread 300 creates new thread, thread 302
  - Thread 300 passes the value 10 to thread 302
- Thread 301 sleeps

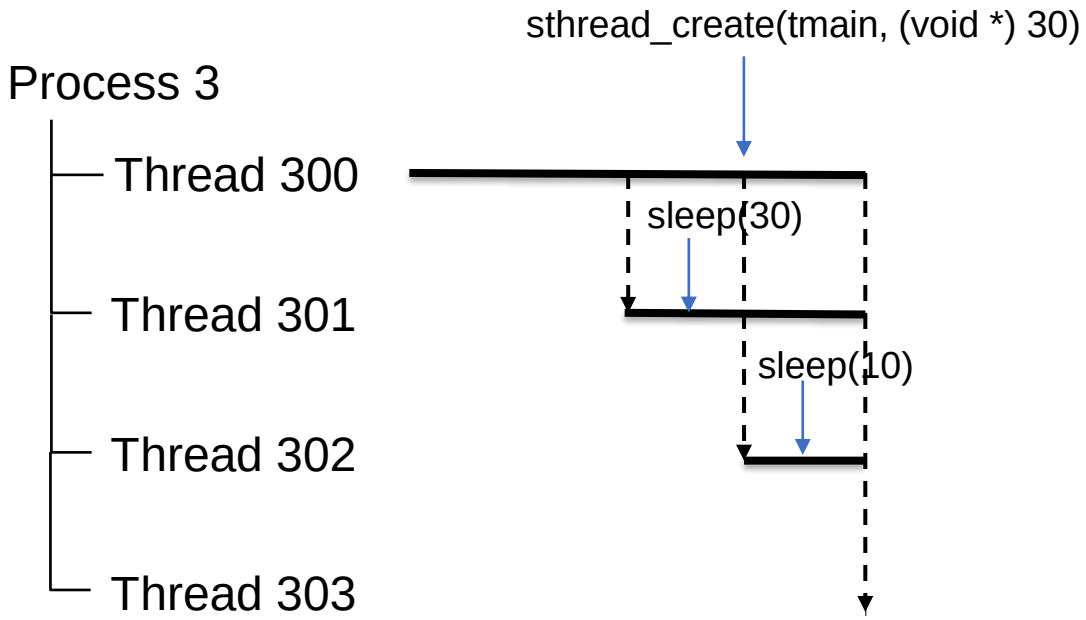



```c
void
main(int argc, char *argv[])
{
  sthread_create(tmain, (void *)30);
  sthread_create(tmain, (void *)10);
  sthread_create(tmain, (void *)20);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

```c
void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

- Thread 300 calls sthread_create(tmain, 20)
  - Thread 300 creates new thread, thread 303
  - Thread 300 passes the value 20 to thread 303
- Thread 302 sleeps

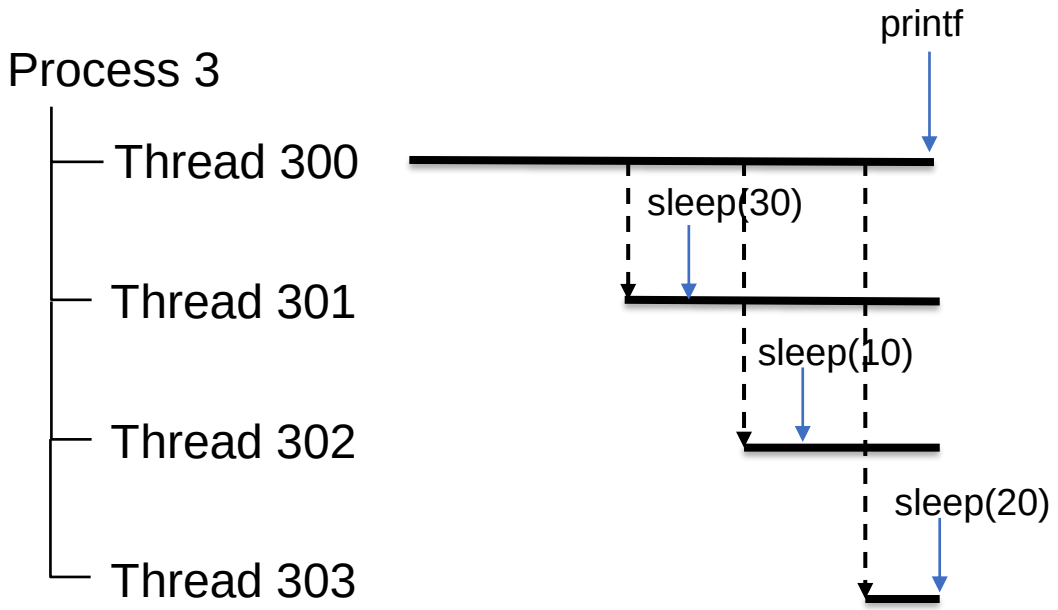



```c
void
main(int argc, char *argv[])
{
  sthread_create(tmain, (void *)30);
  sthread_create(tmain, (void *)10);
  sthread_create(tmain, (void *)20);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

```c
void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

- Thread 300 calls printf
  - Prints "Thread 300 is exiting"
- Thread 303 sleeps

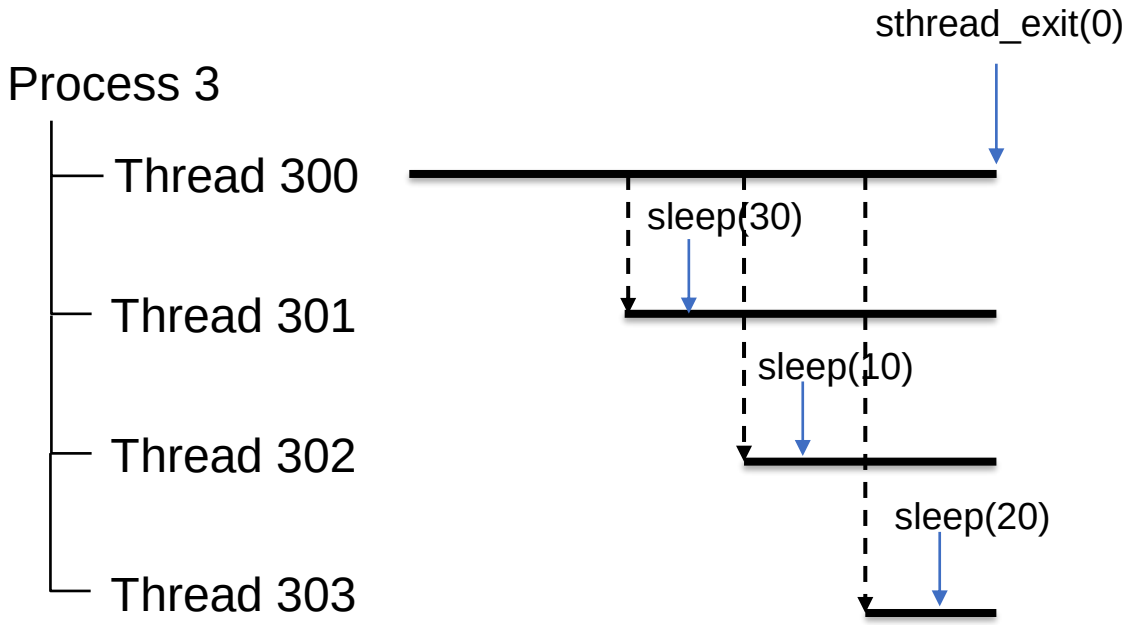



```c
void
main(int argc, char *argv[])
{
  sthread_create(tmain, (void *)30);
  sthread_create(tmain, (void *)10);
  sthread_create(tmain, (void *)20);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

```c
void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

- Thread 300 calls sthread_exit(0)
  - Thread 300 exits with the value 0

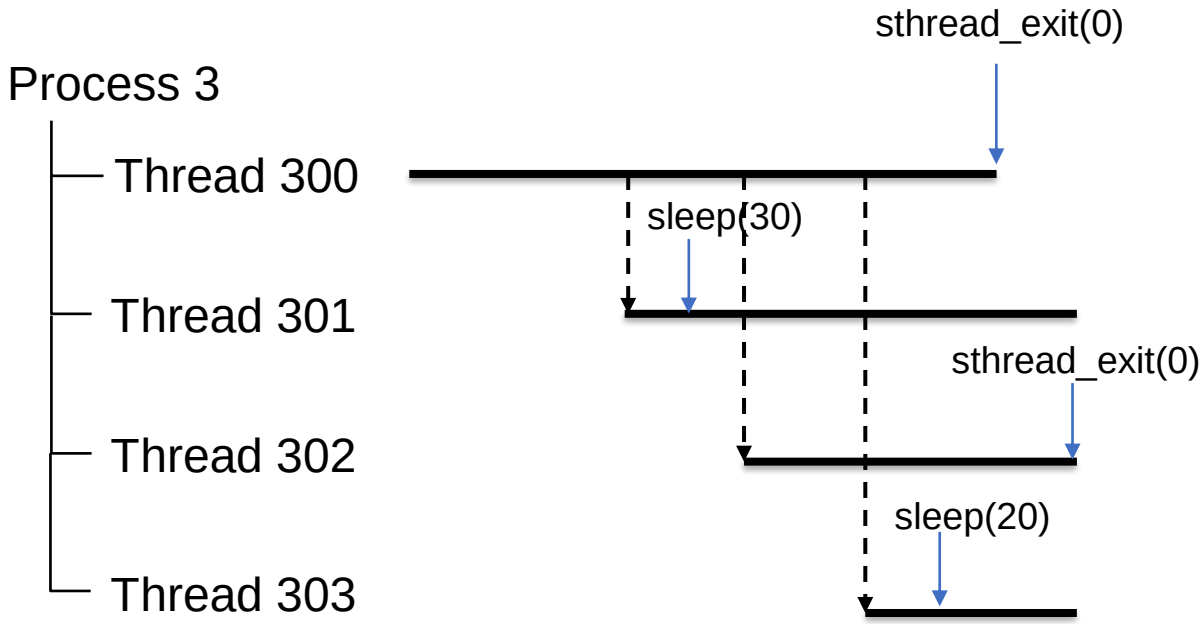



```c
void
main(int argc, char *argv[])
{
  sthread_create(tmain, (void *)30);
  sthread_create(tmain, (void *)10);
  sthread_create(tmain, (void *)20);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

```c
void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

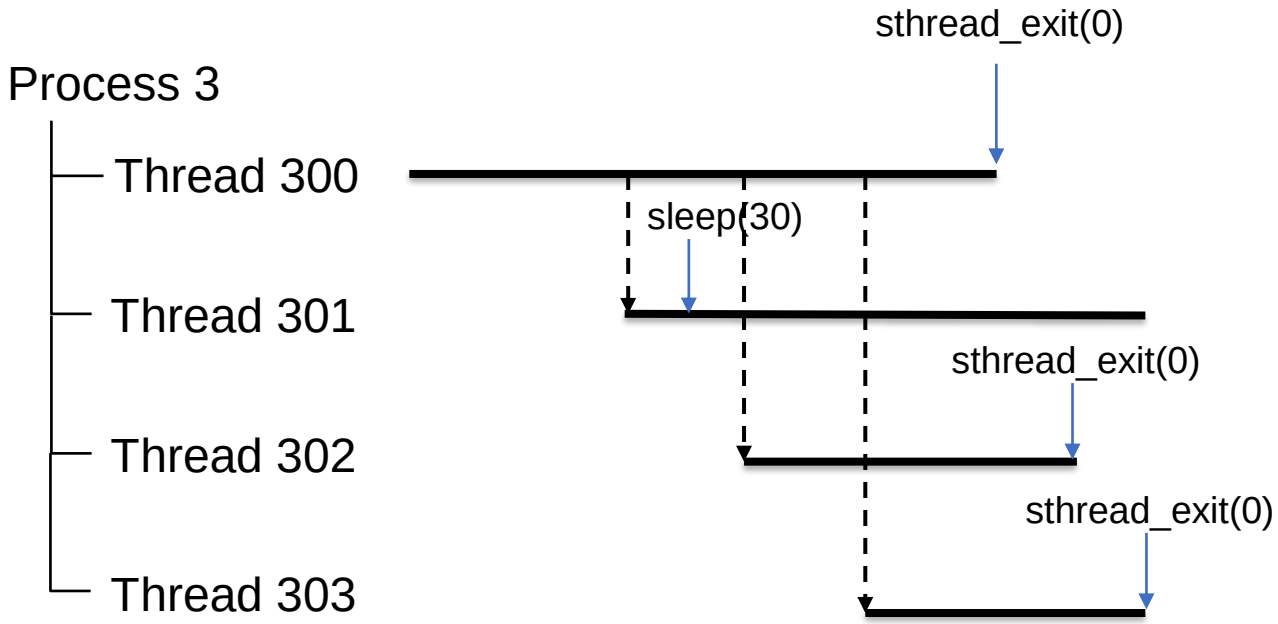
- Thread 302 wakes up and calls printf
  - Prints "Thread 302 is exiting"
- Thread 302 calls sthread_exit(0)




```c
void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

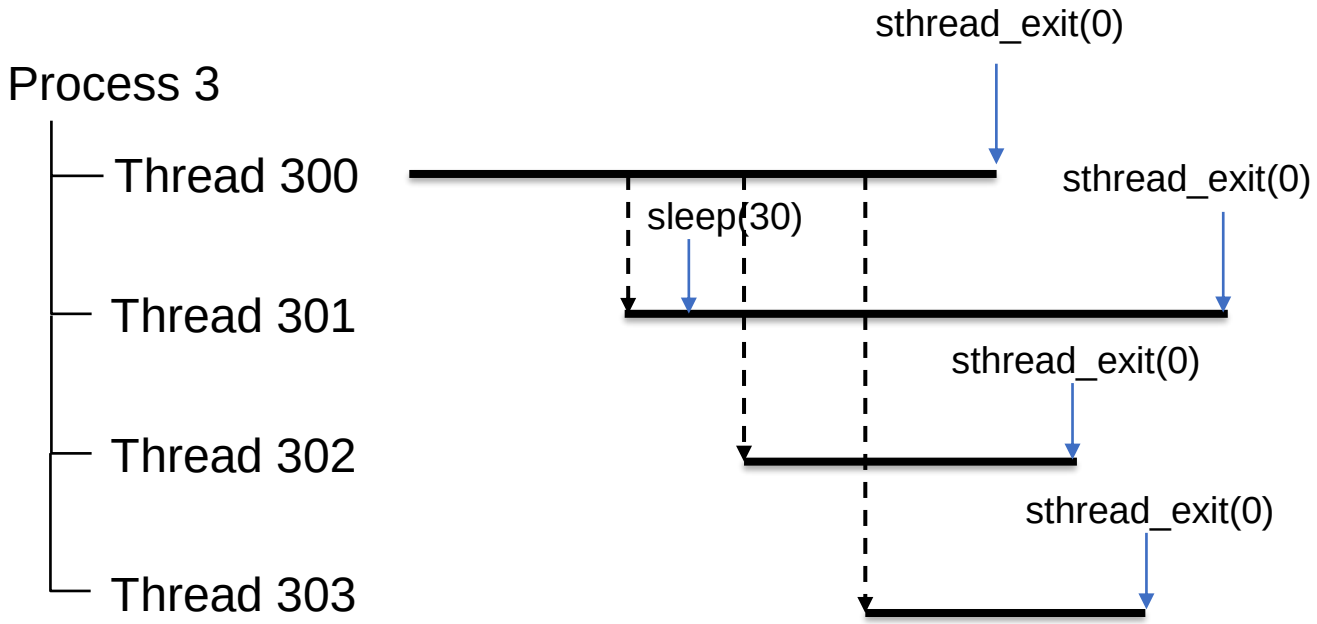
- Thread 303 wakes up and prints
  - Prints "Thread 303 is exiting"
- Thread 303 calls sthread_exit(0);




```c
void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 3

- Thread 301 wakes up and prints
  - Prints "Thread 301 is exiting"
- Thread 301 calls sthread_exit(0);
  - Thread 301 is the last thread in a process, so process is also terminated




```c
void tmain(void *arg)
{
  int t = (int)(long) arg;

  sleep(t);
  printf("Thread %d is exiting\n", sthread_self());
  sthread_exit(0);
}
```

Sample Output 4

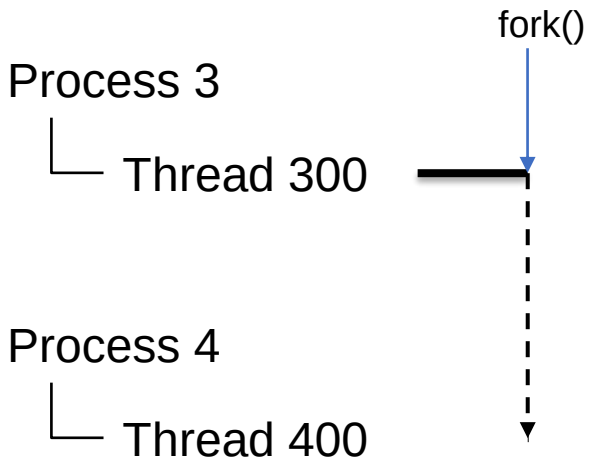
- user/thread4.c

- There will be two processes, process 3 and process 4, whose pids are 3 and 4, respectively

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"


void tmain(void *arg)
{
  char *args[] = {"ls", "/", 0};

  exec("ls", args);
}

void
main(int argc, char *argv[])
{
  int pid;
  int ret = 999;

  if ((pid = fork()) == 0)
  {
    sthread_create(tmain, 0);
    while (1);
  }
  wait(&ret);
  printf("ret = %d\n", ret);
}
```

Sample Output 4

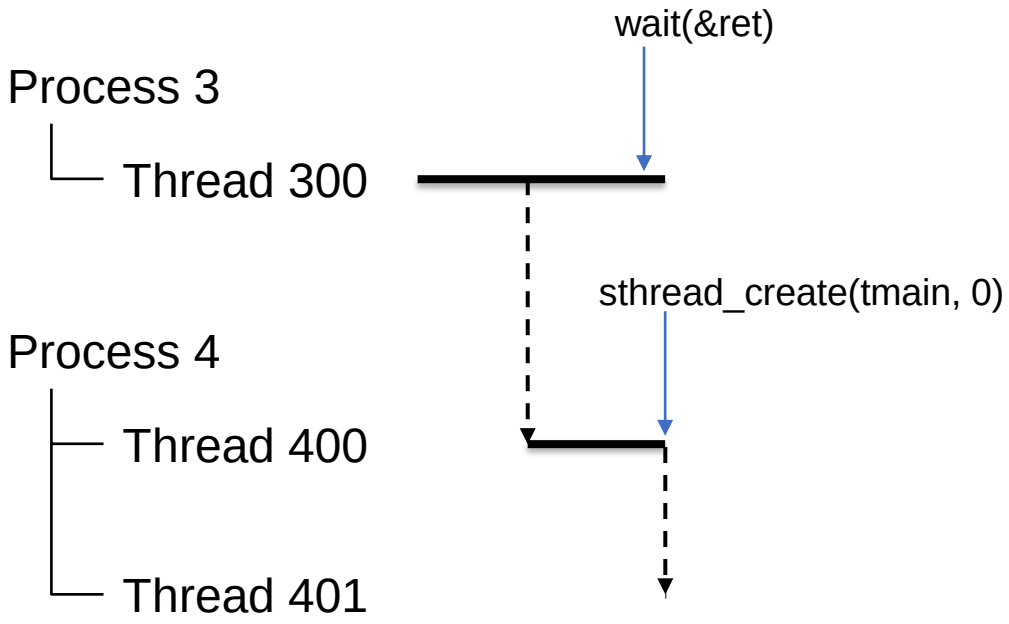
- Thread 300 calls fork()
  - New process, process 4 is created
  - Process 4's default thread, thread 400 is also created.




```c
void
main(int argc, char *argv[])
{
  int pid;
  int ret = 999;

  if ((pid = fork()) == 0)
  {
    sthread_create(tmain, 0);
    while (1);
  }
  wait(&ret);
  printf("ret = %d\n", ret);
}
```

Sample Output 4

- Thread 300 calls wait(&ret)
- Thread 400 calls sthread_create(tmain, 0)
  - Thread 400 creates new thread, thread 401 and passes the value 0



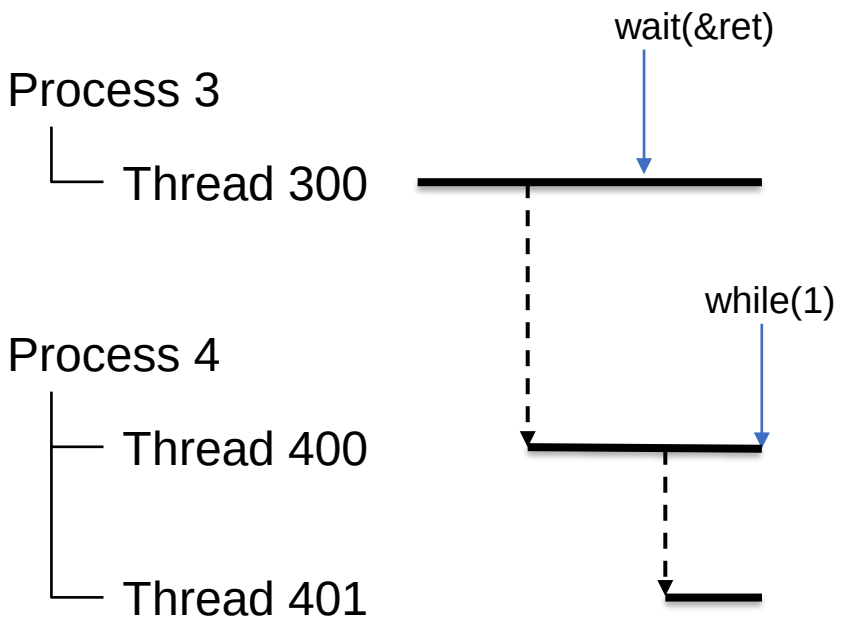
Thread 400

Thread 300

```c
void
main(int argc, char *argv[])
{
  int pid;
  int ret = 999;

  if ((pid = fork()) == 0)
  {
    sthread_create(tmain, 0);
    while (1);
  }
  wait(&ret);
  printf("ret = %d\n", ret);
}
```

Sample Output 4

- Thread 400 runs an infinite loop

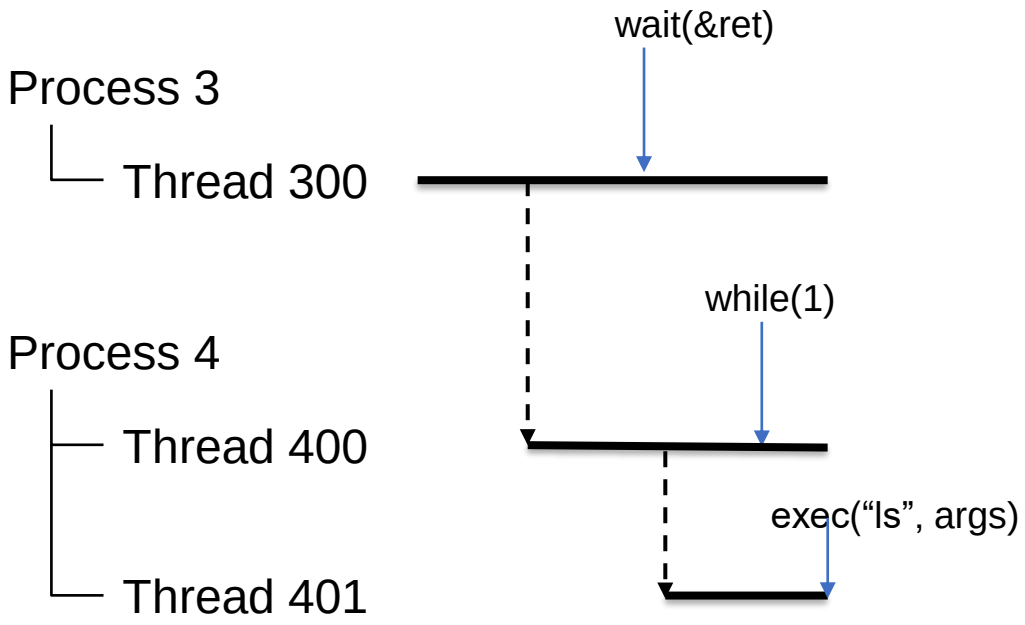



```c
void
main(int argc, char *argv[])
{
  int pid;
  int ret = 999;

  if ((pid = fork()) == 0)
  {
    sthread_create(tmain, 0);
    while (1);
  }
  wait(&ret);
  printf("ret = %d\n", ret);
}
```

```c
void tmain(void *arg)
{
  char *args[] = {"ls", "/", 0};

  exec("ls", args);
}
```

Sample Output 4

- Thread 401 calls exec("ls", args)
  - Thread 400 is terminated
  - Thread 401 runs the ls / command

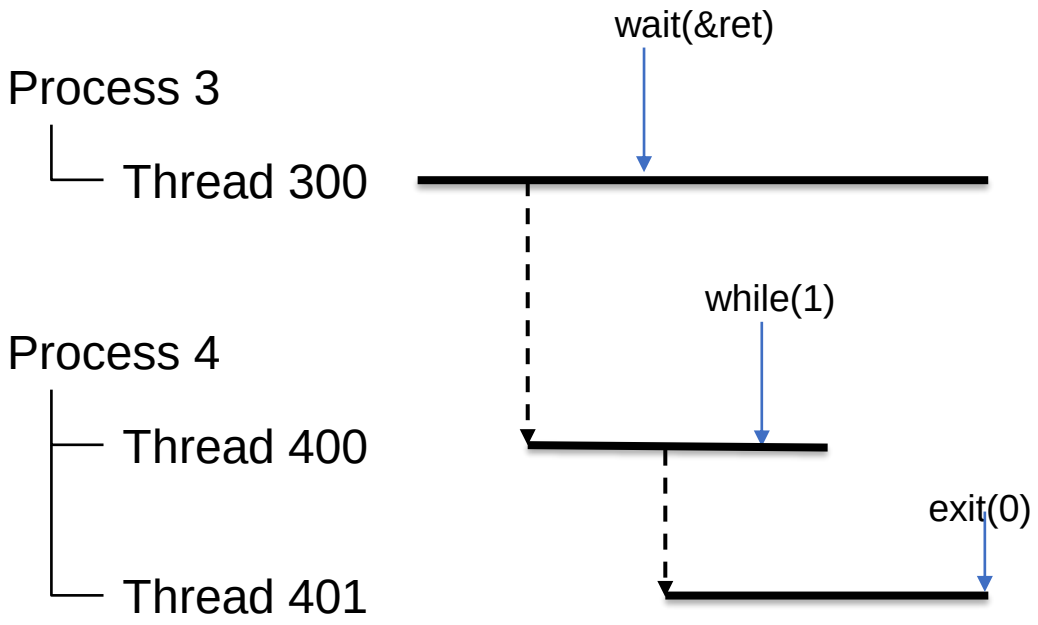



```c
void
main(int argc, char *argv[])
{
  int pid;
  int ret = 999;

  if ((pid = fork()) == 0)
  {
    sthread_create(tmain, 0);
    while (1);
  }
  wait(&ret);
  printf("ret = %d\n", ret);
}
```

```c
void tmain(void *arg)
{
  char *args[] = {"ls", "/", 0};

  exec("ls", args);
}
```

Sample Output 4

- Thread 401 executes ls, and calls exit(0)
  - Process 4 terminates
  - Process 3 receives exit status of process 4 in the ret variable

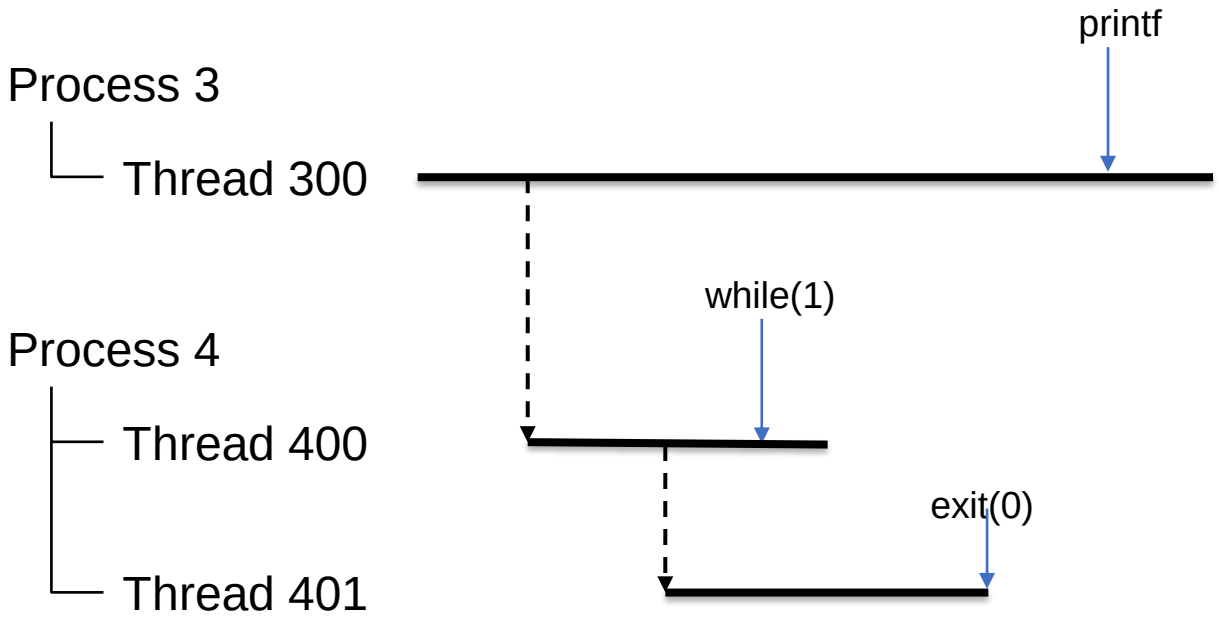



```c
int
main(int argc, char *argv[])
{
  int i;

  if(argc < 2){
    ls(".");
    exit(0);
  }
  for(i=1; i<argc; i++)
    ls(argv[i]);
  exit(0);
}
```

user/ls.c

Sample Output 4

- Thread 300 calls printf
  - Prints "ret = 0"
- Process 3 is terminated




```c
void
main(int argc, char *argv[])
{
  int pid;
  int ret = 999;

  if ((pid = fork()) == 0)
  {
    sthread_create(tmain, 0);
    while (1);
  }
  wait(&ret);
  printf("ret = %d\n", ret);
}
```

Thank you!

- Don't forget to read the detailed description
  - https://github.com/snu-csl/os-pa5
- Since this is the last assignment, you may use all the remaining slip days
  - You can use up to 3 slip days during this semester
- The weights for pa1 to pa5 are 1%, 2%, 7%, 15%, and 15%, respectively.
- Any questions?