Operating Systems

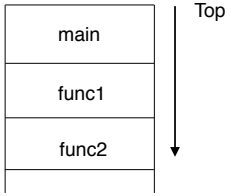
MyThreads

User-level Threads

- Are used to allow programs to switch between multiple threads of execution without involving the operating system
- Use cooperative multitasking rather than preemptive multitasking
- Also consider co-routines
  - Subroutines are a special case that stop the calling routine until the called subroutine has completed (Knuth)

Jumping around in C

- It is often useful to quickly return to the top of a set of called functions when an error occurs
  - Imagine recursive parsing



- **setjmp**() saves stack context for non-local *goto*
  - Non-local as a real goto is function scoped
  - Don't return from the function that called setjmp() or the context will be invalid

setjmp/longjmp

- Requires a global *jmp_buf jmpbuffer;*
  - Since it must be referenced in other functions
- if (**setjmp**(jmpbuffer) != 0) printf("returned\n");
  - That is, setjmp returns 0 when it is first called
- **longjmp**(jmpbuffer, 1);
  - Causes the setjmp to return a 2nd time, returning the value of the 2nd argument to longjmp

Jumping back

- What about the automatic variables on the stack? Variables in the registers?
  - It depends. The standards say it is "indeterminate"
  - Most implementations don't try to roll back
- The **volatile** type qualifier in ANSI C tells the compiler to suppress optimizations that might remove apparently redundant memory accesses
  - volatile int some_int;

Think about stacks and variables

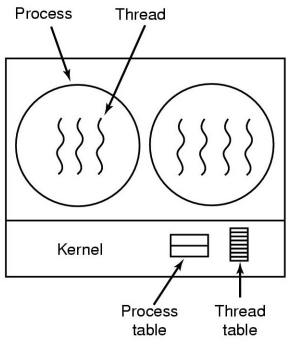
- If you allocate an automatic variable on the stack, that variable can only be used in that stack frame or "below"

```
int *
open_file(void) {
   int fd;

   fd = open(. . .

   return(&fd);
}
```

Implementing User-Level Threads

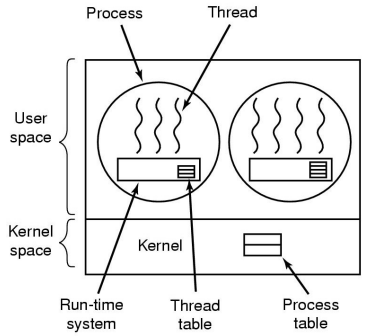
- Requires a user-space (non system) mechanism to save and restore register sets
- How can this be done with what we've talked about?

Implementing Threads in the Kernel



A threads package managed by the kernel

Implementing Threads in User Space

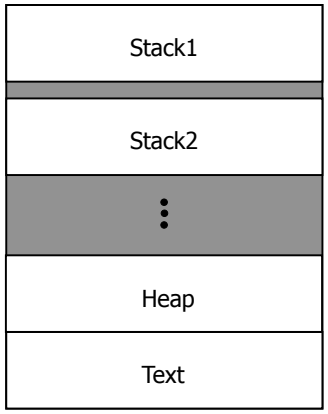


A user-level threads package

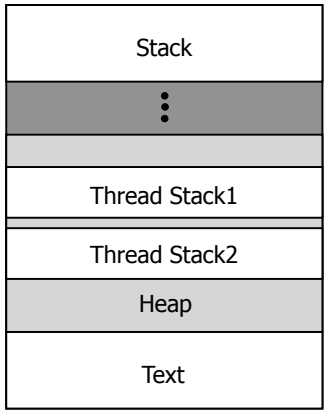
One mechanism for User-Level Threads

- Stack management
- PC management
- Recall the *jmpbuffer*
  - It stores the PC, SP, BP or FP (among others)

Stacks for Kernel Threads



Stacks for User Level Threads



One thread approach

- Make a jmpbuffer
- Allocate a "stack" in the heap
- Modify jmpbuffer to point to your new stack
- Hold on to your hat…

Another way

- System V and later POSIX defined a more general way to manage context

```
typedef struct ucontext {
        struct ucontext *uc_link;
        sigset_t         uc_sigmask;
        stack_t          uc_stack;
             uc_stack.ss_sp…
             uc_stack.ss_size…
        mcontext_t       uc_mcontext;
        ...
} ucontext_t;
```

Context functions

- `int setcontext(const ucontext_t *ucp)`
  - Transfers control to the context in ucp
- `int getcontext(ucontext_t *ucp)`
  - Save the current context into ucp
- `void makecontext(ucontext_t *ucp, void *func(), int argc, ...)`
  - Create an alternate context with entry point func
- `int swapcontext(ucontext_t *oucp, ucontext_t *ucp)`
  - Transfer control to ucp and save current context into oucp

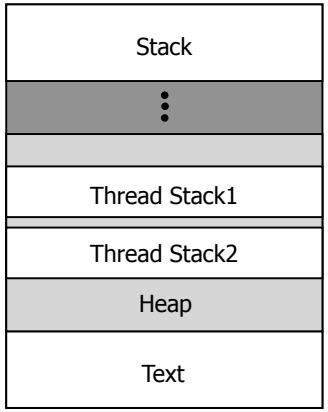
My Threads

- **`void mt_init();`**
- **`void *mt_create`**`(void (*func)(void *), void *arg);`
- **`void mt_join`**`(void *mt);`
- **`void mt_joinall();`**
- **`void mt_exit();`**
- **`void mt_sleep`**`(int seconds);`
- **`void mt_yield();`**
- **`void *mt_self();`**
- **`void mt_kill`**`(void *mt);`

My Threads Semaphores

- `typedef void *mt_sem;`
- **`mt_sem mt_sem_create`**`(int initial_value);`
- **`void mt_sem_up`**`(my_sem sem);`
- **`void mt_sem_down`**`(my_sem sem);`
- **`void mt_sem_destroy`**`(my_sem sem);`
- **`int mt_sem_getval`**`(my_sem sem);`

Stacks for User Level Threads



User-level threads

- Any thread call should invoke the user-level thread scheduler
- The first call to mt_create creates thread state and a stack
- Put the newly-created thread on the run queue
- The scheduler does any necessary housekeeping and chooses a runnable thread
  - In the beginning there is only one
- That thread runs until another thread routine is called
  - Then the scheduler is reentered

setjmp/longjmp

- Declare a global *jmp_buf jmpbuffer;*
- if (**setjmp**(jmpbuffer) != 0) printf("returned\n");
  - That is, setjmp returns 0 when it is first called
- **longjmp**(jmpbuffer, 1);
  - Causes the setjmp to return a 2nd time, returning the value of the 2nd argument to longjmp
- Use **volatile** for automatic variables

User-level threads

- Thread create must
  - malloc a stack for the new thread
  - Put the addresses of that stack in the jmpbuf
    - __jmpbuf[JB_BP] = newstack;
    - __jmpbuf[JP_SP] = newstack-1024;
      - Leave enough space for local variables

Context notes

- uc_link – when a context finishes (the entry function returns), the system swaps to the context pointed to by uc_link
- This could be a context that can clean up the thread state
- Scheduler can be a function or a distinct context

Joinall semantics

- Joinall will wait for all runnable threads
- If threads are blocked on semaphores or in sleep, joinall will return