6 Lecture: Intro. to Concurrency

Outline:

Announcements
Once again: Processes
Lightweight Processes: Threads
Aside: Reentrancy
Introduction to Asgn2
    Nine little functions
    *Scheduling
    Demonstration: threading in action
    Review: context switch
    A thread's context: stack and registers
    *Stack structure: The C calling convention
How to do it
    Race condition, defined
Critical Sections
Mutual Exclusion
Busy Waiting: Software Only
From last time: Review of Busy Waiting
Peterson's Solution

6.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn2	LWP	Mon	Jan 26	23:59	
asgn3	dine	Wed	Feb 4	23:59	
lab03	problem set	Mon	Feb 9	23:59	
midterm	stuff	Wed	Feb 11		
lab04	scavenger hunt II	Wed	Feb 18	23:59	
asgn4	/dev/secret	Wed	Feb 25	23:59	
lab05	problem set	Mon	Mar 9	23:59	
asgn5	minget and minls	Wed	Mar 11	23:59	
asgn6	Yes, really	Fri	Mar 13	23:59	
final (sec01)		Fri	Mar 20	10:10	
final (sec03)		Fri	Mar 20	13:10	

  Use your own discretion with respect to timing/due dates.

- In spite of saying "no support", I posted instructions for building individual tests on the forum
- `~pn-cs453/demos/tryAsgn1 2>&1 | tee logfile`
- MEANINGFUL SYMBOLS (not: SIXTEEN)
- gradebook
- `mkfs.mfs` on 3.1.8 (Read the durned footnote)

6.2 Once again: Processes

Recall (those of you who were awake) the process of a context switch:

- Old process suspended (timer?)
- Old process's registers saved (where?)
- Old process's memory—text,stack,data—saved in a **core image**
- Memory set up for new process (text,stack,data)
- Registers set up for new process
- New process "continued" as if nothing had happened.

  Imaging blinking and discovering that it was three hours later.

6.3 Lightweight Processes: Threads

Traditional Operating Systems only support processes as above.

Sometimes we don't need that kind of isolation.

Other times we don't **want** that kind of isolation.

Sometimes it's necessary (or at least desirable) to have multiple independent threads of control in a single address space: filesystem cache, web browser, etc.

Types of threads:

user-level all at the user level (advantages/disadventages)

POSIX P-threads, Mach C-Threads.

- lightweight (very)
- requires no kernel support. (if a thread blocks, the whole process blocks.)
- has no kernel support. (relies on good behavior—the reason we have a kernel in the first place)

kernel-level With kernel support level (advantages/disadventages)

- More robust
- More expensive (kernel must be aware of threads to schedule them, resulting in a "real" context switch.)

Unforunately, threads changes the programming model:

- Break the model of **sequential process**. no more illusion of sequentiality.
- Hard to retrofit: hidden assumptions (jet engine on a car?)
- Semantic questions:
  - What about `errno`?
  - Signal delivery? alarms?
  - fork(): does the child have all running threads?

  - fork(): what about blocked threads? (couldn't happen w/o threads)
  - Stack management: how to detect stack overflow and fix it.

- issues.
  - non-preemptive
  - preemptive

6.4 Aside: Reentrancy

Reentrant code is code that written such that a single copy in memory can be shared between many applications at once. That is, a reentrant function is one where it is possible to safely have more than one activation at the same time.

What that means:

- no self-modifying code
- no static variables (except those that actually pertain to truly global state).

Examples:

`strcat()` is reentrant.
`strtok()` is not.

6.5 Introduction to Asgn2

6.5.1 Nine little functions

`lwp_create(function,argument)`	create a new LWP
`lwp_start(void)`	start the LWP system
`lwp_yield(void)`	yield the CPU to another LWP
`lwp_exit(int)`	terminate the calling LWP
`lwp_wait(int *)`	wait for a thread to terminate
`lwp_gettid(void)`	return thread ID of the calling LWP
`tid2thread(tid)`	map a thread ID to a context
`lwp_set_scheduler(scheduler)`	install a new scheduling function
`lwp_get_scheduler(void)`	find out what the current scheduler is

Scheduling

The lwp scheduler is a structure that holds pointers to five functions. These are:

void init(void) This is to be called before any threads are admitted to the scheduler. It's to allow the scheduler to set up. This one is allowed, to be NULL, so don't call it if it is.

void shutdown(void) This is to be called when the lwp library is done with a scheduler to allow it to clean up. This, too, is allowed, to be NULL, so don't call it if it is.

void admit(thread new) Add the passed context to the scheduler's scheduling pool.

void remove(thread victim) Remove the passed context from the scheduler's scheduling pool and from the global thread list.

thread next(void) Return the next thread to be run or `NO_THREAD` if there isn't one.

Changing schedulers will involve initializing the new one, pulling out all the threads from the old one (using `next()` and `remove()`) and admitting them to the new one (with `admit()`), then shutting down the old scheduler.

6.5.2 Demonstration: threading in action

- nums
- randomsnakes
- hungrysnakes
- nums with different scheduling algorithms

  `-z` choose the process in slot 0
- hungrysnakes with different scheduling algorithms

  `-z` choose the process in slot 0

  `-h` choose the process that has eaten the most

  `-l` choose the process that has eaten the least

6.5.3 Review: context switch

- Old process suspended (timer?)
- Old process's registers saved (where?)
- Old process's memory—text, stack, data—saved in a **core image**
- Memory set up for new process (text,stack,data)
- Registers set up for new process
- New process "continued" as if nothing had happened.

6.5.4 A thread's context: stack and registers

Registers The register structure of the 64-bit Intel x86_64 is shown in Table 1.

rax	General Purpose A	r8	General Purpose 8
rbx	General Purpose B	r9	General Purpose 9
rcx	General Purpose C	r10	General Purpose 10
rdx	General Purpose D	r11	General Purpose 11
rsi	Source Index	r12	General Purpose 12
rdi	Destination Index	r13	General Purpose 13
rbp	Base Pointer	r14	General Purpose 14
rsp	Stack Pointer	r15	General Purpose 15

Table 1: Integer registers of the x86 CPU

Stack structure: The C calling convention

The extra registers available to the x86_64 allow it to pass some parameters in registers. This makes the overall calling convention a little more complicated, but, in practice, it will be easier for your program since you won't be passing enough parameters to push you out of the registers onto the stack.

The steps of the convention are as follows (illustrated in Figures 11a–f):

a. **Before the call** Caller places the first six integer arguments into registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9. If there are more, they are pushed onto the stack in reverse order. This is shown in the figure, but you won't encounter more in this assignment.

b. **After the call** The `call` instruction has pushed the return address onto the stack.

c. **Before the function body** If the function has more parameters and local varaibles than will fit into the registers it will execute the following two instructions to set up its frame:

```
pushq %rbp
movq %rsp,%rbp
```

Then, it adjusts the stack pointer to leave room for any locals it may need.

Your functions probably will not need to set up a frame.

d. **Before the return** If the function has set up a call frame in the previous step it needs to clean up after itself. To do this, before returning it executes a `leave` instruction. This instruction is equivalent to:

```
movq %rbp,%rsp
popq %rbp
```

The effect is to rewind the stack back to its state right after the call.

If the function did not set up a frame, its `%rsp` register is still pointing to the return address.

e. **After the return** After the return, the Return address has been popped off the stack, leaving it looking just like it did before the call.

Remember, the `ret` instruction, while called "return", really means "pop the top of the stack into the program counter."

f. **After the cleanup** Finally, the caller pops off any parameters on the stack and leaves the stack is just like it was before.

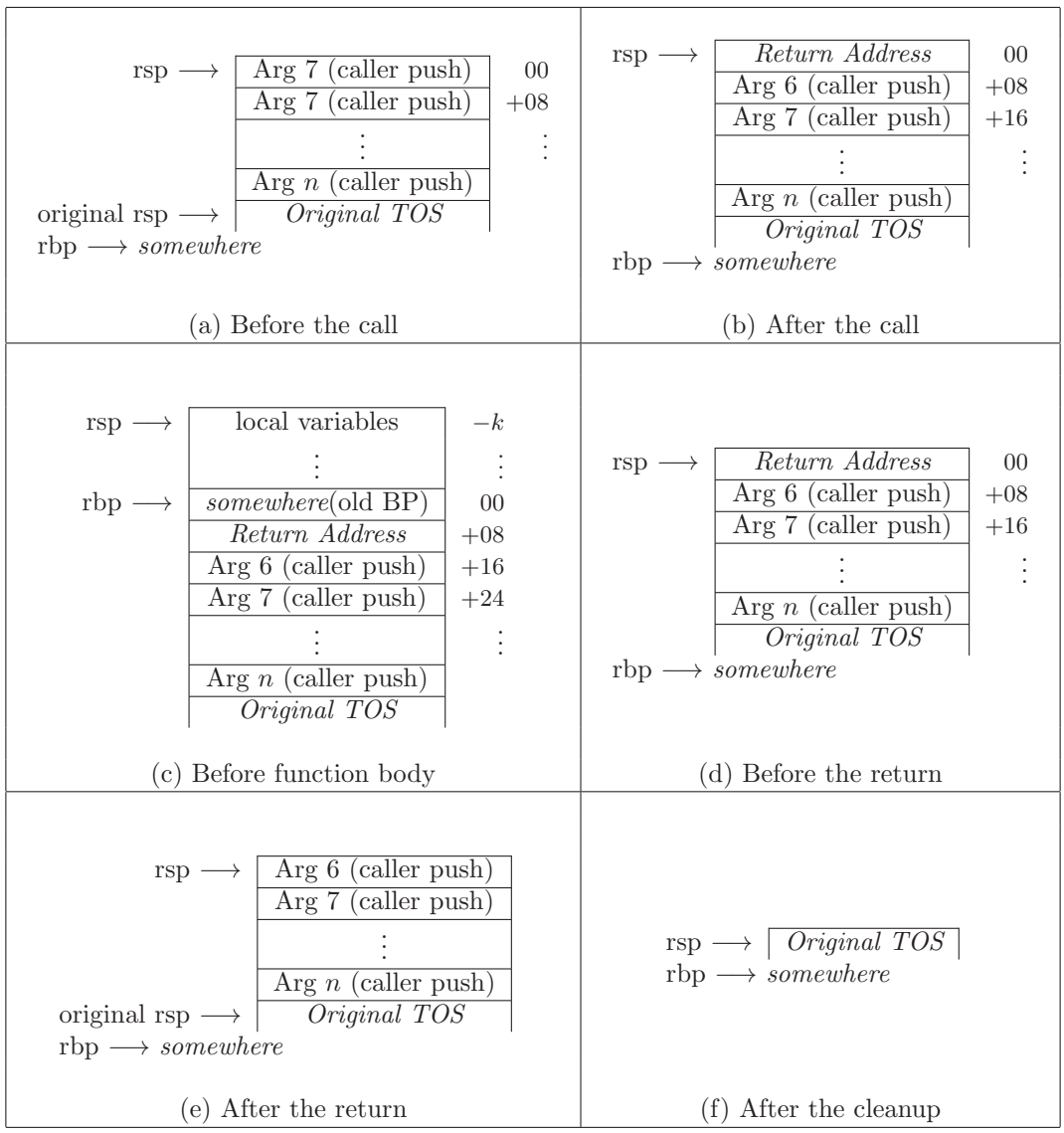
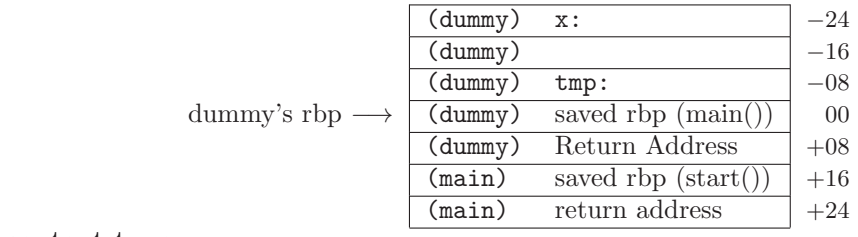


Figure 11: Stack development (Remember that the real stack is upside-down)

Example (real gcc-generated code):

```
dummy:  pushq %rbp                                  long dummy(long x) {
        movq %rsp, %rbp                                 long tmp;
        movq %rdi, -24(%rbp)                            tmp = x;
        movq -24(%rbp), %rax                            return tmp;
        movq %rax, -8(%rbp)                         }
        leave
        ret
```

```
main:   pushq %rbp          # set up frame          int main(){
        movq %rsp, %rbp                                 dummy(5);
→       movl $5, %edi       # load argument         }
        call dummy          # do call
        leaveq
        retq
```



The context type:

```
typedef struct threadinfo_st *thread;
typedef struct threadinfo_st {
  tid_t        tid;          /* lightweight process id  */
  unsigned long *stack;      /* Base of allocated stack */
  size_t       stacksize;    /* Size of allocated stack */
  rfile        state;        /* saved registers         */
  thread       lib_one;      /* Two pointers reserved   */
  thread       lib_two;      /* for use by the library  */
  thread       sched_one;    /* Two more for            */
  thread       sched_two;    /* schedulers to use       */
} context;
```

The scheduler type:

```
/* Tuple that describes a scheduler */
typedef struct scheduler {
  void   (*init)(void);          /* initialize any structures     */
  void   (*shutdown)(void);      /* tear down any structures      */
  void   (*admit)(thread new);   /* add a thread to the pool      */
  void   (*remove)(thread victim); /* remove a thread from the pool */
  thread (*next)();              /* select a thread to schedule   */
} *scheduler;
```

The lwp functions:

`lwp_create(function,argument,stacksize)`	create a new LWP
`lwp_gettid()`	return thread ID of thecalling LWP
`lwp_exit()`	terminates the calling LWP
`lwp_yield()`	yield the CPU to another LWP
`lwp_start()`	start the LWP system
`lwp_stop()`	stop the LWP system
`lwp_set_scheduler(scheduler)`	install a new scheduling function
`lwp_get_scheduler(scheduler)`	find out what the current scheduler is
`tid2thread(tid)`	map a thread ID to a context

Tools for stack manipulation:

Macros for stack manipulation: Remember, these **must** be called first and last.

GetSP(var)	Sets the given variable to the current value of the stack pointer.
SetSP(var)	Sets the stack pointer to the current value of the given variable.

`void swap_rfiles(rfile *old, rfile *new)`. This does two things:

1. if `old != NULL` it saves the current values of all 16 registers to the `struct registers` pointed to by `old`.
2. if `new != NULL` it loads the 16 register values contained in the `struct registers` pointed to by `new` into the registers.

General Notes:

- Remember to allocate enough space for your stacks
- Remember stacks are **upside down**
- Draw pictures!
- Don't even think of turning the optimizer on!
- "Remember" `gcc -S`
- "remember" function pointers

```
typedef void (*lwpfun)(void *); /* type for lwp function */
```

- be careful of locals when climbing from one stack to another
- what happens if your function returns? (Build a wraper function?)

6.6 How to do it

All (most of?) the magic is in creating the thread:

- Allocate the stack
- Figure out what you want the world to look like when the thread starts running

- Wind up the stack so that when yield() chooses the thread, loads its context with swap_rfiles() and **returns to it** that the world looks like that.
- The endgame of swap_rfiles() is:

```
leave
    movq %rbp, %rsp
    popq %rbp
ret
    popq %rip
```