Threads

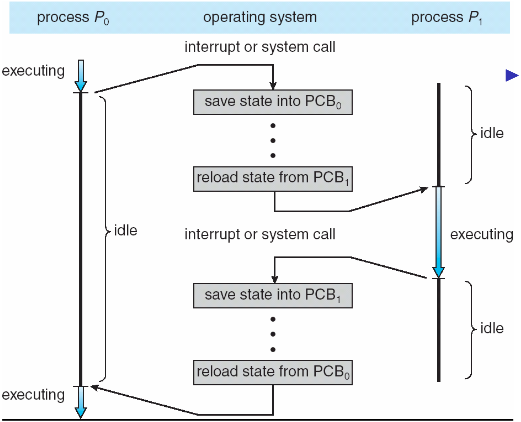
Operating System Design – M1 Info

Instructor: Vincent Danjean
Class Assistant: Florent Bouchez

October 5, 2015

Outline

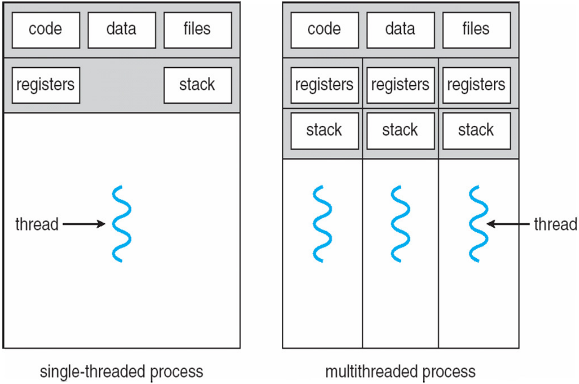
Remember Context Switches



- **Typical things include:**
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations

- **Non-negligible cost**
  - Save/restore floating point registers expensive
  - May require flushing TLB (memory translation hardware)
  - Usually causes more cache misses (switch working sets)
- **Sharing data/information between process may be painful**

Threads



- **A thread is a schedulable execution context**
  - Program counter, stack, registers, . . .
- **Simple programs use one thread per process**
- **But can also have multi-threaded programs**
  - Multiple threads running in same process's address space

Why threads?

- **Responsiveness**
  - Do not block the whole program when only a part of it should be blocked
  - Allows program to overlap I/O and computation (same benefit as OS running emacs & gcc simultaneously)
  - E.g., threaded web server services clients simultaneously:

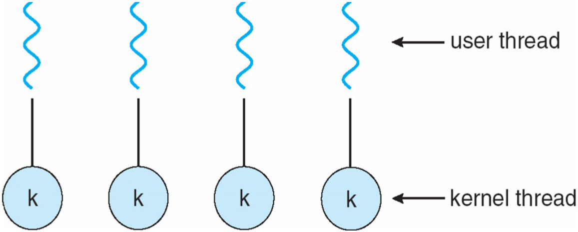
```
for (;;) {
  fd = accept_client ();
  thread_create (service_client, &fd);
}
```

- **Resource sharing**
  - Lighter-weight abstraction than processes (IPC, shmem)
  - All threads in one process share memory, file descriptors, etc
- **Economy**
  - Allocating memory, resources and context switching for process is costly
- **Scalability**
  - A single process can only use a single CPU at a time
  - Allows one process to use multiple CPUs or cores

Thread package API

- `tid thread_create (void (*fn) (void *), void *);`
  - Create a new thread, run `fn` with `arg`
- `void thread_exit ();`
  - Destroy current thread
- `void thread_join (tid thread);`
  - Wait for thread `thread` to exit
- **Plus lots of support for synchronization [next week]**
- **Can have preemptive or non-preemptive threads**
  - Preemptive causes more race conditions
  - Non-preemptive can't take advantage of multiple CPUs
  - Before prevalent SMPs, most kernels non-preemptive

Kernel threads

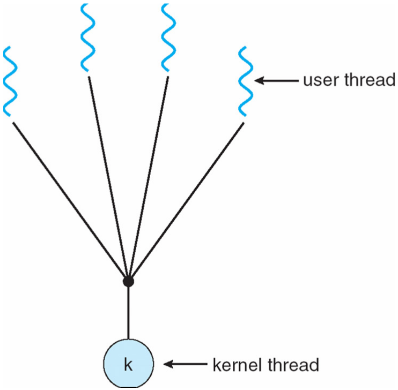


- **Can implement** `thread_create` **as a system call**
- **To add** `thread_create` **to an OS that doesn't have it:**
  - Start with process abstraction in kernel
  - `thread_create` like process creation with features stripped out
    - Keep same address space, file table, etc., in new process
    - `rfork`/`clone` syscalls actually allow individual control
    - LinuxThread have been implemented by hacking `clone` for a long time (threads appeared in the process table and were not optimally managed)
    - Now we have the Native POSIX Thread Library
- **Faster than a process, but still very heavy weight**

Limitations of kernel-level threads

- **Every thread operation must go through kernel**
  - create, exit, join, synchronize, or switch for any reason
  - On Athlon 3400+: syscall takes 359 cycles, fn call 6 cycles
  - Result: threads 10x-30x slower when implemented in kernel
- **One-size fits all thread implementation**
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don't need
- **General heavy-weight memory requirements**
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier-weight processes

User threads



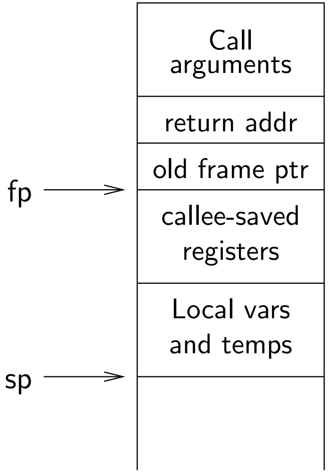
- **An alternative: implement in user-level library**
  - One kernel thread per process
  - `thread_create`, `thread_exit`, etc., just library functions

Implementing user-level threads

- **Allocate a new stack for each** `thread_create`
- **Keep a queue of runnable threads**
- **Replace networking system calls (**`read`**/**`write`**/etc.)**
  - If operation would block, switch and run different thread
- **Schedule periodic timer signal (**`setitimer`**)**
  - Switch to another thread on timer signals (preemption)
- **Multi-threaded web server example**
  - Thread calls `read` to get data from remote web browser
  - "Fake" user-level `read` make `read` syscall in non-blocking mode
  - No data? schedule another thread
  - On timer or when idle check which connections have new data
- **How to switch threads?**

Background: calling conventions

- **Registers divided into 2 groups**
  - Functions free to clobber *caller-saved* regs
    (%eax [return val], %edx, & %ecx on x86)
  - But must restore *callee-saved* ones to original value upon return
- **sp register always base of stack**
  - Frame pointer (*fp*) is old *sp*
- **Local variables stored in registers and on stack**
- **Function arguments go in callee-saved regs and on stack**



Background: procedure calls

save active caller registers
call foo → saves used callee registers
...do stuff...
restores callee registers
jumps back to pc
restore caller regs

- **Some state saved on stack**
  - Return address, caller-saved registers
- **Some state not saved**
  - Callee-saved regs, global variables, stack pointer

Threads vs. procedures

- **Threads may resume out of order:**
  - Cannot use LIFO stack to save state
  - General solution: one stack per thread
- **Threads switch less often**
- **Threads can be involuntarily interrupted:**
  - Synchronous: procedure call can use compiler to save state
  - Asynchronous: thread switch code saves all registers
- **More than one thread can run at a time**
  - Thread scheduling: What to run next and on which CPU?
  - Procedure call scheduling obvious: Run called procedure

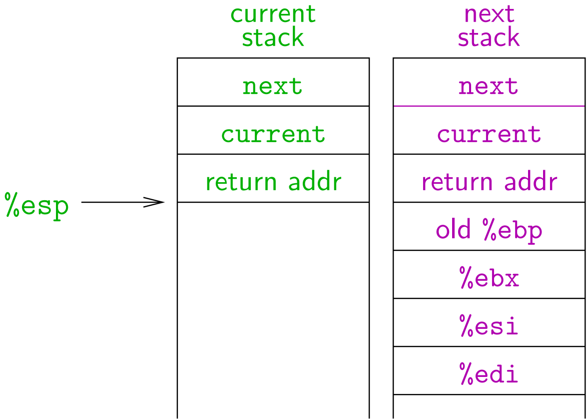
Example user threads implementation

- **Per-thread state in thread control block structure**

```
typedef struct tcb {
  unsigned long md_esp; /* Stack pointer of thread */
  char *t_stack;        /* Bottom of thread's stack */
  /* ... */
};
```

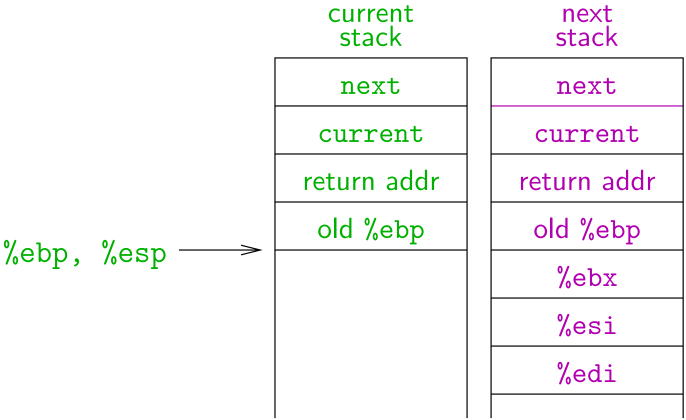
- **Machine-dependent thread-switch function:**
  - `void thread_md_switch (tcb *current, tcb *next);`
- **Machine-dependent thread initialization function:**
  - ```
    void thread_md_init (tcb *t,
            void (*fn) (void *), void *arg);
    ```
    ```

i386 thread_md_switch



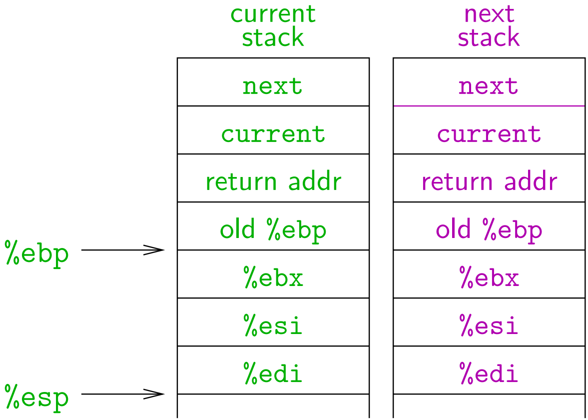
- **This is literally switch code from simple thread lib**
  - Nothing magic happens here when you can read assembly code

i386 thread_md_switch



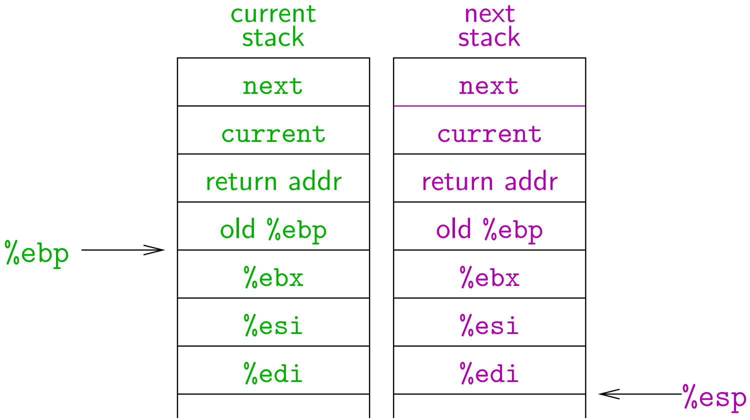
- **This is literally switch code from simple thread lib**
  - Nothing magic happens here when you can read assembly code

i386 thread_md_switch



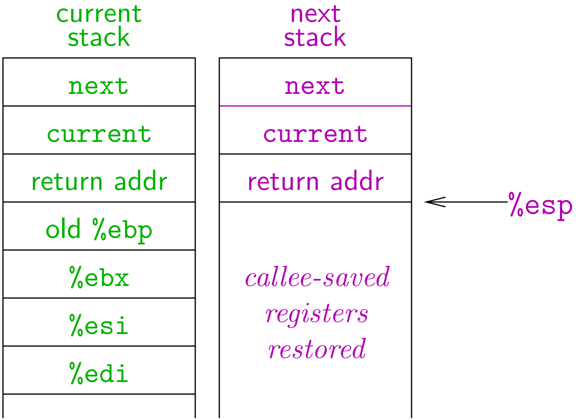
- **This is literally switch code from simple thread lib**
  - Nothing magic happens here when you can read assembly code

i386 thread_md_switch



- **This is literally switch code from simple thread lib**
  - Nothing magic happens here when you can read assembly code

i386 thread_md_switch

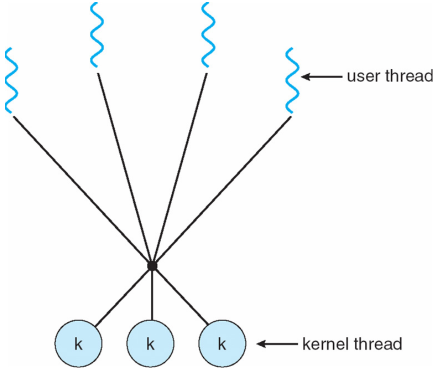


- **This is literally switch code from simple thread lib**
  - Nothing magic happens here when you can read assembly code

Limitations of user-level threads

- **Can't take advantage of multiple CPUs or cores**
- **A blocking system call blocks all threads**
  - Can replace `read` to handle network connections
  - But usually OSes don't let you do this for disk
  - So one uncached disk read blocks all threads
- **A page fault blocks all threads**
- **Possible deadlock if one thread blocks on another**
  - May block entire process and make no progress

User threads on kernel threads



- **User threads implemented on kernel threads**
  - Multiple kernel-level threads per process
  - `thread_create`, `thread_exit` still library functions as before
- **Sometimes called $n : m$ threading**
  - Have n user threads per m kernel threads
    (Simple user-level threads are $n : 1$, kernel threads $1 : 1$)

Limitations of $n : m$ threading

- **Many of same problems as $n : 1$ threads**
  - Blocked threads, deadlock, ...
- **Hard to keep same # ktrheads as available CPUs**
  - Kernel knows how many CPUs available
  - Kernel knows which kernel-level threads are blocked
  - But tries to hide these things from applications for transparency
  - So user-level thread scheduler might think a thread is running while underlying kernel thread is blocked
- **Kernel doesn't know relative importance of threads**
  - Might preempt kthread in which library holds important lock

fork and exec

- **What happens if one thread of a program calls `fork()`?**
  - Does the new process duplicate all threads ? Or i the new process single-threaded ?
  - Some UNIX systems have chose to have two versions of fork()
- **What happens if one thread of a program calls `exec()`?**
  - Generally, the new program replace the entire process, including all threads.

Cancellation

- **One may want to cancel a thread before it has completed**
  - When multiple threads concurrently search for a given data in a database
  - When you hit the stop button of your Web browser, all the threads in charge of loading the core of the page and the various images should be canceled
- **Asynchronous cancellation**
  - One thread **immediately** terminates the target thread.
  - Main issue: what if resources have been allocated and/or the target thread is in the midst of updating data shared with other threads ?
  - May lead to **incoherent state**
- **Deferred cancellation**
  - The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion
  - Such points are called **cancellation points**

Signal Handling

- **There are two types of signals**
  - Synchronous signals (SIGSEGV, SIGFPE), which are delivered to the process that generated the signal.
  - Asynchronous signals (SIGALARM, SIGPIPE, SIGSTOP,...) whose handler may be changed and that may sometimes be ignored.
- **Handling signals in single-threaded programs is straight-forward**
  - signals are always delivered to a process
- **In a multi-threaded program, who should receive the signal ?**
  1. Deliver the signal to the thread to which the signal applies (e.g., SIGSEGV)
  2. Deliver the signal to every thread in the process
  3. Deliver the signal to certain threads in the process
  4. Assign a specific thread to receive all signals for the process

  In many UNIX, the first thread which does not block the signal handles it.
- **POSIX threads have the** `pthread_kill(pthread_t tid, int signal)` **function**

Thread Pools

- **Web servers could create threads upon each request**
  - Although it is better than creating a process, creating thread is costly, especially regarding its corresponding service time
  - If there is no bound on the number of concurrently active threads, we could exhaust the OS resources (CPU, RAM) and trash the system
- **Thread Pool address these two issues**
  - Remember the slab allocator from the kernel ?
  - Create a number of threads at process startup and place them into a pool where they wait for work.
  - When a server receives a request, it awakens a thread from the pool if any available and waits otherwise.
  - When the thread has finished servicing the request, it returns to the pool, awaiting for more work.

Thread specific data

- **All threads share the data of the process**
- **In some circumstances, each thread may need to have its ow copy of certain data**
- **Most thread libraries provide some support for thread-specific data**
  - POSIX provides the following functions:

```
int pthread_setspecific(pthread_key_t key, const void *pointer);
void *pthread_getspecific(pthread_key_t key);
```

  - Each thread possesses a private memory block, the thread-specific data area (TSD)
  - This area is indexed by TSD keys and associates values of type void * to TSD keys.
  - TSD keys are common to all threads, but the value associated with a given TSD key can be different in each thread.

Lessons

- **Threads best implemented as a library**
  - But kernel threads not best interface on which to do this
- **Better kernel interfaces have been suggested**
  - See Scheduler Activations [Anderson et al.]
  - Maybe too complex to implement on existing OSes (some have added then removed such features)
- **Today shouldn't dissuade you from using threads**
  - Standard user or kernel threads are fine for most purposes
  - Use kernel threads if I/O concurrency main goal
  - Use $n : m$ threads for highly concurrent (e.g,. scientific applications) with many thread switches
- **. . . though the next two lectures may dissuade you**
  - Concurrency greatly increases the complexity of a program!
  - Leads to all kinds of nasty race conditions

Outline

Surprising Interleaving

```
int count = 0;

void loop(void *ignored) {
  int i ;
  for (i=0 ; i<10 ; i++)
     count++;
}

int main () {
  tid id = thread_create (loop, NULL);
  loop (); thread_join (id);
  printf("%d",count);
}
```

- **What is the output of this program ?**

Surprising Interleaving

```
int count = 0;

void loop(void *ignored) {
  int i ;
  for (i=0 ; i<10 ; i++)
     count++;
}

int main () {
  tid id = thread_create (loop, NULL);
  loop (); thread_join (id);
  printf("%d",count);
}
```

- **What is the output of this program ?**
- **Any value between 2 and 20.**
  - Remember that `count++` may be transformed into :

    $\texttt{reg1} \leftarrow \texttt{count}$

    $\texttt{reg1} \leftarrow \texttt{reg1}+1$

    $\texttt{count} \leftarrow \texttt{reg1}$

Program A

```
int flag1 = 0, flag2 = 0;

void p1 (void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main () {
  tid id = thread_create (p1, NULL);
  p2 (); thread_join (id);
}
```

- **Can both critical sections run?**

Program B

```
int data = 0, ready = 0;

void p1 (void *ignored) {
  data = 2000;
  ready = 1;
}

void p2 (void *ignored) {
  while (!ready)
    ;
  use (data);
}

int main () { ... }
```

- **Can** `use` **be called with value 0?**

Program C

```
int a = 0, b = 0;

void p1 (void *ignored) { a = 1; }

void p2 (void *ignored) {
  if (a == 1)
    b = 1;
}

void p3 (void *ignored) {
  if (b == 1)
    use (a);
}

int main () { ... }
```

- **Can `use` be called with value 0?**

Correct answers

- **Program A: I don't know**
- **Program B: I don't know**
- **Program C: I don't know**
- **Why?**
  - It depends on your hardware and compiler
  - If it provides **sequential consistency**, then answers all No
  - But not all hardware provides sequential consistency
- **Note: Examples and other frame content from [Adve & Gharachorloo]**

Sequential Consistency

- *Sequential consistency*: The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program. [Lamport]
- Boils down to two requirements:
  1. Maintaining *program order* on individual processors
  2. Ensuring *write atomicity*

- **Without SC, multiple CPUs can be "worse" than preemptive threads**
  - May see results that cannot occur with any interleaving on 1 CPU
- **Why doesn't all hardware support sequential consistency?**

SC thwarts hardware optimizations

- **Complicates write buffers**
  - E.g., read flag$n$ before flag$(2 - n)$ written through in Program A
- **Can't re-order overlapping write operations**
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line
- **Complicates non-blocking reads**
  - E.g., speculatively prefetch `data` in Program B
- **Makes cache coherence more expensive**
  - Must delay write completion until invalidation/update (Program B)
  - Can't allow overlapping updates if no globally visible order (Program C)

SC thwarts compiler optimizations

- **Code motion**
- **Caching value in register**
  - E.g., `ready` flag in Program B
- **Common subexpression elimination**
  - Could cause memory location to be read fewer times
- **Loop blocking**
  - Re-arrange loops for better cache performance
- **Software pipelining**
  - Move instructions across iterations of a loop to overlap instruction latency with branch cost

x86 consistency

- **x86 supports multiple consistency/caching models**
  - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory (e.g., frame buffer)
  - Page Attribute Table (PAT) allows control for each 4K page
- **Choices include:**
  - **WB**: Write-back caching (the default)
  - **WT**: Write-through caching (all writes go to memory)
  - **UC**: Uncacheable (for device memory)
  - **WC**: Write-combining – weak consistency & no caching
- **Some instructions have weaker consistency**
  - String instructions
  - Special "non-temporal" instructions that bypass cache

x86 atomicity

- `lock` **prefix makes a memory instruction atomic**
  - Usually locks bus for duration of instruction (expensive!)
  - Can avoid locking if memory already exclusively cached
  - All lock instructions totally ordered
  - Other memory instructions cannot be re-ordered w. locked ones
- `xchg` **instruction is always locked (even w/o prefix)**
- **Special fence instructions can prevent re-ordering**
  - `LFENCE` – can't be reordered w. reads (or later writes)
  - `SFENCE` – can't be reordered w. writes
  - `MFENCE` – can't be reordered w. reads or writes

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe?

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe?
- **Not atomic on multiprocessor!**
  - Will experience exact same race condition
  - Can potentially make atomic with `lock` prefix
  - But `lock` very expensive
  - Compiler won't generate it, assumes you don't want penalty
- **Need solution to critical section problem**
  - Place `count++` and `count--` in critical section
  - Protect critical sections from concurrent execution

Problem Statement

- n processes all competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

```
do {
  entry section()
    critical section
  exit section()
  remainder section
} while (1);
```

Desired Properties

- **Mutual Exclusion**
  - Only one thread can be in critical section at a time
- **Progress**
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- **Bounded waiting**
  - Once a thread T starts trying to enter the critical section, there is a bound on the number of times other threads get in
- **Note progress vs. bounded waiting**
  - If no thread can enter C.S., don't have progress
  - If thread A waiting to enter C.S. while B repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting