CS 241 Honors
Kernel

Aneesh Durg

University of Illinois Urbana–Champaign

October 11, 2017

What to go over

- What does the (linux) kernel actually do?
- System Calls
- Linking and Loading
- Processes and Threads
- Completely Fair Scheduler

Motivation

```c
#include <pthread.h>

void* hello_thread(void *payload){
    write(1, "Hello world!", 12);
    return NULL;
}

int main(){
    pthread_create(NULL, hello_thread,
        NULL, NULL);
    pthread_exit();
}
```

Motivation

```c
#include <pthread.h> //Dynamic Linked Library

void* hello_thread(void *payload){ //Pthread
    write(1, "Hello world!", 12); //Sys Call
    return NULL;
}

int main(){ //Process Start
    pthread_create(NULL, hello_thread,
        NULL, NULL);
    pthread_exit() //Some kind of scheduling
}
```

Some Kernel Calls Explained

- Operates in 'kernel space' (Where kernel code runs in ring 0 - access to full CPU instruction set).

Some Kernel Calls Explained

- Operates in 'kernel space' (Where kernel code runs in ring 0 - access to full CPU instruction set).
- The kernel can do anything (e.g. accessing devices and IO). Need a layer of security preventing unwated changes in the state of the system, especially in undefined ways.

Some Kernel Calls Explained

- Operates in 'kernel space' (Where kernel code runs in ring 0 - access to full CPU instruction set).
- The kernel can do anything (e.g. accessing devices and IO). Need a layer of security preventing unwated changes in the state of the system, especially in undefined ways.
- Kernel space is also lower level - you don't get the nice abstractions like sbrk(2) or write(2). The kernel has to route each of the requests to the appropriate drivers or handlers.

Some Kernel Calls Explained

- Operates in 'kernel space' (Where kernel code runs in ring 0 - access to full CPU instruction set).
- The kernel can do anything (e.g. accessing devices and IO). Need a layer of security preventing unwated changes in the state of the system, especially in undefined ways.
- Kernel space is also lower level - you don't get the nice abstractions like sbrk(2) or write(2). The kernel has to route each of the requests to the appropriate drivers or handlers.
- Exposed kernel calls are called system calls.

Kernel Call Example

- **void * kmalloc(size_t size, int flags)**
- Flags allow to users to use special kernel actions
- flags are one of many options the kernel handles.
  - `GFP_ATOMIC` - Cannot be interrupted, will not sleep. (Useful inside interrupt handlers)
  - `GFP_NOIO` - No disk I/O can be performed during request.
- We can have an entire lecture on this one call so just believe that this call returns pages of memory that are greater than size. A page is usually 4KB, so it'll be the smallest multiple of that.

Kernel Call Example

- **mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)** Takes a file, puts in to memory.

Booting

- BIOS loads up, MBR loads up, GRUB loads up, Then starts the Kernel

Booting

- BIOS loads up, MBR loads up, GRUB loads up, Then starts the Kernel
- The first thing the kernel does is start init (the main process for your operating system).

Booting

- BIOS loads up, MBR loads up, GRUB loads up, Then starts the Kernel
- The first thing the kernel does is start init (the main process for your operating system).
- Init does a lot of things. One important thing it does is initializing fork(), the magical library call that starts the entire process.

Booting

- BIOS loads up, MBR loads up, GRUB loads up, Then starts the Kernel
- The first thing the kernel does is start init (the main process for your operating system).
- Init does a lot of things. One important thing it does is initializing fork(), the magical library call that starts the entire process.
- Then init checks run level and runs the appropriate startup scripts.

System Calls

Motivation, Again

```c
#include <pthread.h> //Dynamic Linked Library

void* hello_thread(void *payload){ //Pthread
    write(1, "Hello world!", 12); //Sys Call
    return NULL;
}

int main(){ //Process Start
    pthread_create(NULL, hello_thread,
        NULL, NULL);
    pthread_exit() //Some kind of scheduling
}
```

How do we call a system call?

Typically C library calls call system calls but here is some x86 to get the job done.

```
_start:
movl $4, %eax     ; use the write syscall
movl $1, %ebx     ; write to stdout
movl $msg, %ecx   ; use string "Hello World"
movl $12, %edx    ; write 12 characters
int  $0x80        ; make syscall

movl $1, %eax     ; use the _exit syscall
movl $0, %ebx     ; error code 0
int  $0x80        ; make syscall
```

What is a system call?

- A system call is a call a program makes in user space that gets executed by the kernel.

What is a system call?

- A system call is a call a program makes in user space that gets executed by the kernel.
- Use a software interrupt that takes control back to the kernel.

What is a system call?

- A system call is a call a program makes in user space that gets executed by the kernel.
- Use a software interrupt that takes control back to the kernel.
- Kernel traps the signal, routes to the appropriate system call - executes the system call in Kernel space

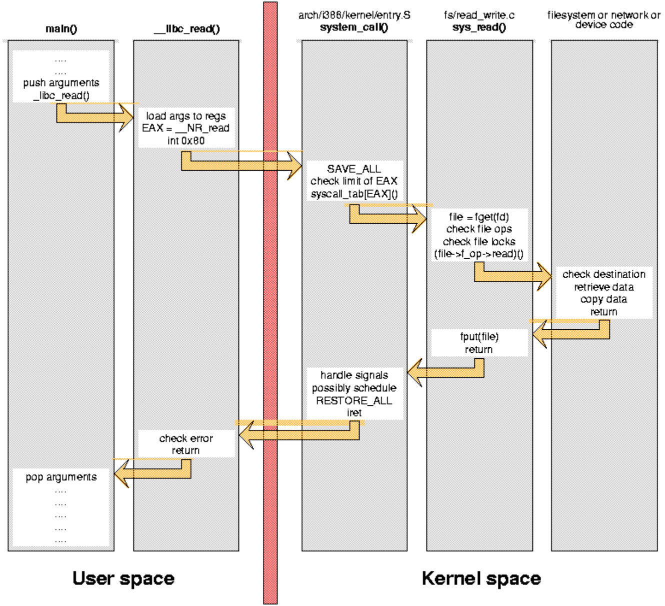
What is a system call?

- A system call is a call a program makes in user space that gets executed by the kernel.
- Use a software interrupt that takes control back to the kernel.
- Kernel traps the signal, routes to the appropriate system call - executes the system call in Kernel space
- We can load up to five of the system call parameters in registers (What if there are more?).

What is a system call?

- A system call is a call a program makes in user space that gets executed by the kernel.
- Use a software interrupt that takes control back to the kernel.
- Kernel traps the signal, routes to the appropriate system call - executes the system call in Kernel space
- We can load up to five of the system call parameters in registers (What if there are more?).
- Return value is stored in a register and returned back to userspace – we have a system call.

How do we call a system call?



Checking parameters

- The kernel has no restrictions, it can touch all memory in the system

Checking parameters

- The kernel has no restrictions, it can touch all memory in the system
- If there was no checks on the arguments passed in, users could get systems calls like memcpy, read or write to get access to files or regions of memory they shouldn't.

Checking parameters

- The kernel has no restrictions, it can touch all memory in the system
- If there was no checks on the arguments passed in, users could get systems calls like memcpy, read or write to get access to files or regions of memory they shouldn't.
- Need to make sure that all memory accesses are within the bounds of the user's virtual memory, do not point to kernel memory, and do not try to read or write from files they do not have permissions to.

Checking parameters

- The kernel has no restrictions, it can touch all memory in the system
- If there was no checks on the arguments passed in, users could get systems calls like memcpy, read or write to get access to files or regions of memory they shouldn't.
- Need to make sure that all memory accesses are within the bounds of the user's virtual memory, do not point to kernel memory, and do not try to read or write from files they do not have permissions to.
- Other permissions accomplished via linux's capabilities macros.

Checking parameters

- CAP_SYS_BOOT - Can reboot
- CAP_CHOWN - Can arbitrarily change file UIDs
- etc... see capabilities(7)

How to define a syscall?

- Awesome! How can I make a new syscall?

How to define a syscall?

- Awesome! How can I make a new syscall?
- Write the actual system call functionality

How to define a syscall?

- Awesome! How can I make a new syscall?
- Write the actual system call functionality
- Register the system call in the syscall table

How to define a syscall?

- Awesome! How can I make a new syscall?
- Write the actual system call functionality
- Register the system call in the syscall table
- Define the syscall number in <asm/unistd.h>

How to define a syscall?

- Awesome! How can I make a new syscall?
- Write the actual system call functionality
- Register the system call in the syscall table
- Define the syscall number in <asm/unistd.h>
- Compile the syscall into the kernel image

unistd.h

```c
#ifndef _UAPI_ASM_IA64_UNISTD_H
#define _UAPI_ASM_IA64_UNISTD_H
#include <asm/break.h>
#define __BREAK_SYSCALL
    __IA64_BREAK_SYSCALL
#define __NR_ni_syscall                1024
#define __NR_exit                      1025
#define __NR_read                      1026
#define __NR_write                     1027
#define __NR_open                      1028
#define __NR_close                     1029
#define __NR_creat                     1030
#define __NR_link                      1031
#define __NR_unlink                    1032
#define __NR_execve                    1033
```

Linking and Loading

Motivation, Again, Again

```c
#include <pthread.h> //Dynamic Linked Library

void* hello_thread(void *payload){ //Pthread
    write(1, "Hello world!", 12); //Sys Call
    return NULL;
}

int main(){ //Process Start
    pthread_create(NULL, hello_thread,
        NULL, NULL);
    pthread_exit() //Some kind of scheduling
}
```

What are we even talking about?

There are two types of libraries, those compiled with your programs and those that are linked dynamically at runtime. There are many benefits to use programs that get compiled with your program, but some drawbacks.

Cost-Benefit Analysis: Compile-Time Libraries

Benefits

- You have the source code/debug checking
- All code is in your code segment
- You can modify the library

Cost-Benefit Analysis: Compile-Time Libraries

Benefits

- You have the source code/debug checking
- All code is in your code segment
- You can modify the library

Drawbacks

- Updating is often tedious
- Your executable is bigger
- Your library cannot be reused by other applications

Solution: Dynamic Libraries

- What if we have one library that a bunch of programs can use (make it read only) and have it dynamically link the function calls in the program?

Solution: Dynamic Libraries

- What if we have one library that a bunch of programs can use (make it read only) and have it dynamically link the function calls in the program?
- Updating your executable's library is a piece of cake

Solution: Dynamic Libraries

- What if we have one library that a bunch of programs can use (make it read only) and have it dynamically link the function calls in the program?
- Updating your executable's library is a piece of cake
- Reduce the size of your executable

Solution: Dynamic Libraries

- What if we have one library that a bunch of programs can use (make it read only) and have it dynamically link the function calls in the program?
- Updating your executable's library is a piece of cake
- Reduce the size of your executable
- That library can be used by other applications.

Dynamic Lookup table?

- Cool! But where are the library functions?

Dynamic Lookup table?

- Cool! But where are the library functions?
- Any problem in computer science and be solved with another layer of abstraction.

Dynamic Lookup table?

- Cool! But where are the library functions?
- Any problem in computer science and be solved with another layer of abstraction.
- Have the functions in your code point to pointer where the functions are going to be.

Dynamic Lookup table?

- Cool! But where are the library functions?
- Any problem in computer science and be solved with another layer of abstraction.
- Have the functions in your code point to pointer where the functions are going to be.
- Have your code jump to the pointer and the pointer jump to the actual function

Where is the table stored

"In Unix-like systems that use ELF for executable images and dynamic libraries, such as Solaris, 64-bit versions of HP-UX, Linux, FreeBSD, NetBSD, OpenBSD, and DragonFly BSD, the path of the dynamic linker that should be used is embedded at link time into the .interp section of the executable's PT_INTERP segment. In those systems, dynamically loaded shared libraries can be identified by the filename suffix .so (shared object)."
- Wikipedia

So what does that mean?

- Exec generates lookup-table (PT_INTERP segment)
- Calls to a library will cause a jump to the lookup table
- Lookup table entry will redirect the program to the library function
- After execution, returns back to your program

So about that library

- During exec, the function checks what libraries you need.

So about that library

- During exec, the function checks what libraries you need.
- If that library is already loaded into memory, increase the reference count and link the functions.

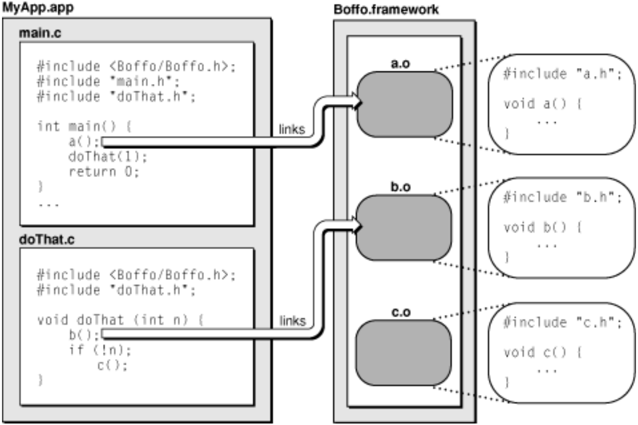
So about that library

- During exec, the function checks what libraries you need.
- If that library is already loaded into memory, increase the reference count and link the functions.
- Else, mmap the library into memory. Set the executable bit (memory can either be executable or writeable) and then link the function. Cache this library's location.

So about that library

- During exec, the function checks what libraries you need.
- If that library is already loaded into memory, increase the reference count and link the functions.
- Else, mmap the library into memory. Set the executable bit (memory can either be executable or writeable) and then link the function. Cache this library's location.
- When a process is done, reduce the reference count and return the page back to the system if need be.

So about that library



Drawbacks: Hacking

- Perfect! Let's dynamically link ALL THE THINGS!

Drawbacks: Hacking

- Perfect! Let's dynamically link ALL THE THINGS!
- Not quite. You have to trust the library you are linking to. What if the library has been hacked?

Drawbacks: Hacking

- Perfect! Let's dynamically link ALL THE THINGS!
- Not quite. You have to trust the library you are linking to. What if the library has been hacked?
- Some viruses redirect library calls (hint: LD_PRELOAD) thus, executing commands that the user is not aware of.

Drawbacks: Hacking with Bugs

- What if there is a bug in the DLL? Added point of failure.

Drawbacks: Hacking with Bugs

- What if there is a bug in the DLL? Added point of failure.
- Example: libc had a buffer overflow bug (any application that uses libc was affected).

Drawbacks: Hacking with Bugs

- What if there is a bug in the DLL? Added point of failure.
- Example: libc had a buffer overflow bug (any application that uses libc was affected).
- But the pros outweigh the cons so DLLs are here to stay.

Drawbacks: Hacking with Bugs

- Demo time!

Processes and Threads

Processes

Starting a process

Here's how to start a process:

Starting a process

Here's how to start a process:

- Make parent sleep

Starting a process

Here's how to start a process:

- Make parent sleep
- Fork off of an existing process (bash, terminal, init, ...)

Starting a process

Here's how to start a process:

- Make parent sleep
- Fork off of an existing process (bash, terminal, init, ...)
- Fork copies the file descriptors, page tables, signal handlers using kmalloc.

Starting a process

Here's how to start a process:

- Make parent sleep
- Fork off of an existing process (bash, terminal, init, ...)
- Fork copies the file descriptors, page tables, signal handlers using kmalloc.
- Wake up child first. (Why is this useful? See next slide)

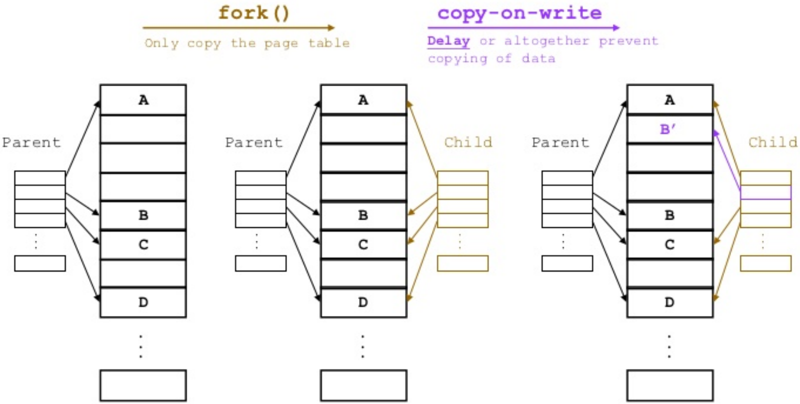
Starting a process

Here's how to start a process:

- Make parent sleep
- Fork off of an existing process (bash, terminal, init, ...)
- Fork copies the file descriptors, page tables, signal handlers using kmalloc.
- Wake up child first. (Why is this useful? See next slide)
- Imagine in the linux kernel there is a struct with all of this stuff – that is what a process essentially is.

Starting a process

Process Creation – Copy-on-Write



Now Exec-ing

- Exec takes an executable and uses the appropriate executable loader (ELF format for UNIX) to reorganize the file into memory. The kernel may mmap into new address spaces.

Now Exec-ing

- Exec takes an executable and uses the appropriate executable loader (ELF format for UNIX) to reorganize the file into memory. The kernel may mmap into new address spaces.
- The kernel then dynamically links libraries and destroys the pages that the old processes had.

Now Exec-ing

- Exec takes an executable and uses the appropriate executable loader (ELF format for UNIX) to reorganize the file into memory. The kernel may mmap into new address spaces.
- The kernel then dynamically links libraries and destroys the pages that the old processes had.
- The kernel resets registers and sets the stack pointer to the entry point of the main function. And finally, does the jump to the entrypoint. Your program is started!

Threads

Threads!

- Two types of threads - user space threads and kernel threads

Threads!

- Two types of threads - user space threads and kernel threads
- Kernel threads are used for things like system calls and monitoring. No need to context switch to user space!
  - kswapd - 'kernel swap daemon' handles swaping to/from disk
  - kworker - handles ACPI signals from BIOS

Threads!

- Two types of threads - user space threads and kernel threads
- Kernel threads are used for things like system calls and monitoring. No need to context switch to user space!
  - kswapd - 'kernel swap daemon' handles swaping to/from disk
  - kworker - handles ACPI signals from BIOS
- We use POSIX Portable Threads - pthreads! Threads in user-space.

Threads!

- Two types of threads - user space threads and kernel threads
- Kernel threads are used for things like system calls and monitoring. No need to context switch to user space!
  - kswapd - 'kernel swap daemon' handles swaping to/from disk
  - kworker - handles ACPI signals from BIOS
- We use POSIX Portable Threads - pthreads! Threads in user-space.
- But to the kernel, there are no things as threads.

What do you mean?

- To the kernel, everything is a process.

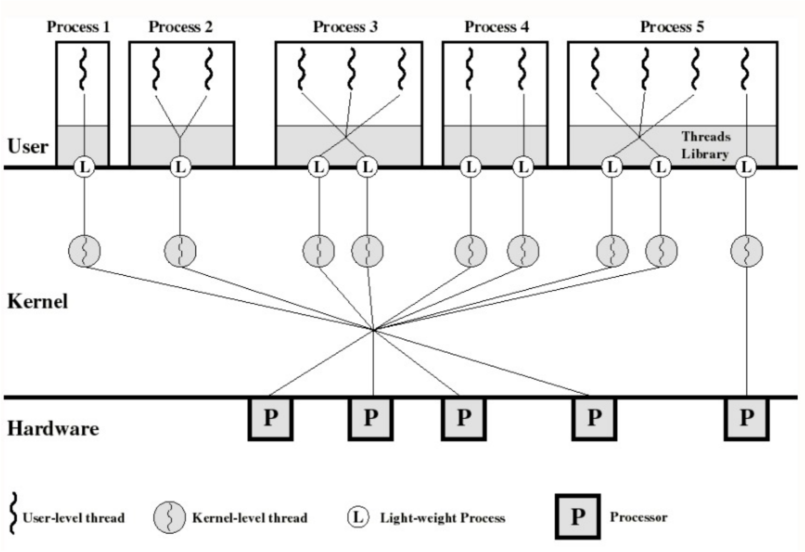
What do you mean?

- To the kernel, everything is a process.
- A thread is just a process that happens to share resources (such as memory, signal disposition, open files etc.) the original process.

What do you mean?

- To the kernel, everything is a process.
- A thread is just a process that happens to share resources (such as memory, signal disposition, open files etc.) the original process.
- This abstraction is really cool – that is why in the systems literature/papers *everything* is a process.

Threading Start - pthread_create



Threading Start - pthread_create

- After pthread initialization, set the attributes like detached state, stack address, and scheduling preferences.

Threading Start - pthread_create

- After pthread initialization, set the attributes like detached state, stack address, and scheduling preferences.
- Then, allocate some stack space in the current program's stack. Then the thread gets added to a table and a lightweight process is created using clone() [Think fork but no copying]

Threading Start - pthread_create

- After pthread initialization, set the attributes like detached state, stack address, and scheduling preferences.
- Then, allocate some stack space in the current program's stack. Then the thread gets added to a table and a lightweight process is created using clone() [Think fork but no copying]
- Then the pthread is added to the pthread table, and returns out of the pthread function.

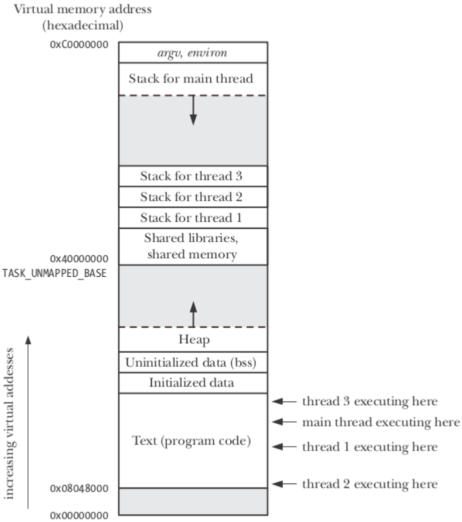
Threading Start - pthread_create

- After pthread initialization, set the attributes like detached state, stack address, and scheduling preferences.
- Then, allocate some stack space in the current program's stack. Then the thread gets added to a table and a lightweight process is created using clone() [Think fork but no copying]
- Then the pthread is added to the pthread table, and returns out of the pthread function.
- Scheduling the process is left up to the completely fair scheduler.

Threading Start - pthread_create

- Here's a quick demo of clone!

Threading Start - pthread_create



Pthread is running

- Acts like another process, that's why you can signal to it.

Pthread is running

- Acts like another process, that's why you can signal to it.
- You can sleep just like another process and use shared mutexes and whatnot because it resides in the same memory.

Pthread is running

- Acts like another process, that's why you can signal to it.
- You can sleep just like another process and use shared mutexes and whatnot because it resides in the same memory.
- Race conditions! All the fun stuff from processes

Pthread is running

- Acts like another process, that's why you can signal to it.
- You can sleep just like another process and use shared mutexes and whatnot because it resides in the same memory.
- Race conditions! All the fun stuff from processes
- (The kernel does know it's supposed to be treated as a thread and uses group scheduling for efficiency)

Pthread join

- Gets return values from the process

Pthread join

- Gets return values from the process
- We get rid of the stack space and the entry from the table.

Pthread join

- Gets return values from the process
- We get rid of the stack space and the entry from the table.
- Return the stack back to the program.

Completely Fair Scheduler

I promise last time.

```c
#include <pthread.h> //Dynamic Linked Library

void* hello_thread(void *payload){ //Pthread
    write(1, "Hello world!", 12); //Sys Call
    return NULL;
}

int main(){ //Process Start
    pthread_create(NULL, hello_thread,
        NULL, NULL);
    pthread_exit() //Some kind of scheduling
}
```

Okay let's backtrack to threads and stuff

- It is not a secret, we have more processes than CPUs – more threads than CPUs even

Okay let's backtrack to threads and stuff

- It is not a secret, we have more processes than CPUs – more threads than CPUs even
- So how does the CPU run all these processes? It switches between them really fast using what we call a scheduler.

Okay let's backtrack to threads and stuff

- It is not a secret, we have more processes than CPUs – more threads than CPUs even
- So how does the CPU run all these processes? It switches between them really fast using what we call a scheduler.
- This is essentially a dining philosopher problem that is solved by pre-emption. The kernel tells processes when they can hog resources like CPUs and tells them to stop whenever else.

Backstory

- Original implementation called the $O(1)$ scheduler

Backstory

- Original implementation called the $O(1)$ scheduler
- Constant time calculation of timeslices and per-processor runqueues

Backstory

- Original implementation called the $O(1)$ scheduler
- Constant time calculation of timeslices and per-processor runqueues
- Scaled reasonably well

Backstory

- Original implementation called the $O(1)$ scheduler
- Constant time calculation of timeslices and per-processor runqueues
- Scaled reasonably well
- Attempted to determine interactive vs non-interactive processes, got it wrong frequently.

Backstory - IO vs Processor

- Processes are either IO-bound or Process-bound

Backstory - IO vs Processor

- Processes are either IO-bound or Process-bound
- IO bound will spend most of it's time waiting for IO
- Needs to be run frequently or the system will appear to be annoyingly slow
- Process bound spends a lot of time actualy computing things
  - e.g. A text-editor vs computing a forward pass of a neural network

Backstory - IO vs Processor

- Processes are either IO-bound or Process-bound
- IO bound will spend most of it's time waiting for IO
- Needs to be run frequently or the system will appear to be annoyingly slow
- Process bound spends a lot of time actualy computing things
    - e.g. A text-editor vs computing a forward pass of a neural network

All processes are equal...

- We want to make sure that all processes are chugging along smoothly – we don't want our networking process to be starved when our process needs networking or our daemon to be starving either.

All processes are equal...

- We want to make sure that all processes are chugging along smoothly – we don't want our networking process to be starved when our process needs networking or our daemon to be starving either.
- What if we stopped thinking about timeslices?

All processes are equal...

- We want to make sure that all processes are chugging along smoothly – we don't want our networking process to be starved when our process needs networking or our daemon to be starving either.
- What if we stopped thinking about timeslices?
- Use percentages instead! Assign a percentage of the CPU to a process.
  - This way when an IO-bound process sleeps, it has a high priority when it wakes up.

All processes are equal...

- We want to make sure that all processes are chugging along smoothly – we don't want our networking process to be starved when our process needs networking or our daemon to be starving either.
- What if we stopped thinking about timeslices?
- Use percentages instead! Assign a percentage of the CPU to a process.
  - This way when an IO-bound process sleeps, it has a high priority when it wakes up.
- The CPU creates a Red-Black tree with the processes virtual runtime (runtime / nice_value) and sleeper fairness (if the process is waiting on something give it the CPU when it is done waiting). Since the red-black tree is self balancing pop operation is guaranteed $O(log(n))$.

But some are more equal than others

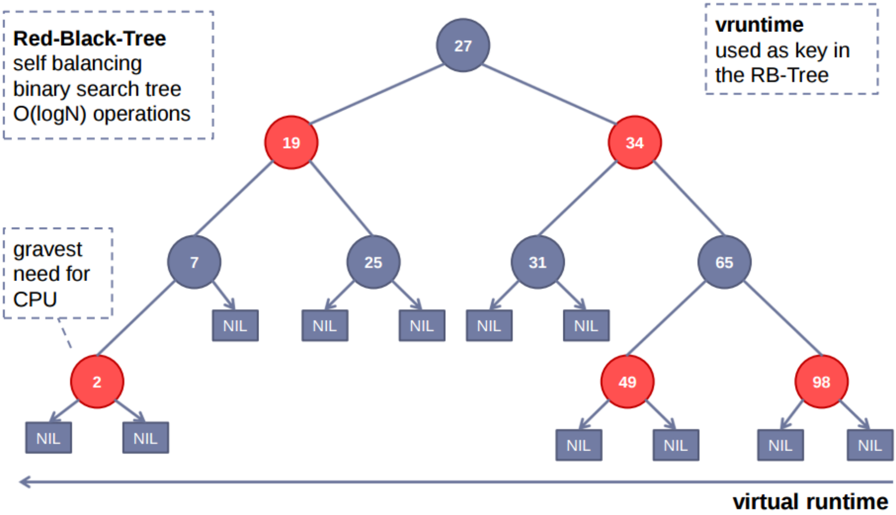
- Nice values are the kernel's way of yielding resources

But some are more equal than others

- Nice values are the kernel's way of yielding resources
- A higher nice value means that you are "nicer" and give up priority

But some are more equal than others

- Nice values are the kernel's way of yielding resources
- A higher nice value means that you are "nicer" and give up priority
- Run top to see examples!

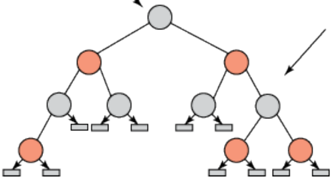

Red-Black-Tree
self balancing
binary search tree
O(logN) operations
vruntime
used as key in
the RB-Tree
27
19
34
7
25
31
65
gravest
need for
CPU
NIL
NIL
NIL
NIL
NIL
2
49
98
NIL
NIL
NIL
NIL
NIL
NIL
virtual runtime

Threads!

```
struct task_struct {
  volatile long state;
  void *stack;
  unsigned int flags;
  int prio, static_prio normal_prio;
  const struct sched_class *sched_class;
  struct sched_entity se;
  ...
};
```

```
struct sched_entity {
  struct load_weight load;
  struct rb_node run_node;
  struct list_head group_node;
  ...
};
```

```
struct ofs_rq {
   ...
   struct rb_root tasks_timeline;
   ...
};
```



```
struct rb_node {
  unsigned long rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
```

Looking back to threads

- Since threads are light-weight processes then they get scheduled like any other process and usually they get scheduled at the same time as your first process which usually results in race conditions for improper code.

Looking back to threads

- Since threads are light-weight processes then they get scheduled like any other process and usually they get scheduled at the same time as your first process which usually results in race conditions for improper code.
- The CFS tends to schedule groups of processes together – taking advantage of cache coherency, open files, open sockets etc.

Looking back to threads

- Since threads are light-weight processes then they get scheduled like any other process and usually they get scheduled at the same time as your first process which usually results in race conditions for improper code.
- The CFS tends to schedule groups of processes together – taking advantage of cache coherency, open files, open sockets etc.
- The CFS handles higher priority and long running processes fairly so no process fades away into the scheduling abyss.

CFS Problems

- Groups of processes that are scheduled may have imbalanced loads so the scheduler roughly distributes the load. When another CPU gets free it can only look at the average load of a group schedule not the individual cores. So the free CPU may not take the work from a CPU that is burning so long as the average is fine.

CFS Problems

- Groups of processes that are scheduled may have imbalanced loads so the scheduler roughly distributes the load. When another CPU gets free it can only look at the average load of a group schedule not the individual cores. So the free CPU may not take the work from a CPU that is burning so long as the average is fine.
- If a group of processes is running, on non adjacent cores then there is a bug. If the two cores are more than a hop away, the load balancing algorithm won't even consider that core. Meaning if a CPU is free and a CPU that is doing more work is more than a hop away, it won't take the work (may have been patched).

CFS Problems

- Groups of processes that are scheduled may have imbalanced loads so the scheduler roughly distributes the load. When another CPU gets free it can only look at the average load of a group schedule not the individual cores. So the free CPU may not take the work from a CPU that is burning so long as the average is fine.
- If a group of processes is running, on non adjacent cores then there is a bug. If the two cores are more than a hop away, the load balancing algorithm won't even consider that core. Meaning if a CPU is free and a CPU that is doing more work is more than a hop away, it won't take the work (may have been patched).
- After a thread goes to sleep on a subset of cores, when it wakes up it can only be scheduled on the cores that it was sleeping on. If those cores are now bus

Conclusion

Any questions? Thanks for sticking along!

Edit history

- Bhuvan Venkatesh
- Aneesh Durg

Sources

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2010). Operating system concepts. Hoboken: John Wiley & Sons.
- Love, R. (2015). Linux kernel development. Upper Saddle River: Addison-Wesley.
- Linux User's Manual