CS 241 Honors
Lecture 4 – Kernel

Bhuvan Venkatesh

University of Illinois Urbana–Champaign

February 14, 2017

What to go over

- Processes and Threads
- Linking and Loading
- System Calls
- Completely Fair Scheduler

Motivation

```
#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

- Kernel Calls are calls only the kernel can make, they are not exposed to the user directly – the calls operate in 'kernel space'. Exposed kernel calls are called system calls.

Some Kernel Calls Explained

- Kernel Calls are calls only the kernel can make, they are not exposed to the user directly – the calls operate in 'kernel space'. Exposed kernel calls are called system calls.
- **void * kmalloc(size_t size, int flags)** size is the number bytes of what you want to malloc; flags are one of many options the kernel handles. 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.

Some Kernel Calls Explained

- Kernel Calls are calls only the kernel can make, they are not exposed to the user directly – the calls operate in 'kernel space'. Exposed kernel calls are called system calls.
- **void * kmalloc(size_t size, int flags)** size is the number bytes of what you want to malloc; flags are one of many options the kernel handles. 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.
- **void kfree(const void* objp)** Self explanatory, frees the pages for later usage.

Some Kernel Calls Explained

- Kernel Calls are calls only the kernel can make, they are not exposed to the user directly – the calls operate in 'kernel space'. Exposed kernel calls are called system calls.
- **void * kmalloc(size_t size, int flags)** size is the number bytes of what you want to malloc; flags are one of many options the kernel handles. 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.
- **void kfree(const void* objp)** Self explanatory, frees the pages for later usage.
- **mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)** Takes a file, puts in to memory (it's a vast simplification, we can and have had another lecture on this).

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 sees what run level you are running at. Init then runs the appropriate startup scripts to start all the processes for your operating system.

Starting a process

Here's how to start a process:

Starting a process

Here's how to start a process:

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

Starting a process

Here's how to start a process:

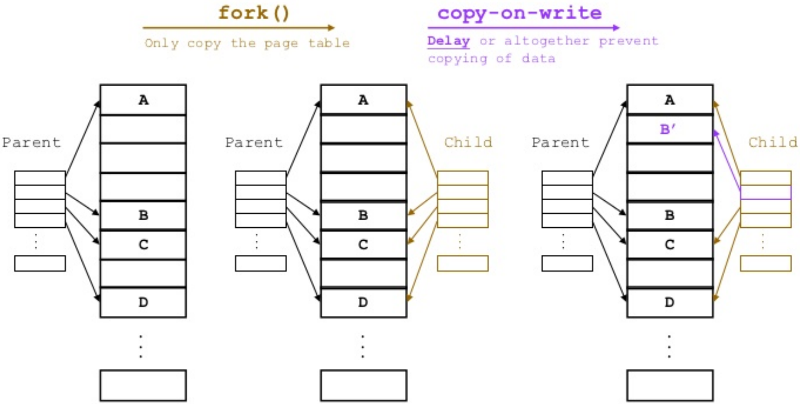
- Fork off of an existing process (bash, terminal, init, ...)
- Fork copies the file descriptors, page tables, signal handlers using kmalloc.

Starting a process

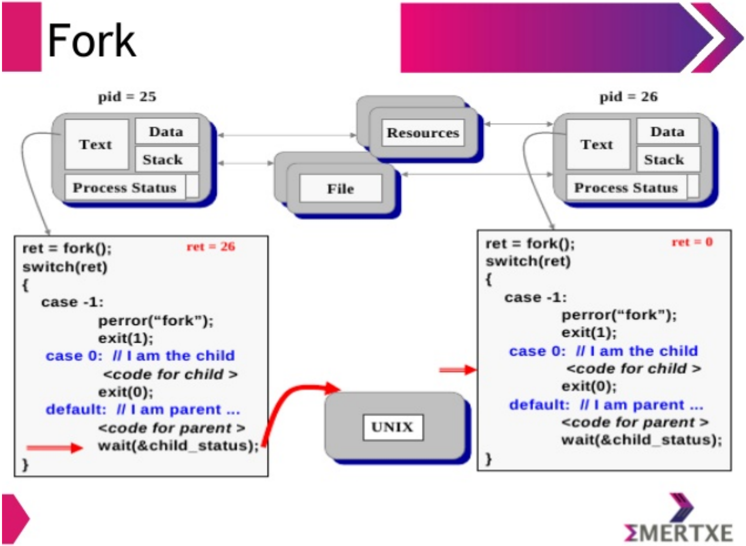
Here's how to start a process:

- Fork off of an existing process (bash, terminal, init, ...)
- Fork copies the file descriptors, page tables, signal handlers using kmalloc.
- Imagine in the linux kernel there is a struct with all of this stuff – that is what a process essentially is.

Process Creation – Copy-on-Write



Starting a process



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 (to be explained later). The kernel also sets the pages that the old processes had to be destroyed on exec.

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 (to be explained later). The kernel also sets the pages that the old processes had to be destroyed on exec.
- 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!

- 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 – you don't use them in normal use but may be used by your threading library.

Threads!

- Two types of threads - user space threads and kernel threads
- Kernel threads are used for things like system calls and monitoring – you don't use them in normal use but may be used by your threading library.
- We use 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 – you don't use them in normal use but may be used by your threading library.
- We use 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. A thread is just a process that happens to have the exact same memory, signal disposition, open files etc. etc. as the original process.

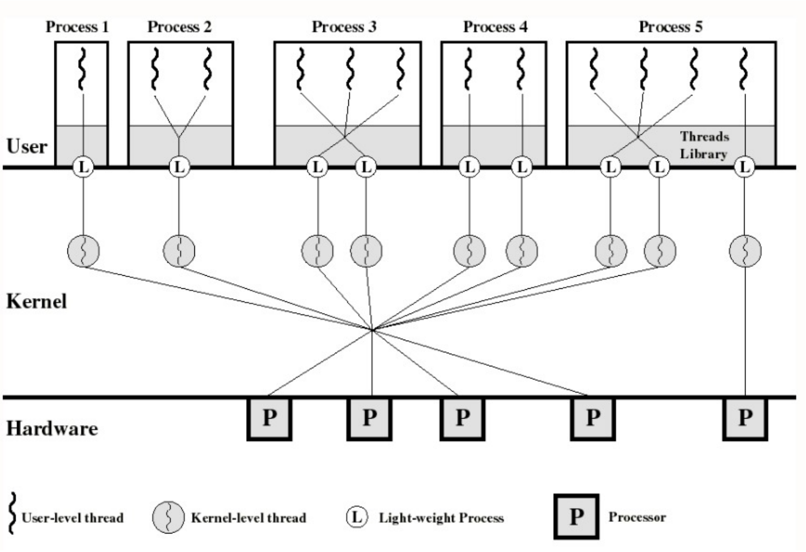
What do you mean?

- To the kernel, everything is a process. A thread is just a process that happens to have the exact same memory, signal disposition, open files etc. etc. as the original process.
- The completely fair scheduler then handles running the threads.

What do you mean?

- To the kernel, everything is a process. A thread is just a process that happens to have the exact same memory, signal disposition, open files etc. etc. as the original process.
- The completely fair scheduler then handles running the threads.
- 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, the pthread sets the attributes like detached state, stack address, and scheduling preferences.

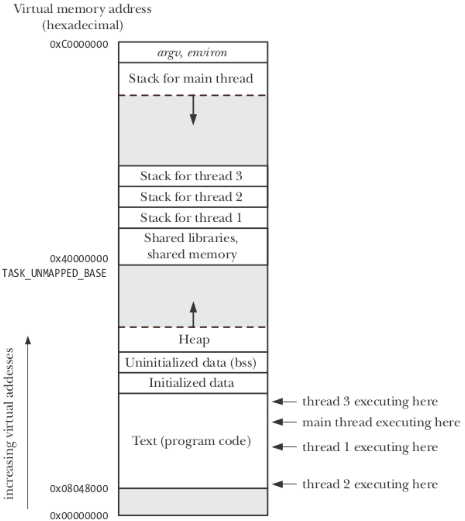
Threading Start - pthread_create

- After pthread initialization, the pthread sets the attributes like detached state, stack address, and scheduling preferences.
- Then pthread allocates some space for the stack 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, the pthread sets the attributes like detached state, stack address, and scheduling preferences.
- Then pthread allocates some space for the stack 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



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.

Linking and Loading

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
}
```

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?

- Any problem in computer science and be solved with another layer of abstraction.

Dynamic Lookup table?

- 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?

- 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 fills in the references to the library calls in the lookup table in the PT_INTERP segment of the program. Whenever the program makes a call to that library then the program will jump to the lookup table which will jump to the appropriate place in memory, executing the function and return control back to your function.

So about that library

- During exec, the function checks what libraries you need then there are two cases.

So about that library

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

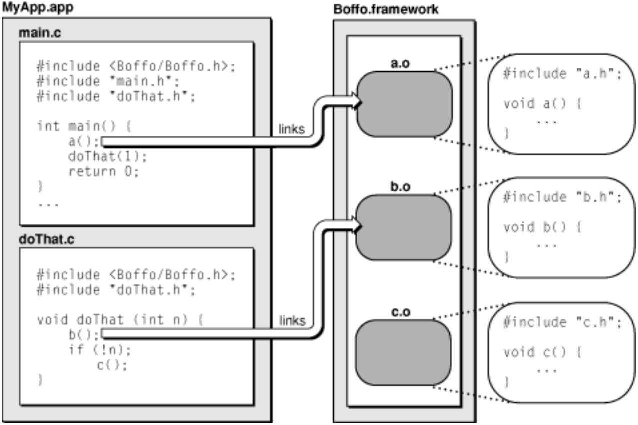
So about that library

- During exec, the function checks what libraries you need then there are two cases.
- If that library is already loaded into memory, increase the reference count and link the functions.
- If not, then mmap the library into memory. Set the executable bit (memory can either be executable or writeable) and then link the function. Store this library's location in case another program needs the same library.

So about that library

- During exec, the function checks what libraries you need then there are two cases.
- If that library is already loaded into memory, increase the reference count and link the functions.
- If not, then mmap the library into memory. Set the executable bit (memory can either be executable or writeable) and then link the function. Store this library's location in case another program needs the same library.
- 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

- One of the problems, inherently, with DLLs is that you have to trust the library you are linking to. Moreover you have to trust the user that the library has not been hacked.

Drawbacks: Hacking

- One of the problems, inherently, with DLLs is that you have to trust the library you are linking to. Moreover you have to trust the user that the library has not been hacked.
- Some viruses come prepackaged with libraries and if you set the LD_PRELOAD variable with a library when you execute commands that the user wouldn't even know.

Drawbacks: Hacking

- One of the problems, inherently, with DLLs is that you have to trust the library you are linking to. Moreover you have to trust the user that the library has not been hacked.
- Some viruses come prepackaged with libraries and if you set the LD_PRELOAD variable with a library when you execute commands that the user wouldn't even know.
- An example if we have time.

Drawbacks: Hacking with Bugs

- Another problem with a DLL is that if there is a bug in the DLL there is a bug in your program

Drawbacks: Hacking with Bugs

- Another problem with a DLL is that if there is a bug in the DLL there is a bug in your program
- libc had a buffer overflow bug not too long ago, any application that uses libc which is 99% of them were affected and could have been hacked away.

Drawbacks: Hacking with Bugs

- Another problem with a DLL is that if there is a bug in the DLL there is a bug in your program
- libc had a buffer overflow bug not too long ago, any application that uses libc which is 99% of them were affected and could have been hacked away.
- But there are always tradeoffs so DLLs are here to stay.

System Calls

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 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.
- System calls are called with interrupts control goes to the kernel and then gets bounced back to the program

What is a system call?

- A system call is a call a program makes in user space that gets executed by the kernel.
- System calls are called with interrupts control goes to the kernel and then gets bounced back to the program
- System calls need to do that because some calls require elevated privileges but within the bounds of the kernel - like accessing devices or doing I/O.

What is a system call?

- A system call is a call a program makes in user space that gets executed by the kernel.
- System calls are called with interrupts control goes to the kernel and then gets bounced back to the program
- System calls need to do that because some calls require elevated privileges but within the bounds of the kernel - like accessing devices or doing I/O.
- These are vital to change the state of the system, or to communicate with other processes for example

User Space and Kernel Space

- Userspace is where you run your programs in, the user runs their programs using the kernel's system calls.

User Space and Kernel Space

- Userspace is where you run your programs in, the user runs their programs using the kernel's system calls.
- We need this separation because the Kernel can do anything. If you don't have this separation then any program could potentially change the state of the system in undefined ways – a huge security hole.

User Space and Kernel Space

- Userspace is where you run your programs in, the user runs their programs using the kernel's system calls.
- We need this separation because the Kernel can do anything. If you don't have this separation then any program could potentially change the state of the system in undefined ways – a huge security hole.
- Kernel space is also lower level meaning that 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.

User Space and Kernel Space

- Userspace is where you run your programs in, the user runs their programs using the kernel's system calls.
- We need this separation because the Kernel can do anything. If you don't have this separation then any program could potentially change the state of the system in undefined ways – a huge security hole.
- Kernel space is also lower level meaning that 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.
- Kernel Space still needs to be exposed to system calls.

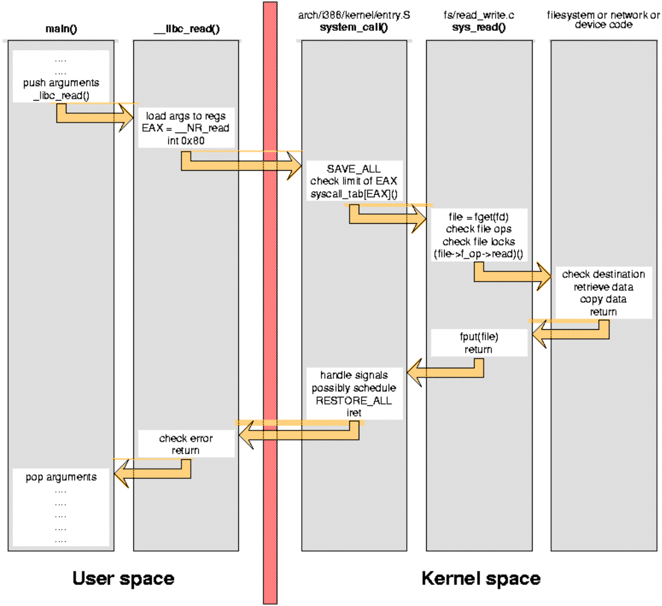
How do we call a system call?

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
```

How do we call a system call?



How do we call a system call?

- We have to load all of the system call parameters in registers. Then we program a software interrupt that takes control back to the kernel.

How do we call a system call?

- We have to load all of the system call parameters in registers. Then we program a software interrupt that takes control back to the kernel.
- Then the kernel traps the signal, routes to the appropriate system call, looks at the registers and executes the system call in Kernel space

How do we call a system call?

- We have to load all of the system call parameters in registers. Then we program a software interrupt that takes control back to the kernel.
- Then the kernel traps the signal, routes to the appropriate system call, looks at the registers and executes the system call in Kernel space
- The kernel stores the result in a register and returns back to userspace – we have a system call.

How do we call a system call?

- We have to load all of the system call parameters in registers. Then we program a software interrupt that takes control back to the kernel.
- Then the kernel traps the signal, routes to the appropriate system call, looks at the registers and executes the system call in Kernel space
- The kernel stores the result in a register and returns back to userspace – we have a system call.
- Demo if I can hack this kernel solution together

Completely Fair Scheduler

I promise last time.

```
#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.

Deadlock solved, now starvation

- 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.

Deadlock solved, now starvation

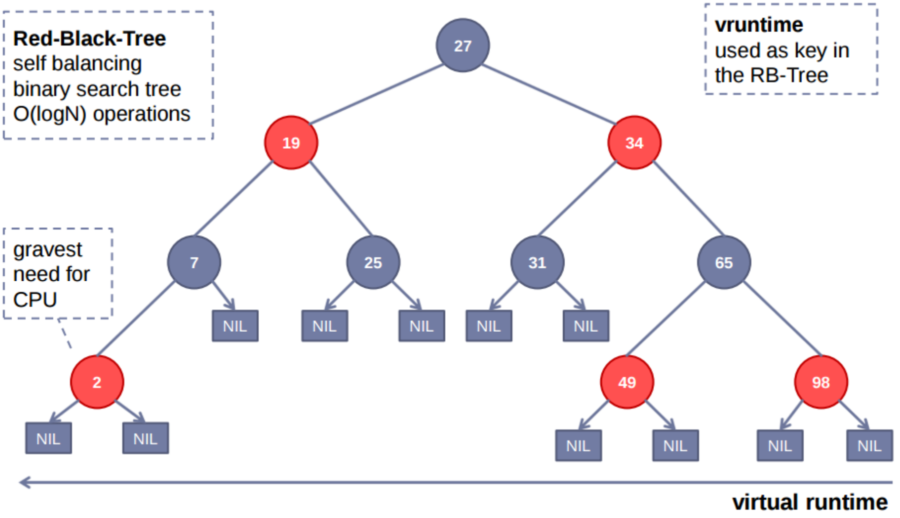
- We want to make sure that all processes are chugging along smoothly – we don't want our networking process to be starved when our process needs networking or our daemon to be starving either.
- The CPU creates a Red-Black tree with the processes virtual runtime (runtime / nice_value) and sleeper fairness (if the process is waiting on something give it the CPU when it is done waiting).

Deadlock solved, now starvation

- 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.
- 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).
- (Nice values are the kernel's way of giving priority to certain processes, the lower the higher priority)

Deadlock solved, now starvation

- 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.
- 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).
- (Nice values are the kernel's way of giving priority to certain processes, the lower the higher priority)
- The kernel chooses the lowest one based on this metric and schedules that process to run next, taking it off the queue. Since the red-black tree is self balancing this operation is guaranteed $O(log(n))$ (selecting the min process is the same runtime)

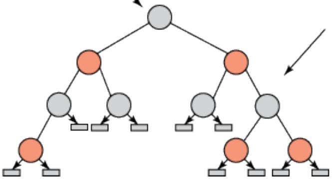

Red-Black-Tree
self balancing
binary search tree
O(logN) operations
vruntime
used as key in
the RB-Tree
gravest
need for
CPU
27
19
34
7
25
31
65
2
49
98
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
virtual runtime

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!