CS 350 Operating Systems

Keven Qiu
Instructor: Bernard Wong
Winter 2024

Contents

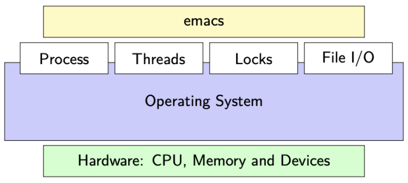
Chapter 1

Introduction

Definition: Operating System (OS)

The layer between applications and hardware.

It usually provides abstractions for applications. This includes managing and hiding details of hardware and can access hardware through low level interfaces unavailable to applications. It often provides protection.



Primitive OS are just a library of standard services (no protection). The system runs one program at a time and there are no bad users or programs.

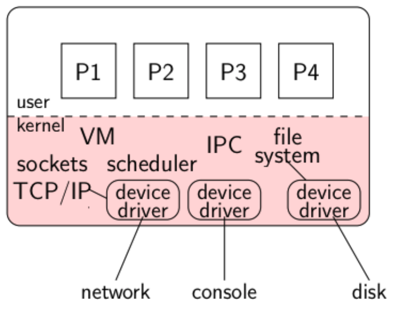
Multitasking is the idea of running more than one process at once. An ill-behaved process can go into an infinite loop and never relinquish the CPU. It can also scribble over other processes' memory. The OS provides mechanisms to address these problems:

- Preemption: take CPU away from looping.
- Memory protection: protect process' memory.

Multi-user OS use protection to serve distrustful users/apps. With n users, the system is not n times slower. Users can use too much CPU (need policies), total memory usage is greater than in the machine (must virtualize), or a super-linear slowdown with increasing demand (thrashing).

The OS structure: Most software runs as user-level processes and the OS kernel runs in privileged mode.

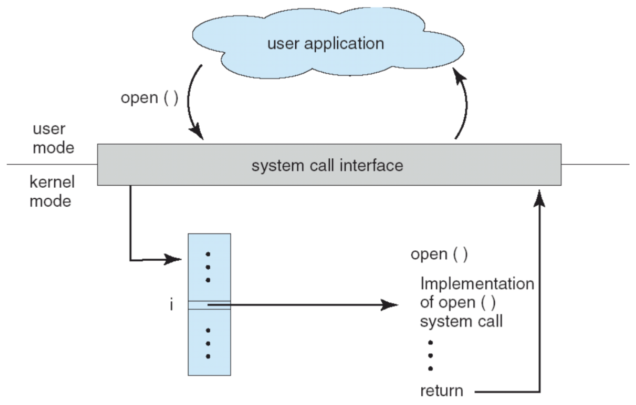
Protection:



- Pre-emption: give application a resource, take it away if needed elsewhere.
- Interposition/mediation: place the OS between application and "stuff". Track all pieces that the application is allowed to use. On every access, look in the table to check that access is legal.
- Privileged & unprivileged modes in CPUs: applications are unprivileged and the OS is privileged. Protection operations can only be done in privileged mode.

Definition: System Calls

Applications can invoke the kernel through system calls. These are special instructions that transfers control to the kernel.



The goal for system calls is to do things applications cannot do in unprivileged mode. The kernel supplies well-defined system call interface.

Applications set up syscall arguments and trap to kernel. The kernel performs operation and returns result.

Higher-level functions are built on the syscall interface. E.g. `printf`, `scanf`, `gets`, etc.

These call the POSIX/UNIX interface. E.g. `open`, `close`, `read`, `write`, etc.

Chapter 2

Processes

Definition: Process

An instance of a program running and the environment for running the program.

Modern OSes run multiple processes simultaneously.

- Multiple processes increase CPU utilization: overlap one process' computation with another.
- Multiple processes can reduce latency: running A then B may require more time than running A and B concurrently.

Each process has its own view of the machine. It has its own address space, open files, and virtual CPU.

2.1 User View of Processes

2.1.1 Creating Processes

`int fork(void);` creates a new process that is an exact copy of the current one. It returns the process ID of the new process in the parent. It returns 0 in the child.

The only part that is shared between the parent and child are the open files, otherwise the address space and code are its own.

`int waitpid(int pid, int *stat, int opt);` waits for a child process to terminate. `pid` is the process to wait for, or -1 for any. `stat` contains the exit value, or signal. `opt` is usually 0 or `WNOHANG`. `waitpid` returns the process ID that it waited for, or -1 if error.

2.1.2 Deleting Processes

`void exit(int status);` exits the current process. By convention, status of 0 is success and non-zero is error.

`int kill(int pid, int sig);` sends the signal `sig` to process `pid`. `SIGTERM` is the most common value and kills the process by default. `SIGKILL` is stronger and always kills the process.

2.1.3 Running Programs

`int execve(char *prog, char **argv, char **envp);` executes a new program. `prog` is the full path name of the program to run, `argv` is the argument vector that gets passed to main, and `envp` are the environment variables (PATH, HOME, etc.).

It is generally called through wrapper functions.

- `int execvp(char *prog, char **argv);` searches PATH for `prog` and use current environment.
- `int execlp(char *prog, char *arg, ...);` list arguments one at a time and finish with `NULL`.

2.1.4 Manipulating File Descriptors

`int dup2(int oldfd, int newfd);` closes `newfd`, if it was a valid descriptor and makes `newfd` an exact copy of `oldfd`.

Two file descriptors will share the same offset, i.e. `lseek` will affect both.

2.1.5 Pipes

This allows reading and writing between two processes like the parent and child.

`int pipe(int fds[2]);` returns two file descriptors in `fds[0]` and `fds[1]`. `fds[0]` reads and `fds[1]` writes. When the last copy of `fds[1]` is closed, `fds[0]` will return EOF. `pipe` will return 0 on success and -1 on error.

Operations on pipes:

- read/write/close.
- When `fds[1]` is closed, `read(fds[0])` returns 0 bytes.

- When `fds[0]` is closed, `write(fds[1])` kills the process with SIGPIPE or if the signal is ignored, it fails with EPIPE.

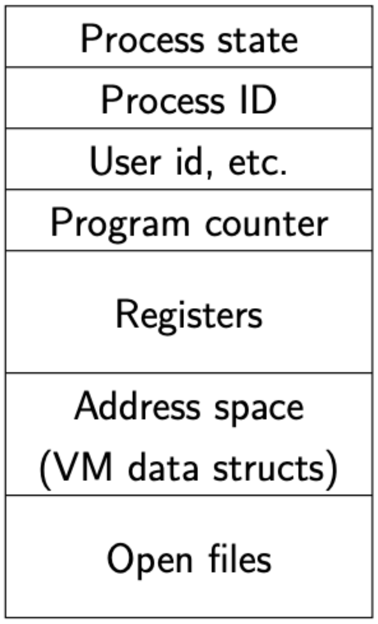
Most calls to fork are followed by execve. We can also combine into one spawn system call.

Without fork, we requires tons of different options.

2.2 Kernel View of Processes

2.2.1 Implementing Processes

The OS keeps a data structure for each process called the Process Control Block (PCB). It is called `proc` in Unix, `task_struct` in Linux, and `struct Process` in COS.



PCB

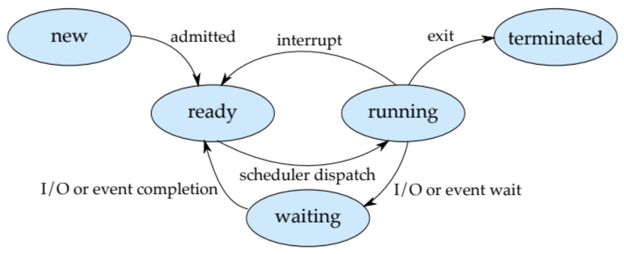
It tracks the state of the process (running, ready, blocked, etc.) and includes information necessary for it to run such as registers, virtual memory mappings, etc. There are also other data about the process like credentials, signal mask, controlling terminal, priority, etc.

Process States:

- New/Terminated: at beginning/end of life.
- Running: currently executing.
- Ready: can run, but kernel has chosen a different process to run.

- Waiting: needs async even (e.g. disk operation) to proceed.

If 0 processes are runnable, run an idle loop or halt CPU, otherwise if there is 1, run that process. If > 1 processes are runnable, it must make a scheduling decision.



Scanning the process table for the first runnable process is expensive.

We can use FIFO/round-robin to pick which process to run first.

Preemption: We can preempt a process when the kernel gets control. A running process can vector control to the kernel, a period timer interrupt, device interrupt, or changing running process to another (context switching).

Context Switching: Typical things include

- Saving program counter and integer registers.
- Save floating point or other special registers.
- Save condition codes.
- Change virtual address translations.

Some non-negligible costs are saving/restoring floating point registers, flushing the TLB, and usually causes more cache misses.

Chapter 3

Threads

Definition: Thread

A schedulable execution context.

An execution context contains a program counter, memory, etc. to tell where to execute code from.

Multi-threaded programs share the address space.

Definition: Kernel Thread

A thread that is scheduled for execution by the kernel.

Each kernel thread is assigned to one CPU core.

Definition: User Thread

A thread that is scheduled for execution by a user space threading library.

POSIX Thread API:

- `int pthread_create(pthread_t *thr, pthread_attr_t *attr, void *(*fn)(void *), void *arg);`: creates a new thread identified by `thr` with optional attributes, run `fn` with `arg`.
- `int pthread_exit(void *return_value);`: destroys current thread and return a pointer.
- `int pthread_join(pthread_t thread, void **return_value);`: wait for thread `thread` to exit and receive the return value.
- `void pthread_yield();`: tell the OS scheduler to run another thread or process.

We can implement `pthread_create` as a system call. To add `pthread_create` to an OS:

1. Start with process abstraction in kernel.
2. `pthread_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.

This is faster than a process, but still very heavy weight.

Limitations of Kernel-Level Threads:

- Every thread operation must go through the kernel: syscall takes 100 cycles, function calls take 2 cycles. This results in 10-30x slower threads when implemented in a kernel.
- One-size fits all thread implementation.
- General heavy-weight memory requirements.

User Threads: Implement a user-level library where there is one kernel thread per process.

Implementing User-Level Threads:

1. Allocate a new stack for each `pthread_create`.
2. Keep a queue of runnable threads.
3. Replace blocking system calls.
4. Schedule periodic timer signal (`setitimer`)

Limitations of User-Level Threads:

- Cannot take advantage of multiple CPUs or cores.
- A blocking system call blocks all threads.
- A page fault blocks all threads.
- Possible deadlock if one thread blocks another.

User Threads on Kernel Threads: There are multiple kernel-level threads per process. This is sometimes called $n : m$ threading; have n user threads per m kernel threads.

Limitations of $n : m$ Threading:

- Many of the same problems as $n : 1$ threading.
- Hard to keep same number of kernel threads as available CPUs.
- Kernel does not know relative importance of threads.

3.1 Case Study (Go Language)

3.1.1 Go Routines

Go Routines are very light-weight. Running 100k go routines is practical. It runs on a segmented stack and the OS thread typically allocated 2 MiB fixed stacks.

Go routines are on top of kernel threads with the $n : m$ model.

Each kernel-level thread finds and runs a go routine (user-level thread). Every logical core is owned by a kernel thread when running. It converts blocking system calls when possible to non-blocking ones in the runtime. This yields the CPU to another core.

3.2 Implementing Threads in CastorOS

AMD64/x86-64 Calling Conventions:

- Registers are divided into 2 groups: functions are free to clobber caller-saved registers (`%r10, %11`), but must restore callee-saves ones to original value upon return (`%rbx, %r12-%r15`).
- `%rsp` register is always the base of the stack.
- Local variables are stored in registers and on stack.
- Function arguments go in caller-saved registers and on the stack: the first six arguments are `%rdi, %rsi, %rdx, %rcx, %r8, %r9`.
- Return value is `%rax` and `%rdx`

3.2.1 Threads vs. Procedures

Threads may resume out of order.It cannot use LIFO stack to save state. A general solution is to use 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:

3.2.2 CastorOS Threads

CastorOS supports both kernel and user threads.

`Thread *Thread_KThreadCreate(void (*f)(void *), void *arg);` creates a kernel thread associated with the process.

`Thread *Thread_UThreadCreate(Thread *oldThr, uint64_t rip, uint64_t arg);` creates a userspace thread (and associated kernel thread).

All thread switches go through `Sched_Scheduler()` and `Sched_Switch()`. `Sched_Switch()` calls `Thread_SwitchArch` that runs `switchstack`. `switchstack` switches from one stack to other while saving and restoring registers.

Chapter 4

Concurrency

Recall that a process is an instance of a running program and a thread is an execution context. The POSIX thread API contains `pthread_create()`, `pthread_exit()`, `pthread_join()`.

4.1 Critical Sections

Definition: Critical Section

Part of a concurrent program in which a shared object is accessed.

```
int total = 0;
void add() {
    for (int i = 0; i < n; i++)
        total++;
}
void sub() {
    for (int i = 0; i < n; i++)
        total--;
}
```

Let thread 1 run add and thread 2 run sub.

1. Schedule 1 (add then sub): This increments then decrement, so the total is 0.
2. Schedule 2 (Alternate assembly instructions): Both loads, then add 1/sub 1 in the register, but the store in total is -1.
3. Schedule 3 (Load/add, sub, store, store): Total gets 1.

To prevent race conditions, we can enforce mutual exclusion on critical sections in the code.

Desired properties of a solution:

- Mutual exclusion: only one thread can be in the critical section at a time.
- Progress: if no process is currently in the critical section, one of the processes trying to enter will get in.
- Bounded waiting: once a thread T starts trying to enter the critical section, there is a bound on the number times other threads get in.
- Progress vs. Bounded waiting: if no thread can enter critical section, we don't have progress. If thread A is waiting to enter the critical section while B repeatedly leaves and re-enters, we don't have bounded waiting.

4.1.1 Peterson's Solution

```
// t is the current thread's id
int not_turn = 0; // not this thread's turn to enter CS
bool w[2] = {false, false}; // w[i] if thread i wants to enter CS
int total = 0;
void add() {
    for (int i = 0; i < n; i++) {
        w[t] = true;
        n_turn = t;
        while (w[1-t] && n_turn == t);
        total++;
        w[t] = false;
    }
}
void sub() {
    for (int i = 0; i < n; i++) {
        w[t] = true;
        n_turn = t;
        while (w[1-t] && n_turn == t);
        total--;
        w[t] = false;
    }
}
```

Mutual exclusion: both threads cannot be both in the critical section, `n_turn` prevents this.

Progress: If T_{1-i} is not in the critical section, it can't block T_i.

Bounded waiting: If T_i wants to lock and T_{1-i} tries to re-enter, T_{1-i} will set `n_turn=1-i`, allowing T_i in.

Peterson is expensive. This implementation requires *Sequential Consistency*.

4.2 CPU and Compiler Consistency

Definition: 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 in the program.

The boils down to two requirements

1. Maintaining program order on individual processors.
2. Ensuring write atomicity.

Without sequential consistency, multiple CPUs can be worse than preemptive threads.

Sequential consistency thwarts hardware optimizations. It cannot reorder overlapping write operations, reduces opportunities for data prefetching, and makes cache coherence more expensive.

Sequential consistency thwarts compiler optimizations. It causes code motion, caching value in register, common subexpression elimination, and loop blocking.

Often we reason about concurrent code assuming sequential consistency. However, for low-level code, we need to know the memory model. For most code, avoid depending on the memory model.

Hardware-Specific Synchronization Instructions: Used to implement synchronization primitives like locks. Example using `xchg`:

```
Xchg(value, addr) {
    old = *addr;
    *addr = value;
    return(old);
}

Acquire(bool *lock) {
    while (Xchg(true, lock) == true);
}

Release(bool *lock) {
    *lock = false; // give up lock
}
```

If `Xchg` returns true, the lock was already set and we continue to loop. If it returns false, then the lock was free, so we have acquired it.

This is known as a *spinlock*, since a thread busy-waits (loops) in Acquire until the lock is free.

This implementation is almost correct without sequential consistency. This is because Release can be inlined when it is called, which can interfere with other stores. We can fix it by adding a fence or replacing `*lock=false` with `Xchg(false, lock)`.

Spinlocks provide mutual exclusion and progress, but no bounded waiting. A thread has no bound in calling `Xchg`. Another inefficiency is that spinlocks keep spinning until a lock can be acquired. Using mutexes, we let the thread know when a lock is ready.

4.3 Mutexes and Condition Variables

Most operating systems provide locks (also known as a mutex). They enforce mutual exclusion.

```
mutex_lock(lock);
// critical section
mutex_unlock(lock);
```

Spinlocks spin, locks block. A thread that calls `spinlock_acquire` spins until the lock can be acquired. A thread that calls `mutex_lock` blocks until the lock can be acquired. Mutexes utilize spinlocks for implementation.

Sometimes a thread needs to wait for something, such as

- a lock to be released by another thread.
- data from a relatively slow device.
- input from a keyboard.
- busy device to become idle.

When a thread blocks, it stops running. The scheduler chooses a new thread to run. A blocked thread is signaled and awakened by another thread.

4.3.1 Wait Channels

To implement thread blocking, wait channels are used. It manages a list of sleeping threads and abstract details of the thread scheduler. It is a queue.

- `void WaitChannel_Lock(WaitChannel *wc);` locks wait channel operations and prevents a race between sleep and wake.
- `void WaitChannel_Sleep(WaitChannel *wc);` blocks calling thread on wait channel wc.

- `void WaitChannel_WakeAll(WaitChannel *wc);` unblocks all threads sleeping on the wait channel.
- `void WaitChannel_Wake(WaitChannel *wc);` unblocks one thread sleeping on the wait channel.

There can be many different wait channels, holding threads that are blocked for different reasons.

PThread Mutex API:

- `int pthread_mutex_init` initializes a mutex.
- `int pthread_mutex_destroy` destroys a mutex.
- `int pthread_mutex_lock` acquires a mutex.
- `int pthread_mutex_unlock` releases a mutex.
- `int pthread_mutex_trylock` attempts to acquire a mutex, returns 0 if successful, -1 otherwise (`errno == EBUSY`).

All global data should be protected by a mutex. Global means it is accessed by more than one thread with at least one write.

Compiler/Runtime Contract

Assuming no data races, the program behaves sequentially consistent.

There are problems when trying to acquire a mutex a thread already owns. It has to rely on something else to wake the thread up.

Improved Producer/Consumer: In the producer, we add a `mutex_unlock` and `mutex_lock` in the loop to wait for the consumer to consume data. Similarly in the consumer.

This is not a good approach since the loop keeps spinning.

4.3.2 Condition Variables

To handle spinning, we want to inform the scheduler of which threads can run. This is done with condition variables.

We have the functions

- `int pthread_cond_init`: initializes with specific variables.
- `int pthread_cond_wait`: atomically unlock mutex m and sleep until c is signaled.

- `int pthread_cond_signal`
- `int pthread_cond_broadcast`: wake one/all threads waiting on c.

Now in the while loop, we can do a condition wait for the condition required. We can signal at the end of the critical section to signal a condition.

We use a while loop, not an if statement. This is because we want to always recheck the condition on wake-up.

We cannot separate the mutex and condition variables because the the producer can wait forever. So we have the mutex unlock and lock in the condition wait call.

4.4 Semaphores

Definition: Semaphore

A way to hold a counter.

Initialized with an integer N, `int sem_init(sem_t *s,..., unsigned int n);` and provides two functions:

- `sem_wait(sem_t *s);`: decreases the counter.
- `sem_post(sem_t *s);`: increases the counter.

`sem_wait` will return only N more times than `sem_post` called.

If $N = 1$, then this is a binary semaphore which is a mutex, with `sem_wait` as lock and `sem_post` as unlock.

Semaphores can give elegant solutions to some problems.

```
Semaphore_Wait(Semaphore *sem) {
    Spinlock_Lock(&sem->sem_lock);
    while (sem->sem_count == 0) {
        WaitChannel_Lock(sem->sem_wchan);
        Spinlock_Unlock(&sem->sem_lock);
        WaitChannel_Sleep(sem->sem_wchan);
        Spinlock_Lock(&sem->sem_lock);
    }
    sem->sem_count--;
    Spinlock_Unlock(&sem->sem_lock);
}
Semaphore_Post(Semaphore *sem) {
    Spinlock_Lock(&sem->sem_lock);
    sem->sem_count++;
    WaitChannel_Wake(sem->sem_wchan);
    Spinlock_Unlock(&sem->sem_lock);
}
```

4.5 Data Races