1 Introduction

An OS manages resources (hardware resources, resources among running programs, etc), creates execution environments, loads programs, provides common services and utilities. A real-time OS is one which guarantees a response to events within specific timing constraints. Also has shorter quantums for scheduling so that the OS is more responsive. Used in things such as anti-lock braking systems in cars.

The execution environment provides a program with the resources it needs to run, isolates running programs from one another, and provides an interface to system hardware (I/O devices, storage, etc) for the program to use.

The kernel is the part of the OS which stands between the user and hardware and abstracts low-level hardware so that user programs cannot interact with hardware directly. Besides the kernel, the OS also includes programs such as command interpreters (bash), utility programs (task managers), programming libraries, etc. The kernel is responsible for:

1. creating execution environments for user programs
2. handling interrupts, exceptions raised by the CPU; interrupts are raised by devices and exceptions are raised by software
3. responding to system calls
4. managing memory
5. implementing concurrency

Doing basically anything, even just adding two numbers, may require the kernel, which means requiring the OS. A monolithic kernel (eg Windows, Linux, macOS) includes everything in the kernel, such as device drivers, file systems, virtual memory, IPC, etc. A microkernel only puts the absolute necessities in the kernel and everything else is a user program.

Hosted environments are when programs are run with an OS available and the program enters in main. Freestanding does not have an OS available. Programs in freestanding include the bootloader and the OS itself.

2 Threads and Concurrency

A thread is a sequence of instructions. Normal sequential programs consist of one thread of execution. Threads do not have relationships (such as parent thread, etc). Multi-threading allows for,

1. Improved performance from threads executing simultaneously
2. Better resource utilization (don't let the CPU idle for too long)
3. Better responsiveness, for example dedicating certain threads to load the UI
4. A concept of priority where high priority threads run on the CPU for longer

In the OS, a thread is represented as a structure or object. All threads share access to the program's (process's) global variables and heap but each thread has its own stack and set of registers.

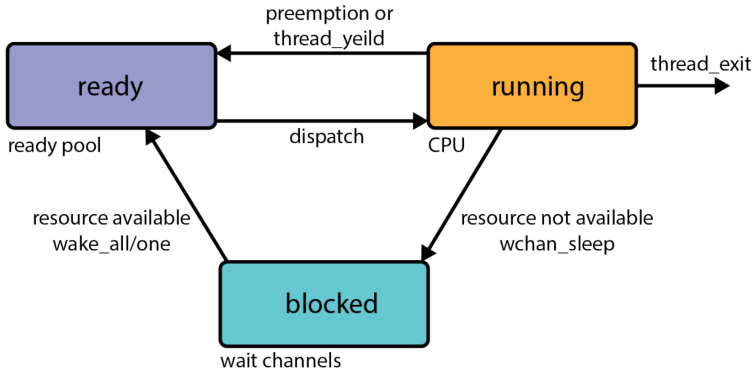
2.1 Interface (in OS/161)

```
/* Creates a new thread */
int thread_fork(const char *name,      // name (only for the programmer)
                struct proc *proc,     // process which the thread belongs to
                void (*func) (void *, unsigned long), // function to start at
                void *data1, unsigned long data);     // data to give to function
/* Terminates the calling thread. Otherwise, a thread terminates when it finishes
   running the function it was given on creation. */
void thread_exit(void);
/* Voluntarily yield execution, but is not a guarantee if there are no other threads
   to run, or it is the highest priority. Might be used to improve performance. */
void thread_yield(void);
```

Another popular function, although not in OS/161, is `thread_join`. If `t1` calls `t2.join`, then `t1` will block until `t2` finishes. This can be useful for threads which need the computation result from other threads before proceeding.

2.2 Implementations of multi-threading

There are three thread states: running (currently executing), ready (ready to execute), and blocked (waiting for something, cannot currently execute, is "asleep"). Transitions are,



Hardware Support: This is "true" multi-threading. If there are P processors, each with C cores, and each core with M pipelines (meaning M instructions can run at the same time), then PCM threads can execute simultaneously.

Timesharing: Threads take turns and are switched between rapidly to create the illusion that they are running simultaneously. The switch from one thread to another is called a **context switch**. Context switches can occur from four possible situations: `thread_yield`, `thread_exit`, block (using `wchan_sleep`), or be preempted. A context switch comprises of three steps,

1. Scheduler decides which thread to run next
2. Save registers of current thread onto its stack
3. Load registers of new thread

In OS/161, step 1 is done by a C function `thread_switch` (which would be called by something like `thread_yield`), which saves the caller-save registers then calls `switchframe_switch` (Assembly) which saves the callee-save registers of the old thread and restores the callee-save registers of the new thread (steps 2 and 3). The switchframe pointer points to the switchframe (top of the stack) and in `switchframe_switch`, register a0 contains the switchframe pointer of the old thread and register a1 contains the one for the new thread. These values for a0 and a1 are passed to `thread_switch` as parameters which push them onto the stack for `switchframe_switch`, since registers a0 to a3 are designated to hold parameters.

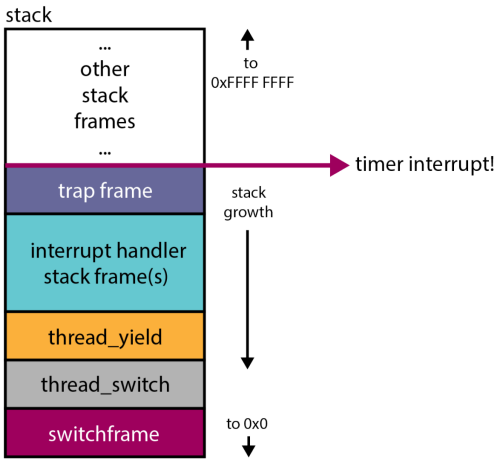
Quantum: An upper bound on the amount of time a thread can execute for. This is needed to achieve timesharing, a quantum is imposed on every thread. When this time finishes, an interrupt is raised to force the thread off the CPU. The running time is tracked through periodic timer interrupts and is not kept track of by the thread itself. Threads may block or yield before the quantum time is up. The quantum resets to 0 every time a thread goes from ready to running but does not reset during an interrupt, since an interrupt "hijacks" the current thread to run the handlers, so technically the current thread is still running.

Interrupt: Caused by system devices (hardware), eg a timer, network interface, etc. When an interrupt occurs, the hardware transfers control to a fixed location in memory (this location is kept in the trap table), where the interrupt handler lives. The handler will,

1. Create a **trap frame** to record thread context at the time of the interrupt. Every register (not only callee-save registers) are saved in the trap frame since the CPU (hardware) had called the interrupt handler, so there was no software caller to save the caller-save registers.
2. Determine which device caused the interrupt and do device-specific things
3. Restore the saved thread context from the trap frame and resume execution

While in the interrupt handler, interrupts are disabled and incoming interrupts are either forgotten about or saved for after the handler finishes, this is hardware dependent. Executing the interrupt handler itself does not involve a context switch. The interrupt handler uses the stack of the current thread to put the trap frame on. In reality, there are 2 stacks, one in the kernel and one for user-space code (read later on for more info on this).

For preemption, when it is discovered that the time is up and an interrupt is raised, the interrupt handler `mips_trap` will call the specific interrupt handler `mainbus_interrupt` which calls `hardclock` which calls `thread_yield` (which then calls `thread_switch`, then `switchframe_switch`, as above). Thus, for preemption, both a trap frame and switch frame are created. However, even though both the trap frame and switch frame save registers, they are saving different contexts. The trap frame contains the context right before the interrupt. The switch frame contains the context up to and including `thread_switch`. As seen in the stack diagram below,



In the above image, when this thread gets scheduled to run again, the top of the stack has the switch frame which records the state of "we had been running an interrupt handler", which gives information on what had happened in the interrupt handlers but this information is **not** in the trap frame. When the thread runs, the values in the switch frame are restored, then `thread_switch`, `thread_yield` finish and pop off the stack, then the interrupt handlers pop off the stack, then the values in the trap frame which record the state before the interrupt are restored and we continue from there. `switchframe_switch` does not explicitly have its own stack section in the diagram above is because it's the same as the `switchframe` part. `switchframe_switch` pushes the callee-save registers of the old thread on the stack, then changes the stack pointer to point to the top of the new stack. When this thread is later scheduled to run again, the registers are restored, and `switchframe_switch` returns.

Another important thing to realize is that out of the four possible ways a context switch can occur (listed above), only one causes the thread to permanently end (`thread_exit`). The others are such that the thread will yield, and eventually run again, because notice that `wchan_sleep` and preemption both involve calls to `thread_switch`, which calls `thread_yield`. Since `thread_yield` then does the switch frame stuff, this means that when a thread goes to sleep, the top of its stack contains (in stack order): the switch frame, `thread_switch`, `thread_yield`, so we can ensure that when a thread goes from ready to running, the top of the stack is **always** the switchframe.

Moreover, if the thread had preempted, then there must have been a timer interrupt, so below the `thread_yield` in the stack must be the interrupt handler stack frames and the trap frame (see image above). This is always the case for preempted threads.

Slides $24 - 41$ of the Threads slides has a great example of what stacks look like during a context switch from preemption.

3 Synchronization

A **critical section** is a region of code which involves access to a global or heap variable, since threads share global/heap data. A **race condition** is when the program result depends on the order of execution of the threads and occur when multiple threads are reading and writing to a piece of shared data at the same time. Race conditions can present themselves differently every time the program is run and might be a seg fault, invalid value, memory leak, etc. Even two threads on different CPUs might have a race condition because if they are trying to write to a piece of memory at the same time, since the memory controller can only do one write at a time, one will have to occur before the other and the order is random. Constants and memory that all threads only read do not cause race conditions. There are three causes of race conditions:

1. Bad implementation, not enforcing proper mutual exclusion on critical sections.
2. Compiler might optimize by caching values in registers but since each thread has its own set of registers, the values will not be synched. Use the volatile keyword to disable this optimization and force the compiler to fetch from RAM every time. Shared variables should be declared with volatile.
3. CPU might re-order loads and stores inappropriately, can use barrier/fence instructions to prevent this

To enforce safety in critical sections, we need mutual exclusion. A naive implementation of this is to use a global boolean,

```
Acquire(bool *lock) {
    while (*lock == true); // spin until lock is free
    *lock = true;          // get the lock
}
Release(bool *lock) {
    *lock = false;
}
```

This doesn't work because `*lock == true` is not atomic and the thread might context switch out while in the middle of this check. What we need to implement synchronization primitives is an atomic (cannot be split) test-and-set. There are hardware specific instructions (eg, `lc`, `sc` in MIPS),

```
MIPSTestAndSet(addr, value) {
    // "load linked", put value at address addr into tmp
    tmp = ll addr
    // "store conditional", store the value into addr if the value
    // at addr has not changed since the ll. Returns success if
    // this is the case, false otherwise (it has changed)
    result = sc addr, value
    if (result == SUCCEED) return tmp
    return TRUE
}
Acquire(bool *lock) {
    while (MIPSTestAndSet(lock, true) == true) {};
}
Release(bool *lock) {
    *lock = false;
}
```

Thus, `MIPSTestAndSet` will return false iff the original value in `addr` is false, and it does not change at the `sc` (and true is loaded into `addr` to represent the lock being newly acquired). Otherwise, the while loop will keep spinning and the lock has not yet been acquired.

3.1 Spinlock

A spinlock performs busy waiting: it keeps spinning in a loop, repeatedly testing lock availability, until the lock is available. The CPU is being actively used by spinlocks. Thus, they are not a general-purpose lock and although might be faster with small critical sections than a lock which has overhead associated with wait channels and is also not ready for use early in the boot process, generally you do not want to use spinlocks over locks, especially for large critical sections. The implementation in OS/161 is very similar to written above (uses `ll, sc` instructions and an unsigned "data" to represent whether the lock is currently held or not). Interrupts are disabled right at the start in `spinlock_acquire` and re-enabled at the end of `spinlock_release` to minimize spinning, this means there can be no preemption so it is efficient for very large critical sections. Since this is the case, the running thread which owns the spinlock effectively "owns" the CPU, so basically the owner of a spinlock is a CPU, rather than a thread. However, this thread can still yield, exit, or block. Interrupts are disabled while a spinlock is held because if it is preempted (requires an interrupt), then if the new thread which is running wants the spinlock, it will be pointlessly spinning since the thread which owns it is not able to give it up. Since interrupts are disabled, there is effectively no spinning (assuming a single CPU system).

3.2 Lock

Whereas spinlocks spin, a lock blocks. This means that a thread calling `lock_acquire` will block (go to sleep on a wait channel) until it can acquire the lock. This is more efficient than spinning. A lock is owned by a thread and interrupts (and thus, preemption) is not disabled while a lock is owned. A lock also makes use of a spinlock because the lock functions themselves are critical sections, and a wait channel. Besides `lock_create`, `lock_destroy`, `lock_do_i_hold`, the implementations of acquire and release are,

```
void lock_acquire(struct lock *l) {
    spinlock_acquire(l->spinlock);
    // need a while loop in case another thread acquires after we wake up
    while (l->owner != NULL) {
        // do this before releasing spin to prevent the owner from
        // releasing the lock while you're in here
        wchan_lock(l->wc);
        // always release spinlocks before going to sleep, or else
        // another thread will spin indefinitely trying to acquire it
        spinlock_release(l->spinlock);
        // this is why wchan_sleep cannot lock itself (there is a
        // line of code above separating the two function calls)
        wchan_sleep(l->wc);
        spinlock_acquire(l->spinlock);
    }
    // we can now take the lock
    l->owner = curthread;
    spinlock_release(l->spinlock);
}
void lock_release(struct lock *l) {
    spinlock_acquire(l->spinlock);
    l->owner = NULL;
    // only wake one because lock only has one owner anyway
    wchan_wakeone(l->wc);
    spinlock_release(l->spinlock);
}
```

The key things with writing this kind of code is that the synchronization primitives should ensure that

invariants are held. For example, in the code in `lock_acquire` in the while loop after we check that the lock is owned, we do not want it to be possible for the lock to suddenly be released because then we would go to sleep on a lock that is not owned by anyone. Also need assertions to check that the lock is not NULL, that you are not trying to acquire a lock you already own (will result in the thread going to sleep on the wait channel forever), and that you are not releasing a lock you do not own (this would be weird). Also, locks should not have an owner when `lock_destroy` is called. It is the responsibility of the programmer to not have a possible context switch in `lock_destroy` to another thread trying to use a partially destroyed lock.

3.3 Wait channels

Sometimes a thread might need to wait for something, eg a lock to be released or a key to be pressed. In this case, it will block and the scheduler will choose a new thread to run and there will be a context switch. The blocked thread is put on a wait channel (queue). Calling `wchan_sleep(wc)` will put the thread on the wait channel `wc`. Using `wchan_wakeone/wakeall(wc)` will wake either one or all threads blocked on `wc`. Waking means to put the thread from blocked to ready. Each thread can only be asleep on a single wait channel, because once it goes to sleep on the first one, it is in a blocked state, so cannot call `wchan_sleep` on a different wait channel. Also, adding and removing from wait channels are critical sections. All wait channel functions such as `wchan_wakeone` will lock and unlock the spinlock associated with `wc` for you. But `wchan_sleep` cannot lock itself, so the programmer must explicitly `wchan_lock(wc)` first. The lock will later be unlocked within `wchan_sleep`. See the implementation of `lock_acquire` to see why `wchan_sleep` cannot unlock itself. Waking a channel with no threads sleeping has no effect.

3.4 Semaphore

A semaphore has an integer count value and supports two operations: P, which will decrement the count if it is above 0, otherwise wait until it is above 0 then decrement. And V which will increment the count. These are both atomic operations because a semaphore uses a spinlock/lock. This count represents the number of "resources" and more than one thread can take a resource, unlike locks. The value of the count itself should never be accessed by user programs. Compared to locks, the count can start off at 0, V can come before P, and semaphores do not have owners (can be P'd, V'd by any thread). There are three major types:

1. Binary semaphore, has a count of 1 and behaves like a lock (but with no owner)
2. Counting semaphore, has an arbitrary number of resources
3. Barrier semaphore, used to force one thread to wait for others to complete, initial count is typically 0, can be used like thread join

The implementation of P is very similar to `lock_acquire` (threads will be put to sleep if trying to P but count is 0) and the implementation of V is very similar to `lock_release`.

Producer/consumer problem: there is a buffer with capacity N and consumers cannot take if the buffer is empty and produces cannot produce if the buffer is full. We can use three semaphores to synchronize this problem: one which starts at N and represents the number of remaining spaces, one which starts at 0 and represents the number of produced items, and a binary semaphore to protect the buffer itself. Since we cannot access a semaphore's count, we need two semaphores for each the number of spaces and number of items.

3.5 Condition variable

Suppose we wanted an arbitrary condition to be true in a critical section. A naive implementation might be,

```
lock_acquire(l);
while (num == 1) {
    lock_release(l);
    // release to give another thread a chance to change num, we
```

```
    // might thread_yield here as well to give a greater chance
    lock_acquire(l);
}
```

But this is bad because it involves busy spinning. Instead, use a CV.

```
lock_acquire(l);
while (num == 1) {
    // pass in the lock that you own
    cv_wait(cv, l);
}
```

And in another thread, it might look like this,

```
lock_acquire(l);
num = 2;
if (num != 1) {
    cv_signal(cv, l);
}
lock_release(l);
```

CVs are intended to be used with a lock, within a critical section which is protected by the lock. Contains a wait channel which threads waiting for the condition block on. The lock is re-acquired before the end of `cv_wait` so from the perspective of the calling function, ownership has never changed. The three important operations are,

```
void cv_wait(struct cv *cv, struct lock *l) {
    // at this point, we already own the lock
    wchan_lock(cv->wchan);
    // release l after locking wchan for the same reason as in lock_acquire
    lock_release(l);
    wchan_sleep(cv->wchan);
    // after waking up, acquire the lock again, if the lock is
    // taken, we might go to sleep again, this time on the lock's wchan
    lock_acquire(l);
}
// also cv_broadcast, which calls wchan_wakeall
void cv_signal(struct cv *cv, struct lock *l) {
    wchan_wakeone(cv->wchan);
    // mesa style cvs don't use the lock here, but it is needed for
    // wait morphing (pass a thread blocked on the cv's wchan onto
    // the lock's wchan, this can improve performance)
}
```

Also, remember the common assertions (lock is not null, cv is not null, the lock is owned at the start of all these functions). The producer/consumer surplus can now also be solved with a lock and two CVs, one which producer threads sleep on while the buffer is full and one which consumer threads sleep on while the buffer is empty.

A CV is like a user-level wait channel and is roughly analogous to how a lock is a higher level spinlock. The high level feature here is that a CV handles the acquire and release of the associated lock. You could do this with a wait channel but it's more difficult.

As well, a CV is more general than a semaphore and anywhere you could use a semaphore, you could use a CV. But it is generally more complicated to use. Since semaphores abstract the idea of getting resources, they are a more natural fit for the producer/consumer problem.

3.6 Deadlock

Deadlock occurs when two threads prevent each other from accessing some resource, so progress halts. An example is a thread trying to acquire a lock it already owns (it will block but nobody can wake it up), or two threads which mutually want a lock that another owns. They will both block on the wait channels of each other's locks. Two strategies to fix this are:

No hold and wait: Get every resource at once. If you cannot, release what you have so far and try again. Needs a special function,

```
// returns true iff acquired
bool try_acquire(struct lock *l) {
    spinlock_acquire(l->spinlock);
    if (l->held) {
        spinlock_release(l->spinlock);
        return false;
    }
    l->owner = curthread;
    l->held = true;
    spinlock_release(l->spinlock);
    return true;
}
```

To use this,

```
lock_acquire(lock1);
// busy spin until we get both lock1 AND lock2
while (!try_acquire(lock2) {
    lock_release(lock1);
    // optional thread_yield here to give more time
    lock_acquire(lock1);
}
```

Resource ordering: Does not require a special implementation like the first strategy nor does it require busy spinning. Instead, assign each resource a number and acquire only in increasing order (or only in decreasing order). For example, if holding resource 5, cannot try to acquire resource 3. Is prone to programmer errors if they get the order wrong, needs documentation.

4 Processes and the Kernel

A process is an environment in which an application program runs. It includes virtualized resources that its program can use, such as: one or more threads which execute instructions, virtual memory for code and data, and other resources like file descriptors. Processes are created and managed by the kernel.

4.1 Interface (in OS/161)

These were implemented in Assignments A2a, A2b,

1. `fork()`: Creates a child process that is separate but identical to the parent process, returns 0 in the child process and the child PID in the parent process. Both processes return from `fork` because the PC is the same in both.
2. `_exit()`: Terminates the process which calls it, supplies an exit status code in case a parent calls `waitpid` on it.
3. `getpid()`: Returns the PID of the process which calls it. Note that PID can never be 0 and must be unique to all existing processes.
4. `waitpid(child_pid, &child_exit, 0)`: Process waits for another to terminate and retrieves its exit status. Processes can only `waitpid` on their immediate children. This function provides synchronization between processes. The thread in the parent process which called this function will sleep on the child's condition variable. Then, when a thread in the child process calls `_exit`, it will wake up the parent thread. Thus, `_exit` and `waitpid` work together.
5. `execv("program_location", args)`: Changes the program that the process is running by destroying the current virtual memory of the process and giving it a new virtual memory initialized with the code and data of the new program to run. PID remains the same. All code after the `execv` in the original program will never be run. Normally used after `fork` to run a new program. Might fail if not enough memory or program name was spelt incorrectly, etc. Parent-child relationship still holds after `execv`.

4.2 System calls

Privilege: The CPU implements different levels of execution privilege for security and isolation. Kernel code runs at the highest privilege and application code runs at a lower privilege level to prevent user programs from doing things such as halting the CPU, modifying page tables. Thus, application code cannot directly call kernel functions or access kernel data structures. The CPU will throw an exception if unprivileged code tries to do something that requires privilege. Unprivileged code, such as functions like `fork`, which are called by user programs are in the **system call library**. There are two ways to get kernel code to run,

1. There is an interrupt (generated by hardware). Interrupt handlers are part of the kernel, so the CPU switches to privileged mode when it transfers control to the interrupt handler.
2. There is an exception (generated by instruction execution/software). For example, dividing by 0. Similar to interrupts, the CPU will transfer control to the exception handler, which is located in a fixed location in memory and the CPU switches to privileged mode. The exception handler is part of the kernel. In MIPS, everything is an exception, and an interrupt is a type of exception. System calls are also a type of exception.

System calls: The interface between processes and the kernel. Very expensive (slow). Anything which messes with the kernel, such as thread forking, thread blocking, etc are system calls. To perform a system call, the application program will raise an exception of type `EX_SYS` using the `syscall` instruction. The exception handler then checks the type of exception call and sees that it is a system call. To specify the type of system call, the application places a code in register `v0`, like `li v0, 0`. Here, 0 is the code for `fork`, each system call gets its own code (eg, `#define SYS_waitpid 4`). The exception handler checks this type then

calls the appropriate system call function. These codes are part of the Application Binary Interface (ABI) and are known to both applications and the kernel. The ABI also has error codes. To pass parameters and get return values from system calls, things are put into pre-determined registers. For example, parameters go in registers `a0, a1, a2, a3` (or go onto the heap if more than 4 parameters are needed, then the heap address is put into one of these registers). The return code (0 if the result is a success, 1 if fail) goes into register `a3` and the return value (either an error code, or a result) goes into `v0`. The exact steps are,

1. Application calls system call library wrapper function (eg `fork`)
2. Library function performs `syscall` instruction and prepares parameters such as `v0, a0 - a3`
3. Go into privileged mode and the kernel exception handler runs
   (a) Trap frame is created to save state
   (b) Determine that this is a system call exception and determine which kind of system call
   (c) Does the work for the system call
   (d) Restores state from trap frame and returns
4. Library function finishes and returns, application continues execution. Note that in the above, the trap frame was created but no switch frame because there was no context switching done.

Each process thread actually has **two stacks**: the user/application stack is used while application code is executing and is contained in the process's virtual memory. This stack is made by the kernel when the virtual address memory is set up for the process. The second stack is the kernel stack which is used while executing kernel code (after an exception or interrupt). In OS/161, every thread has a pointer to this kernel stack but if this pointer is dereferenced in a user program, there will be some sort of permission exception/error. Trap frames and switch frames are held in the kernel stack, the system call library functions in user-space (unprivileged) are held in the user stack. The reason two stacks are needed is for safety (don't want user code to access kernel data, which would be possible if it was all on the same stack) and if the user stack is almost full, the kernel should not overflow the user's stack. The kernel can access user stacks but users cannot access the kernel stack. The kernel stacks for threads are allocated using `kmalloc`, which puts it in the kernel's memory and is not user-addressable.

The steps for exception handling in OS/161 are,

1. CPU turns off interrupts for us
2. Assembly code at a pre-determined location is run, and it calls the Assembly code `common_exception`
3. Go into privileged mode, then `common_exception` saves the application stack pointer, then switches the stack pointer to the thread's kernel stack. Then, saves the application state (registers) and the address of the instruction that was interrupted (PC) in a trap frame on the kernel stack. Then, calls `mips_trap`, passes a pointer to the trap frame as a parameter.
4. `mips_trap` will,
   (a) Determine what type of exception this is by looking at the exception code
   (b) Call the separate handler for each type (eg `mainbus_interrupt` for interrupts, `syscall` for system calls, `vm_fault` for address translation exception)
   (c) Remember that when the exception first came in, interrupts were disabled. So when `mips_trap` was called, interrupts are still disabled. If the exception type is an interrupt, then `mips_trap` does not re-enable interrupts but for every other kind of exception, (eg a system call) it does. And it does this before calling the handlers (eg `syscall`). After the handler returns, interrupts are once again turned off before returning from `mips_trap`. Therefore, it is not possible to have two trap frames back to back in the kernel stack because interrupts are disabled when the trap frames are

being created and not re-enabled until halfway through `mips_trap`, so at the very least every two trap frames are separated by a `mips_trap`.

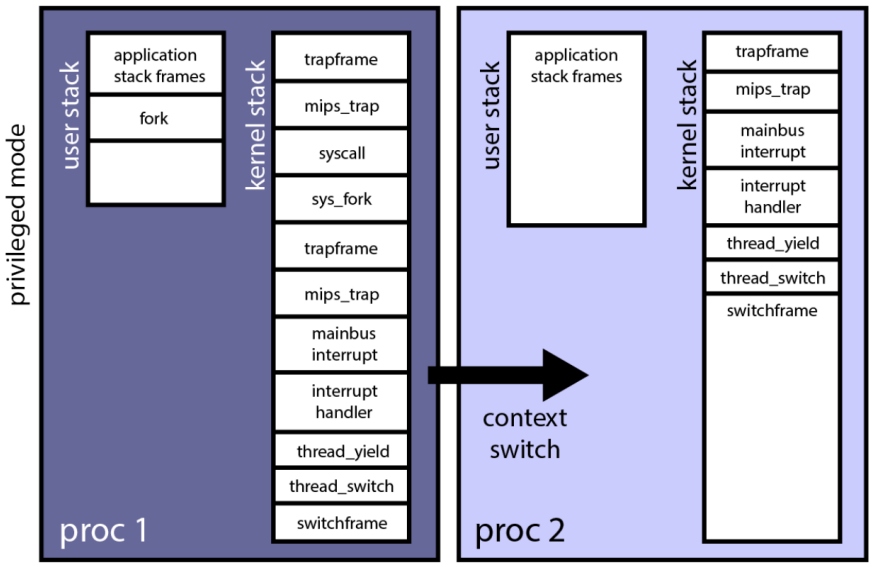
5. After the exception handler and `mips_trap` finish, the application state is restored from the trap frame (including the stack pointer), and the `rfe` instruction is used to switch back from privileged to unprivileged mode.
6. Go to `mips_usermode` (which turns interrupts back on), jump back to the application instruction that was interrupted and switch back to unprivileged execution mode

Multi-processing: All processes share the available hardware resources (eg physical memory, CPUs, I/O devices), the OS coordinates the sharing. Each process needs at least one thread to execute.

Inter-process communication: The OS separates process from one another but they can still communicate using IPC. Some examples include: sockets (sending data via network interface between processes on different computers), pipe, shared memory (two processes writing and reading from the same piece of memory, requires synchronization and is not particularly flexible or safe), message passing, file.

The example from slide 25 onwards is very useful. Key takeaways:

1. The timer interrupt repeatedly fires on a thread until it is seen that the quantum has expired, in which case the typical chain of events starting from `thread_yield` occurs and leads to a context switch.
2. Since `mips_trap` turns interrupts back on before calling `syscall` (which then calls `sys_fork` for example), it is very possible that the timer interrupt comes while we are in `sys_fork`. All these things go on the kernel stack.
3. An example of one part of this, after context switching when there is a timer interrupt in `sys_fork` and the quantum had run out,



4.3 Implementations in OS/161

These are the important functions to understand. They return either 0 for success or a non-zero value (eg `ECHILD`) on error then `syscall` packages this return value and the "actual" return value passed by pointer as `retval` into the right registers.

1. `syscall(*trapframe)`: Called by `mips_trap`, is given a pointer to the trapframe, calls the appropriate handler depending on the type of system call. Then, sets up the return value and error code and result from whichever specific function it called (eg `sys_fork`) in the trap frame, and increments PC (to prevent repeating the `syscall` instruction endlessly).

2. `sys_fork(*retval, *trapframe)`: Creates process structure for child process, creates and copies and attaches the address space and data into the new process, assigns the PID and creates the parent-child relationship (using a parent pointer and child array), creates a thread for the child process. Now, note the trap frame the pointer points to is on the parent thread's kernel stack. But the child thread also needs a trap frame with very similar values. Copy the trap frame onto the heap in case the parent's trap frame gets popped first because then the child's pointer would be garbage. Next, the child thread puts the trap frame onto its (kernel) stack by simply allocating a stack variable, and modifies it so that it returns the right value (should return 0 rather than the child PID which the parent thread returns). The child thread then calls `mips_usermode` to go back to userspace. This will set the stack pointer to point to the trap frame on the child thread's kernel stack, then the values in the trap frame are restored, etc. So basically `mips_usermode` is equivalent from finishing with `mips_trap`. We need to "fake" this return because `mips_trap` was never called in the child thread since it's brand new and the kernel stack only contains the trap frame (side note: the user stack is an identical copy to the parent's user stack).

3. `sys__exit(exitcode)`: Cleans up the address space, detaches the current thread from the process. If an orphan, go through the children processes and delete zombies, or sets the parent pointer to `NULL` on non-zombies. Otherwise, mark it as a zombie, record the exitcode it was given (recall the user-space syscall library API is `_exit(code)`), and why it exited (`__WEXITED`), and signal its CV (in case the parent was sleeping on it and waiting for it to exit). Finally, `thread_exit`. Doesn't return anything because `thread_exit` should stop it.

4. `sys_waitpid(pid, status, options, *retval)`: Find the child process that corresponds to the PID it was given. Block (on the child's CV) until the child exits (becomes a zombie). Then retrieve the error code and status and delete the child, since we can only `waitpid` on a child once. Copy this status out from kernel-space to user-space using `copyout`, then set `retval` to the child's PID that it was given, or $0/-1$ on error.

5. `sys_execv(userptr_t progname, userptr_t args)`: Copy the program name from user space into kernel space. This is needed because the address space of the old process will be deleted so the kernel memory is used as an intermediate buffer. Also, the pointer we are given might be malicious so need to verify with `copyinstr`. Similarly, copy the arguments into kernel memory. Everything after this is almost identical to `runprogram`. Open the file associated with the program name, create a new address space, switch to it, load the executable (which was opened earlier), close the file, then call `as_define_stack` and give it the copied arguments, stack pointer (which is now for the new address space). This function will push the arguments onto the userstack (and thus push down the stack pointer), taking into consideration that the stack pointer must end up a multiple of 8 and pointers must end up a multiple of 4. Uses `ROUNDUP(a,b)` to deal with this issue. Then, destroy the old address space (we saved a pointer to it earlier since `curproc_setas` returns the old as) then call `enter_new_process`, which goes to user mode and sets up the trap frame. This function is very similar to `enter_forked_process`. Destroy the old address space as late as possible in case there's an error with some other step, we need to return the error code to the address space of the process which called `execv`. The reason `runprogram` does not need to delete the old address space is because there is no old address space, `runprogram` is creating a new one.t

A process can be fully deleted in three situations: `sys__exit` was called and it is an orphan (no living parent), after calling `waitpid` on it (because you can only `waitpid` on a child process once), or it is exiting and has dead children then the dead children can be deleted. Overall, a process can be deleted iff `waitpid` will not be called on it in the future. A zombie process is one which exited (with `_exit`) but cannot be deleted yet because its parent is still alive and the parent needs access to its exit status code.

Also, the address space only contains the code, stack, and heap. The thread and process structures themselves are stored in the kernel's section of memory (since they were allocated using `kmalloc`).

If you wanted to re-use PIDs, then re-use PIDs for orphan processes. This is because the PID is important for `waitpid` but orphan processes cannot be `waitpid`'d by anyone. Re-using a PID that is not an orphan will cause issues if the parent is still alive and spawns another child which re-uses this PID, then it will have two children with the same PID. We can also re-use a PID after calling `waitpid` with it, since `waitpid` is called one each process at most once.

The reason `mips_trap` re-enables interrupts for system calls (among other cases) is because system calls are typically quite slow and you don't want to lose too many interrupts while you are dealing with them. On the other hand, it does not re-enable interrupts for actual (hardware) interrupts because these are not generated by software on purpose and should be dealt with as soon as possible.

5 Virtual Memory

If physical addresses are P bits, then the max amount of **physical memory** addressable is 2^P bytes. The actual amount of physical memory on a machine may be less than the max amount that can be addressed. For security and encapsulation/isolation, the kernel provides each process with separate **virtual memory** (which holds the code, data, and stack). Similarly, if virtual addresses are V bits, then the max amount of virtual memory addressable is 2^V bytes. Running applications only see virtual addresses: PC, stack pointer, variable pointers, jump/branches all use virtual addresses. Virtual addresses allow us to run more processes than you have physical memory to support (good for multi-processing), meaning a process can have a larger virtual address space than the amount of actual physical memory it has (eg, 16 bits to address physical memory, 18 bits to address virtual). Processes are only aware of virtual memory.

Fragmentation occurs when memory is used inefficiently and there is potentially enough space for a program but cannot be allocated due to its placement. Internal fragmentation is when a program allocates a large chunk of memory, but within that chunk, a lot is unused. Occurs in dynamic relocation, as seen later. External fragmentation is when the memory needed does not exist in contiguous blocks and occurs when allocation is in different sizes and occurs in segmentation but not paging, as seen later.

Memory Management Unit (MMU): A piece of hardware that is part of the CPU and does the translation from virtual memory (which the process sees) to physical memory. Translation is done in hardware because it is much faster and every instruction needs at least one translation, since the PC itself is a virtual address. Also checks for and raises exceptions when the translation cannot be done.

5.1 Dynamic Relocation

The MMU will have two registers: one is the offset (relocation), R, which is the position in physical memory where the process's memory begins, and the other is the limit, L, which is the amount of memory used by the process. Thus, for a virtual address v, first the MMU checks that $v < L$, then the physical address is $p = v + R$. If $v \geq L$, then raise an exception. These L, R registers need to be updated when context switching to another process's thread. Though dynamic relocation is efficient overall since it only involves simple addition, it will result in (internal) fragmentation since if memory is very sparsely laid out in the virtual address space, a large block needs to be allocated to contain everything.

5.2 Segmentation

Instead of mapping the entire virtual address space to one big block of physical memory, separate the virtual memory into segments. For example, one segment for the code, another for data, another for stack. The kernel maintains an offset and limit for each segment (then does the same calculation for each segment as in dynamic relocation). A virtual address now can be thought of as a segment ID and an offset within a segment. If there are K bits for the segment ID, V bits for the virtual address, then there are up to 2^K segments and 2^{V-K} bytes per segment. The segment bits are always the highest K bits of the address and the offset is the rest. Be very careful about decomposing hex digits into binary first. The term "segmentation fault" comes from this strategy.

One approach to this is for the MMU to have 2 registers per segment (one for R_i and one for L_i, then do the same calculation as above, where the segment ID is extracted, then the offset within the segment is compared to the limit of the segment, etc). However, this will take up a lot of registers.

The second approach is to have a segment table (in RAM). Have one register which is the base address (physical address) of the segment table, and the other register contains the length of the segment table (ie, the number of segments). If the segment number given is greater than the number of segments, throw an exception. Otherwise, use this number to lookup the limit and relocation values from the segment table. Another advantage to this is you can store other info such as whether a piece of memory is read-only in the segment table for each segment. Overall, this strategy only requires 2 registers.

Segmentation has no internal fragmentation but still has external fragmentation.

For a context switch to threads of the same process, need to update the relocation and limit of the stack but not the other segments. For a context switch to threads of a different process, need to update all relocation and limits. The process structure itself contains info for relocation and limits.

5.3 Paging

Recall there are 8 bits in a byte. There are 2^{10} bytes in a KB.

Physical memory is split into **frames** (commonly 4KB). Virtual memory is split into **pages** of the same size. Then, each page maps to a different frame, any page can map to any page, and the mapping is done using a **page table**. Frames don't necessarily have to be stored in RAM, they could also be on disk or CD drive. Thus, paging allows us to have more processes running than can physically support in RAM by putting extra pages on disk, which will be fetched when needed through on-demand paging.

Paging does not eliminate internal fragmentation but does eliminate external fragmentation because all pages/frames are the same size and external fragmentation comes from allocating physical memory of different sizes.

5.4 Single level paging

Each row in the page table is a page table entry (PTE, the size of this will always be given on examinations). The table is indexed by page number. Since not all pages of virtual memory will be used by the address space, there is also a valid bit in each PTE. The number of PTEs in the table is equal to the max size of virtual memory divided by the page size. The number of PTEs in the table is equal to the number of pages needed to cover all the virtual memory.

To translate addresses, the MMU has a page table base register which points to the page table for the current process (each process has its own page table). The page number and offset are determined from the virtual address (page number is higher bits, offset is lower bits), then look up the PTE in the page table. If the valid bit is off, then raise an exception. Otherwise, combine the frame number in the PTE with the offset to get the physical address. Since the page size matches the frame size, the offsets will work. No limit comparisons needed. PTEs also contain other info such as the write protection bit (if the page is read-only), page usage bit/dirty bit. These usage bits are set by the MMU and read by the kernel.

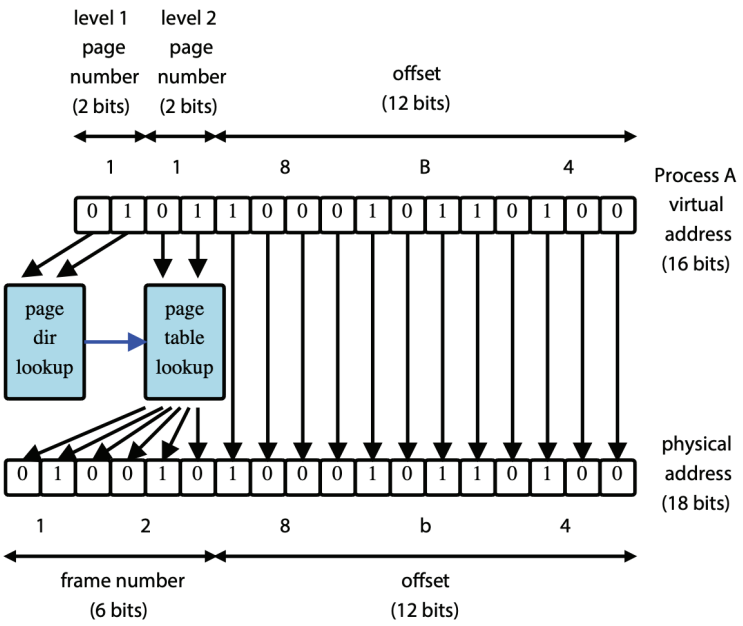
The number of bits for the offset is the logarithm (base 2) of the page size. The number of bits for the page number is the logarithm of the number of PTEs in the page table. Be careful, if the number of bits for the page number is not a multiple of 4, you need to decompose the hex digits first. Since one PTE corresponds to one page, the page table size is the number of pages in it multiplied by the PTE size.

Page tables can get very large. If $V = 48$ bits (thus, max virtual memory size is 2^{48} bytes, and the page size is 4 KB, PTE size is 4 bytes, then the page table size is 256 GB. Page tables are kernel data structures, so live in kernel memory (in physical memory, so itself is not virtual addressed).

5.5 Multi-level paging

Split the page table into multiple levels in a tree-like structure (there is always one root, called the directory). The table is no longer contiguous (in kernel memory), it is replaced by smaller tables which each fit into a single page themselves (for efficiency). This saves space because if a table contains no valid PTEs, it does not need to be constructed. All leaf nodes in this tree contain the frame number, the non-leaf nodes contain pointers to the children (the next table to look at).

Each entry in each sub-table in this tree will also have valid bits, as before. The look-up scheme now looks like,



The MMU's page table base register points to the page table directory for the current process. Besides the offset bits, the remaining are split into parts $p_1, \ldots, p_n$, where p_1 is used for lookup in the directory to find the second-level page table, then p_2 is used, etc., until p_n is used to find the PTE containing the frame number at the leaf level. At each step, if the valid bit is off, raise an exception. In the end, the frame number combined with the offset gives the physical address.

To calculate the number of levels needed, let o be the number of offset bits in the address (recall $o = \lceil \log p \rceil$, where p is the page size. Suppose there are V bits in the virtual address. We need to break down these $V - o$ bits to represent the different levels. Also, we know we need $\lceil 2^V / p \rceil$ pages. Suppose the PTE size is x. Then, p/x PTEs can fit on a single page. Since we want each page table to itself fit on a single page (this is very important!!), this means we need $\lceil \log(p/x) \rceil$ bits to represent the row in the page table. The number of PTEs is equal to the number of pages, so we need $\lceil 2^V / p \rceil$ PTEs. Therefore, we want to split up the $V - o$ bits into sections of $\lceil \log(p/x) \rceil$. So overall, the number of levels needed is $\lceil (V-o) / \lceil \log(p/x) \rceil \rceil$.

In the tree, each valid entry at a non leaf level points to another table below it. Thus, the number of tables at the next level is equal to the number of valid entries in total at the previous level across all the page tables at that previous level. Since this tree takes up memory as well, multi-level paging might actually use more memory than single-level if every (or almost every) virtual address is being used, but this is rare.

In summary, the kernel manages the MMU registers on address space switches, creates and manages page tables, manages physical memory, and handles exceptions raised by the MMU. The MMU is on the CPU and translates virtual to physical and raises exceptions when necessary.

5.6 Translation lookaside buffer (TLB)

RAM is connected to the CPU through an address bus, which is slow to travel across. So, since each instruction requires at least one page table lookup (to translate the address of the instruction itself), it is much better to use a cache of address translations, which is the TLB (located in the MMU which is on the CPU). Each TLB entry stores a page number to frame number mapping. If on a lookup, there is nothing in the TLB, then go to memory then put that translation in the TLB and kick something old out if necessary. TLB entries are different for different address spaces so if the MMU cannot differentiate this then the kernel must clear the TLB on each context switch from one process to another (thus different address space). TLB misses are more expensive the deeper the multi-level page table goes since a single miss requires up to h

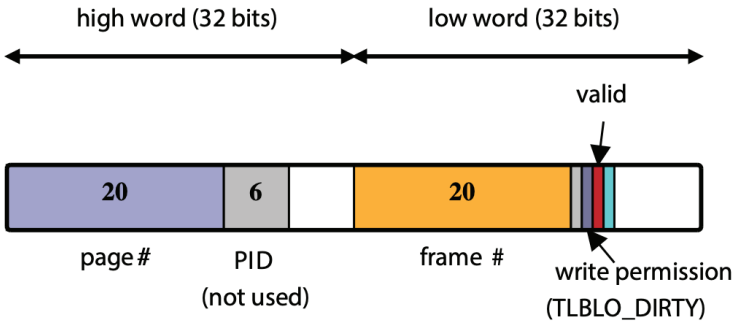
memory accesses, where h is the height of the tree, one for each intermediate page table and one for the final address translation.

Hardware-managed TLB: the MMU handles TLB misses by looking up the page table in the kernel (MMU must know the kernel's page table format to do this) and replaces TLB entries. Faster than software-managed but requires coordination between hardware and kernel programmers to format the page table a certain way.

Software-managed TLB: the TLB raises an exception on a miss (the needed translation is not in the TLB) and the kernel determines the frame number for the page then adds the mapping to the TLB, evicting another entry if needed. Then, after the miss is handled, the instruction that caused the exception is re-tried, except the TLB now has the required translation. The kernel decides how/which data structure to record page to frame mappings.

5.7 Implementation in OS/161, MIPS (dumbvm)

The MIPS TLB has space for 64 entries, each 64 bits in size. Pages are 4 KB. Each entry is laid out like,



MIPS uses 32-bit paged virtual and physical addresses and has a software managed TLB. `vm_fault` is the handler for TLB misses, it uses info from the `addrspace` structure to determine the correct frame mapping given the page number.

In the naive implementation, OS/161 places all pages of each segment contiguously in physical memory, so `addrspace` structure contains fields such as the base address (both physical and virtual) of the code segment and data segment and how big each segment is in pages, as well as the base physical address of the stack. Since stacks are a fixed size (12 pages) and start in a fixed location in virtual memory (0x8000 0000), the structure does not need to keep track of these things. Specifically, the top of the stack is 0x8000 0000, and the base is $0\text{x}8000\ 0000 - 12 \cdot 4096$. `vm_fault` takes in a fault address (the virtual address that the TLB does not have a translation for) and since memory is segmented into data, code, and stack, we can see which segment the address belongs to based on the base and sizes recorded in the `addrspace` structure. Then, find the offset by subtracting the segment base from the fault address and add the physical base to get the physical address. Return `EFAULT` (is an exception) if the address is not in any segment. However, this introduces external fragmentation.

In the paged implementation, pages of a segment do not necessarily appear contiguously in physical memory. Instead of the physical base fields in the `addrspace` structure, each segment has a corresponding pointer to a page table, which maps page numbers to frame numbers (one page table per segment). `vm_fault` uses the page table to find the frame number, and since the base address of physical memory is known, the physical address the page corresponds to can be calculated, by adding in the offset in the same way as before. Since the segments are a fixed size, we know how many pages there are, so the page table is exactly big enough, which is quite efficient and no valid bit is needed. The pages for the three segments are pre-allocated in `as_prepare_load` one page at a time and de-allocated in `as_destroy`. Since contiguous allocations can now be scattered anywhere in physical memory, instead of allocating several pages at once, we allocate one

page at a time, so the page table always has exactly the page to frame mappings. The logic from the naive implementation regarding traversing the coremap to deal with contiguous allocation is no longer needed. In `as_copy`, after allocating the new address space segments, we copy the data frame by frame, since the new frames may be in a different order than the old frames and all we can guarantee is that within a frame, the data is identical.

Regarding the coremap, during the bootstrap process, the kernel allocated memory using `ram_stealmem` to get enough space for kernel data structures. After this, the remaining physical memory is logically partitioned into fixed size frames, each 4 KB in size. The status of each frame (whether it is allocated or not) is tracked in the coremap. The coremap does not track its own memory, since the coremap lives for as long as the OS does.

When the kernel creates a process for a program, it must create the address space and also load the program's code and data into that address space. OS/161 pre-loads the address space before the program runs whereas many other OS load pages on demand (since programs can be quite large). A program's code and data is described in an **executable file**. In OS/161, this file must be in the **Executable and Linking Format (ELF)** format. So, the program parameter given to `execv` was actually the name of the ELF executable file for the program that is to be loaded into the address space.

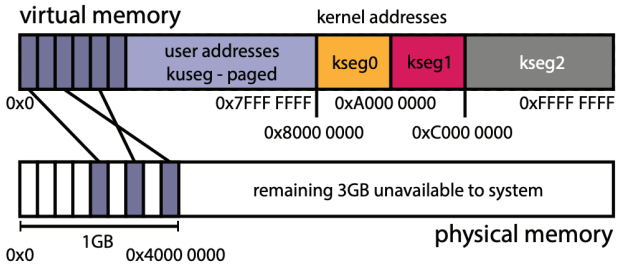
ELF files: contain address space segment descriptions. The ELF header describes the segment images (exact copies of the binary data to be stored in the address space): the virtual address of the start of the segment, length of the segment in the virtual address space, location and length of the segment in the ELF. The ELF file also identifies the virtual address of the program's first instruction (the **entry point**) and other useful information such as whether a segment is read only, symbol tables for compilers, linkers, etc. In OS/161, the ELF file contains two segments: a text segment (program data and read-only data) and a data segment (any other global program data) but does not describe the stack, since it is standard (12 pages long, ends at 0x7FFFFFFF for all processes) and controlled by the OS and not the compiler. If the image in the ELF is smaller than the segment it is loaded into in the addres space, the rest of the address space segment is zero-filled. `load_elf` loads the ELF file given. Note that addresses from read-only segments (such as the data segment) cannot have the dirty bit off in the TLB at this stage because the OS is trying to load data in so must be able to write. In A3, this problem can be fixed using a global flag that keeps track of whether `load_elf` has completed and after `load_elf` finishes, the TLB is cleared so that these dirty bit on entries can be flushed and replaced with dirty bit off entries in later use.

The kernel itself lives in virtual memory but there are two problems,

1. Bootstrapping: since the kernel implements virtual memory, how can it run in virtual memory when the kernel is just starting up? The solution to this is architecture-specific.
2. Sharing: how can data be copied between the kernel and user programs if they are in different address spaces? The solution to this is having the kernel's virtual memory overlap with the process' virtual memory. That is, for every process, virtual addresses 0 to 0x7FFFFFFF are user addresses and 0x80000000 to 0xFFFFFFFF are kernel addresses. This way, it looks like the kernel is in the top half of every virtual address space. However, it technically is not there. Kernel memory is its own thing.

In OS/161, there is 4 GB of physical memory, of which 3 GB is unavailable. The lowest 1 GB from 0x0 to 0x4000 0000 has everything and is divided into frames. These frames are managed by the coremap. User virtual memory, `kuseg` is paged and the kernel maintains the page-to-frame mappings for each process and the TLB (translation cache) is used to translate these addresses to physical ones.

The kernel's virtual memory is divided into three segments: `kseg0, kseg1, kseg2`,



`kseg0` (size 512 MB, ranges from 0x8000 0000 to 0x9FFF FFFF) is for the kernel's data structures, stack, and code. To translate these addresses to physical ones, subtract 0x8000 0000. `kseg1` (size 512 MB, ranges from 0xA000 0000 to 0xBFFF FFFF) is for accessing devices so if an address is in this segment, we are trying to talk to a device. To translate these addresses to physical ones, subtract 0xA0000000. `kseg2` is not used and the MMU will throw a bus error if it sees an address in this range (from 0xC000 0000 to 0xFFFF FFFF). Both `kseg0, kseg1` are not paged nor are addresses in these ranges translated by the TLB. Instead, the MMU does dynamic relocation using 0x8000 0000 and 0xA000 0000, respectively. Also, both map to the first 512 MB of physical memory but there is no collision because they generally do not use the ranges of addresses. Leftovers from this first 512 MB may be used by user programs if necessary.

5.8 Secondary storage, page faults

To allow for virtual address spaces that are larger than the physical address space (and also allow greater multiprogramming levels by using less of the available primary memory for each process), we store virtual memory pages in secondary storage, such as disks, and swap pages between secondary storage and main/physical memory (RAM) so that they are in RAM when needed.

In order for swapping to work, some pages of virtual memory will be in physical memory (the set of such pages is called the resident set of a process) and others will not be in memory and will be in secondary storage. As pages are swapped in and out, the resident set will change. To track which pages are in physical memory, each page table entry has a present bit, which is used in combination with the valid bit. If the valid bit of a PTE is 0, then the page is invalid. If the valid bit is 1 and the present bit is 1, the page is valid and in physical memory. If the valid bit is 1 and the present bit is 0, the page is valid and not in physical memory.

A **page fault** is when you attempt to access a non-resident page. It can be detected because the page's present bit is 0 but the valid bit is 1. On a machine with a hardware-managed TLB, the MMU detects this when it checks the page's PTE and generates an exception for the kernel to handle. On a machine with a software-managed TLB, the kernel detects the problem when it checks the page's PTE after a TLB miss, since the TLB should never contain any entries that are not present in physical memory. Non-resident pages are never in the TLB because they do not have a corresponding frame number. Page faults are detected through TLB misses. Specifically, when a TLB misses and an exception is thrown, the kernel will,

1. Check the present bit of the PTE. If it is 1, then the page is already in RAM but the TLB is just not properly updated.
2. If the present bit was 0, then this was a page fault.
   (a) Swap the page into memory from secondary storage by finding a frame to put it in, evicting another page from memory if necessary
   (b) Update the PTE with the frame number from step 1 and set the present bit to 1
   (c) Update the TLB with the recently updated PTE
   (d) Return from the exception so that the application can retry the virtual memory access that caused the page fault, except now the appropriate translation is in the TLB

So overall, a TLB miss is when the translation is not in the TLB. A page fault is caused by a TLB miss and is when the page is not even in RAM. Page faults are extremely costly because accessing secondary storage is slow. So to improve the performance of on-demand paging, we reduce page faults by limiting the number of processes (so there is enough physical memory per process), try to be smart about which pages are kept in physical memory and which are evicted, and prefetching pages before a process needs them to hide latency (although this will take CPU time away from doing other things). Also, the disk block size matches page size to minimize I/O service request time on a page fault because it would only require one I/O operation per page fault.

5.9 Page replacement strategies

The replacement policy dictates which page should be evicted when the kernel needs space in physical memory for a new page on a page fault.

FIFO: replace the page that has been in memory the longest. This is fair and easy to implement but is not good for minimizing the number of page faults.

MIN (optimal): replace the page that will not be referenced for the longest time. But this is impractical because the computer cannot predict the future.

Least recently used (LRU): replace the page that was least recently used. Takes advantage of temporal and spatial locality (programs are more likely to access pages that were accessed recently and parts of memory that are close to recently accessed parts of memory). However, LRU is impractical to implement because the MMU does address translation in successful translations and the kernel does not know about it unless there was an exception from a TLB miss/page fault. So to keep track of LRU, the kernel would need to know about every translation since the MMU does not have time stamps for when each TLB entry was used. But this can slow down the kernel. To solve this, the MMU hardware can either track page accesses or each PTE can have a use/reference bit which is set by the MMU each time the page is used (each time the MMU translates a virtual address on that page) and can be read and cleared by the kernel periodically. The use bit does not exactly correspond to LRU but does give some information about recently used pages. **Clock replacement algorithm**: makes use of the use bit. This algorithm has a "victim" pointer which cycles through the page frames until it sees a frame which has the use bit not set. The page is then put into this frame and the victim pointer is incremented to the frame after for the next use. However, once the page is put into RAM, its use bit is not set to 1. This is because after we add this translation to the TLB and re-try the translation, the MMU will be able to translate it and set the use bit to 1 for us. The code is,

```
while use bit of victim is set:
    clear use bit of victim // this step is important!
    victim = (victim + 1) % num_frames
choose victim for replacement // put the page into RAM
victim = (victim + 1) % num_frames // set up for next use
```

6 Scheduling

Given a set of jobs, each with an arrival time a_i and run time r_i, in a system where only one job can run at once, how can we schedule the jobs intelligently to minimize the response time (time between the job's arrival and when it starts to run) and/or the turnaround time (time between the job's arrival and when the job finishes)? Typically, a "job" is just a thread but the concepts in this unit generalize.

6.1 Simple strategies

First come first served: run the jobs in order of arrival (break ties arbitrarily) until completion using a queue. This avoids starvation but doesn't particularly optimize for either response time nor turnaround time. Also since jobs are run in completion, there is no multi-threading which gives the user the illusion that different jobs are running simultaneously, so this is an inefficient use of resources.

Round robin: similar to first come first serve but with preemption. Implemented with a ready queue. Jobs do not necessarily run until completion; when a job (thread) yields or is the quantum expires and it is preempted, it goes to the back of the queue. New jobs which arrive go to the back of the queue as well. If a job finishes before its quantum is over, a new one is selected and the quantum resets itself (quantum always resets when a different job starts to run). If there is only one job but its quantum expires, it continues to run anyway. Similar to how an actual thread will continue to run if its the only one in the system. This strategy ensures fair sharing of the CPU with all available jobs and has timesharing and no starvation. Downsides is that it does not minimize response time nor turnaround time, similar to the first come first serve strategy.

Shortest job first: run the shortest jobs (to completion) first. This can cause starvation in where a long running job never gets to run because shorter running jobs keep entering the system. However, this strategy does minimize the average turnaround time (see CS 341 notes on greedy scheduler for proof).

Shortest remaining time first: similar to shortest job first except when a new thread arrives that has a shorter run time than the current shortest remaining time thread in the system (which is currently running), the currently running thread is preempted and the new thread runs. Preemption here only comes from newly arriving short threads, not a quantum, so if there are no new arriving threads or all arriving threads have longer run time, then run the current job to completion. Starvation is still possible. Minimizes the average turnaround time just like shortest job first.

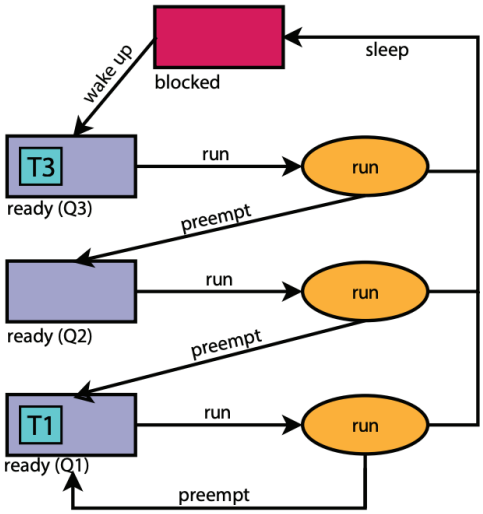
Actual CPU scheduling: the problem with the simple strategies above such as shortest job first in real life implementations is that the run times of threads are normally unknown (can vary based on the state of the system), threads are sometimes not runnable (blocked), and these strategies do not take into account priority. A real CPU scheduler will attempt to balance: responsiveness (threads get to run regularly and there is no starvation), fairness, and efficiency.

6.2 Multi-level feedback queue (MLFQ)

This is a common, modern scheduling algorithm which is used in Windows, macOS, and used to be used in Linux. The idea is to distinguish between interactive threads (ones which frequently block to wait for user input, etc) from non-interactive threads which typically run until quantum expiry. So, interactive threads are given higher priority so that they are more responsive.

The algorithm makes use of n round-robin style ready queues where Q_n has the highest priority with quantum q_n and quantums $q_i \leq q_j$ if $i > j$ (higher priority queues have shorter quantum, there is no one system-wide quantum anymore). The scheduler then chooses a thread from the highest priority which has a non-empty queue to run. Preempted threads (from the quantum running out) or threads which voluntarily yield go onto the back of the next lower-priority queue (or Q_1 if it originally came from Q_1); a thread from Q_n when preempted goes to the end of Q_{n-1}. When a thread wakes up after blocking, it is put on the back of Q_n. So since interactive threads frequently block, they "live" in the higher-priority queues while non-interactive

threads sift to the bottom. To prevent starvation (a thread in Q_1 never running because Q_n is never empty), all threads are periodically placed in the highest-priority queue to "refresh" them. An example with 3 queues,



In this example, suppose T_3 had just been blocked and was woken up (by T_1). Many variants of MLFQ will preempt low-priority threads when a thread wakes up (and thus goes to Q_3) to ensure a fast response to an event. In this case, T_1 had been running at level 2, but it is "denoted" to Q_1 upon T_3 waking up and Q_3 is scheduled to run next. If T_1 had been running at level 3, T_3 waking up would not preempt T_1 because they both have priority 3.

The reason higher priority queues have shorter quantum is because if a thread is indeed interactive and blocks often, it will block before being preempted even if its quantum was short. On the other hand, if the short quantum runs out, then perhaps that thread does not block as frequently as we had thought (not as interactive). Also, if a thread gets preempted, it means it wanted more CPU time than its quantum had allowed, so it goes to the next level and is given more quantum. Thus, this system lets interactive and non-interactive threads self-identify as such. Newly woken threads go on the highest priority queue because we want to respond to whatever condition caused it to be woken (eg a key was pressed and the thread had been waiting for that key to be pressed). Note that every thread starts off at Q_n, whether it is newly arriving or just waking up.

6.3 Completely fair scheduler (CFS)

This strategy is used in Linux. Each thread is assigned a weight (known by the OS) and the "actual runtime" (amount of time that the thread has run on the CPU so far) is recorded for each thread (starts at 0 for new threads). Unlike MLFQ, the quantum is the same for all threads but higher priority threads get a higher weight. The scheduler attempts to give each thread CPU time which is proportional to its weight (high priority should have more CPU time). The "virtual runtime" is computed for each runnable thread: if w_i is the weight and a_i is the actual runtime of thread i, then the virtual runtime v_i is $a_i \cdot (\sum_j w_j/w_i)$. The scheduler chooses to run the thread with the lowest virtual runtime. The reasoning behind this calculation is so that high weight threads have virtual runtimes which advance slowly in proportion to the actual runtime and low weight threads have virtual runtimes which advance quickly. Assuming all threads have the same actual runtime, the high weight threads will have the highest virtual runtime. Thus, this scheduler runs high weight (high priority) threads more but also balances it out with taking into account how long the thread has run for already. A low weight thread that has barely run might have a lower virtual runtime than a high weight thread that has run for an extremely long time, so it will get chosen accordingly. If there is a tie in virtual runtime, then maybe choose the one that ran more frequently, or run the newer thread, etc. There

are two potentially unfair situations in where a thread gets to run before it "should":

1. A thread had been blocked for a very long time and just woke up. Thus, its actual runtime is very low compared to every other thread's actual runtime, so its virtual runtime might be much lower as well, regardless of its priority.
2. A new thread has an actual runtime of 0, so its virtual runtime is 0, regardless of its weight.

In these two cases, we might assign a virtual runtime somewhere between the min and max of the virtual runtimes of the other threads so far. That way, this anomaly thread does not get a very unproportional amount of CPU time. Or maybe you want new threads/threads which had been blocked but just woke up to run first so it can "catch up"? Implementation dependent!

6.4 Other

Scheduling on multi-core processors: you can either have a separate ready queue of threads for each core of have one shared queue and threads can be scheduled on any core. The shared queue needs synchronization (two cores might request a thread at the same time) but then this causes CPUs to wait for each other and this does not scale well. Also, the shared queue has bad **cache affinity** because a thread might run on a different core each time, so CPU caches are not as effective since data must be reloaded to populate the new core's cache even though the core that the thread had previously run on would have had the data in cache already. Cache affinity measures how much data can be left behind in cache and re-used. On the other hand, per-core queues have better **scalability** because they do not have synchronization problems and also have better cache affinity since the CPU cache data can be re-used the next time the thread runs. The problem with this design is that jobs might not be equally distributed in cores (one CPU might have an empty queue, another CPU has a very full queue); this results in poor **load balancing**. A solution to this is to re-distribute all the threads in the queues equally among the cores every once in a while or allow threads to move from heavily loaded cores to lightly loaded ones (**thread migration**).

7 Devices and I/O

A device allows a computer to receive input and produce output. Examples are keyboards, printers, text screen, sound cards, graphics cards, etc.

A **bus** is a communication pathway between devices in a computer. An **internal bus** is for communication between the CPU and RAM and devices also listen in on this bus. A **peripheral bus** is between devices of the computer. A **bridge** connects two different buses.

7.1 Device registers

Communication with devices occurs through device drivers, which exist in the device itself. There are three primary types:

1. status: typically read, tells something about the device's current state. For example, a register on a keyboard that says whether the caps lock key is on.
2. command: write a value to this register to issue a command to the device. For example, the printer will print whatever is in this register.
3. data: used to transfer larger blocks of data to/from the device and is both read and written to.

Some registers are a combination of the three primary types. For example, a status and command register which is read to discover the device's state and written to issue a command.

In SYS/161, an important device is the timer/clock which keeps track of current time and is used for preemptive scheduling. It has six registers in total, each of size 4 bytes:

1. two status registers for the current time in seconds and nanoseconds since Jan 1, 1970
2. a command register "restart-on-expiry" tells the clock to automatically restart the countdown time when its done
3. a status and command register "interrupt" which is written to to indicate that the countdown finished and you write to it to clear this and let the clock proceed.
4. a status and command register "countdown time" in microseconds. Writing to this register starts a countdown for the time written and reading tells you how much time on the countdown is left. This may be used in preemption. For example, if the quantum is 40 ms, then maybe set a countdown for every 5 ms so that every 5 ms, there is a timer interrupt and the kernel can check whether the quantum has expired and if so, to do a context switch using `thread_yield`
5. a command register "speaker" which causes (error) beeps when it is written to. Supposed to be used to indicate errors with the motherboard.

Another example is a serial console, which has three registers, each 4 bytes:

1. a command and data register "character buffer" which tells the device to put a character on the console when something is written to it and reading gives any character typed onto the console.
2. writeIRQ and readIRQ status registers (recall IRQ is interrupt) have something to indicate when a read/write is in process and something to indicate that it was successful. It is the responsibility of the programmer to check these (synchronized) and not do things which cause undefined behavior such as writing while another write is in progress or reading while a write is in process.

The registers for each of these devices is described by whoever made the hardware for others to write drivers.

7.2 Device drivers

A device driver is a piece of software that interacts with a device (hardware). Most drivers are part of the kernel but some exist in user-space (for example, most printer drivers in Linux).

An example of a driver that writes a character to a serial output device using polling (driver repeatedly checks device status):

```
P(output_write_sem) // ensure only one write at a time, thread-safe
write character to device data register
status = not completed
while status is not completed: // wait for the console to finish
    status = read writeIRQ register
write "complete" to writeIRQ register // so the device is ready again
V(output_write_sem)
```

Polling is inefficient (much like spinlocks busy-waiting), use interrupts instead since devices raise interrupts frequently and will raise one when the write/read is done. So, we can have two separate handlers, one to initialize the write and another to acknowledge,

```
// in device write handler
P(output_write_sem)
write character to device data register
// probably block after this to wait for the interrupt to be raised

// in interrupt handler
write "incomplete" to writeIRQ register
V(output_write_sem)
```

So, we needed a (binary) semaphore here and not a lock because the thread that writes (and then blocks) is not going to be the same one that handles the interrupt. Recall locks can only be released by the owners but binary semaphores do not care. However, some devices might not be able to throw such interrupts (eg USBs), so the drivers must do polling. Or, maybe have a separate USB controller that does polling that fires interrupts for the device driver.

7.3 Accessing devices

There are two options to allow a device driver to access device registers, a system may use both:

1. Port-mapped I/O with special Assembly I/O instructions: outside RAM, there is a piece of memory in a separate, smaller address space that we are addressing using "port numbers" and each port is mapped to a device register. So the special Assembly instructions let you read and write to ports, and the devices watch these address lines and when they see a port number that corresponds to one they are using (are mapped to), then the device knows the data read or data requested at that specific piece of memory was for that specific device register. This is fast but there are not many ports available since this separate piece of memory is quite small and you need to use the special in and out Assembly instructions to read and write from this piece of memory. If you want to manipulate data for a port, it needs to be copied into RAM, modified, then copied back.
2. Memory-mapped I/O: finds a region of RAM and dedicates it for device registers then devices listen in on the primary bus and know which RAM addresses it needs to pay attention to (is mapped to) and when a device sees data written to a mapped address, it knows that the data was for itself. Each device register has a physical memory address. This is more flexible than port-mapped, has lots of space, and doesn't require special instructions but devices need to translate every address (from electrical signals to an actual address) to determine if that address applies to it and this is slow.

OS/161 uses memory-mapped I/O and each device is assigned to one of 32 64 KB device "slots" in kseg1, regardless of how much of the slot it uses. Also, hardware providers will document where in the slot each register is. For example for a clock, the current time register might be in the 4 bytes starting at offset 4 in the slot.

The above two strategies are better for small data transfers. For large data transfers, there are two strategies,

1. Program-controlled I/O: the CPU (specifically, the device driver) is responsible for performing the memory transfer itself. For example, a 10 MB chunk might need to be copied from RAM to disk. But then, the CPU cannot do anything else during this.
2. Direct memory access (DMA): the device is already connected to the address bus, so the device can do the transfer itself. The CPU initiates the contact with the device (tells the device what to do, eg what to copy) then the device can directly communicate with memory and do the copy itself. When the device is done, it fires an interrupt to tell the CPU. During the data transfer, the CPU is free to do something else. This whole process is managed by a DMA controller which has its own driver.

Some devices can only use program-controlled I/O and does not support DMA but many use both.

7.4 Persistent storage devices

A persistent storage device (also called non-volatile) is one where data persists even when the device has no power. Examples are SSD, hard disks, CDs, etc, not RAM.

Hard disk: has circular platters (each are possibly double sided) and an arm with one read/write head for each platter. Is always spinning because the time to slow down and start up again is too much. Each platter is divided into concentric rings called tracks and each track is divided into segments (also called blocks, or sectors). Assume each track and segment has the same capacity as each other and each track has the same number of segments. So, to read/write, we must move the read/write head to the correct track then the platter has to rotate so that the correct block is underneath the read/write head. The head can only read one bit at a time so you must continue rotating the platter underneath the head so that it can read the whole block. Since there is so much movement involved, hard disks are quite slow. Blocks are the unit of transfer between disk and memory (typically, one or more contiguous blocks can be transferred in a single operation). To do calculations for I/O performance (request service time), assuming read takes the same time as write, consider the following factors:

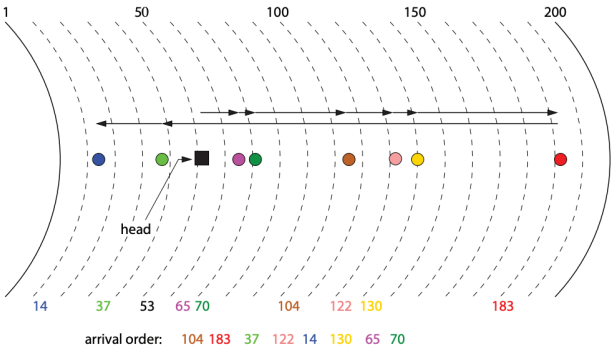
1. Seek time: time to move the read/write head to the right track. Depends on the distance (in tracks) between the last and current request. The minimum is 0 ms, if we are already on the right track and the max is the time to move from the innermost to outermost track. Take average seek time to be max seek time (typically given) divided by 2 for simplicity.
2. Rotational latency: time to rotate the platter to find the right sector. Min time is 0 ms if already on the right sector, max is the time for one full rotation (ie, 60 divided by the RPM of the disk gives the number of seconds to do one rotation and multiplying this by 1000 gives the time in ms, since there are 1000 ms in 1 s). Again for simplicity, the average latency is the max divided by 2.
3. Transfer time: time for all the sectors we want to read/write to to spin past the read/write heads after we find the correct track and sector. The sector latency (time to transfer 1 sector) is the max rotational latency divided by the number of sectors per track and this is also the min transfer time since disks do sector based addressing so you cannot read or write part of a sector, you must at least read an entire sector. So if we want to read 10 consecutive sectors, multiply the sector latency by 10 to get the total transfer time.

The total request service time is the sum of the three times above. You cannot simultaneously find the right sector while finding the right platter because the platters never stop spinning. So, rotational latency comes entirely after seek time. If the head or platter position is unknown, use average seek and average rotation

latency times. To improve performance, we generally want to improve seek time since you cannot do much about rotational latency because the disk is always spinning. Large transfers are more efficient than smaller ones because the time per byte is smaller for larger transfers and sequential I/O is faster than non-sequential I/O because you do not need as many seeks and seeks are usually the dominating cost. However, if we did store everything sequentially then there would be segmentation. So there is a tradeoff between space and time efficiency. Also, note that the number of bytes per track is the total disk capacity divided by the number of tracks and the number of bytes per sector is the number of bytes per track divided by the number of sectors per track. Lastly, the time cost model is still the same with multiple platters because even though each platter has its own read/write head, they all move together so you cannot read from more than one head at once.

To reduce seek times, the OS or device or both may control the order in which I/O requests are serviced using various disk head scheduling algorithms:

1. First come first served: fair and simple but no seek time optimization
2. Shortest seek time first: greedily choose the closest request that minimizes seek time but may cause request starvation (a request that is never serviced)
3. Elevator algorithm (SCAN): the disk head moves in one direction until there are no more requests in front of it then reverses direction, avoids starvation and optimizes seek time



For each of these to work, there must be an outstanding disk request queue. And requests typically arrive at a much faster rate than can be immediately serviced. For example, on demand paging produces lots of disk I/O requests.

Another example of a device is the Sys/161 disk controller, which has 5 registers:

1. status register for the number of sectors, status and command register for the status (if you read from it, you get whether a read/write is in progress or an error number and if you write to it, you tell the disk whether to read/write), command register to indicate which sector number to read/write from, status register for the RPM
2. data register called the transfer buffer which is exactly the size of one sector (recall disks do sector based addressing so read/write occurs in sector units)

And the corresponding handler and interrupt handler for write is,

```
// device write handler (one write at a time)
P(disk_sem)
copy data from memory to device transfer buffer
write target sector number to disk sector number register
write "write" command to disk status register
P(disk_completion_sem)
```

```
V(disk_sem)

// interrupt handler
write "completed" in disk status register
V(disk_completion_sem)
```

There are two semaphores because the handler which initiated the write should block until the write has finished and the interrupt handler has written "completed" to the disk status register. The drivers for read are similar, except the `P(disk_completion_sem)` comes before copying the data from the transfer buffer into memory. The thread which started the write must block until the write finishes so that it can be the one that deals with any errors, rather than some other thread dealing with errors. For reading, this is especially important since the thread that started the read must also be the one that copies the data from the transfer buffer into RAM.

SSD: does not have any mechanical parts and uses integrated circuits for persistent storage rather than magnetic surfaces, which is why it is faster than hard disks. There are two main implementations: DRAM, which is made of small transistors which are powered by capacitors and these need to be constantly refreshed by some other power source, and flash memory, which "traps" electrons in a quantum cage and does not need constant power. Flash memory is logically divided into blocks and pages (blocks are made of pages). A value of 1 represents "unwritten" because going from a 0 to a 1 requires a much higher voltage than the other way around and we want fast writing to new pages (explained later). So, going from 1 to 0 is writing a new page and 0 to 1 is overwriting or deleting a page. However, the high voltage required from 0 to 1 can only be applied at the block level. Otherwise, reading/writing and writing a new page can be done at the page level. The naive and slow method for writing and deleting from flash memory is to read the whole block into RAM, re-initialize the block into all 1's, update the block, then write it back into the SSD. The faster way, which is how the SSD controller does it, is to mark the deleted/overwritten page as invalid, write to a new unused page (all 1's), and update the translation table (the SSD's own internal page table which keeps track of which pages are in use, free, or garbage). This is faster because going from 1 to 0 is faster than 0 to 1. Periodically, the controller must do garbage collection to clear out these "invalid" marked pages, and if the controller dies, you cannot ever recover your SSD data since you no longer have access to the translation table. Also, since there is no external fragmentation on SSDs, defragmentation is useless and since there are no moving parts, sequential I/O has the same cost as non-sequential I/O. Lastly, each block has a limited number of write cycles before it becomes read-only, and the SSD controller will mark it as such. To slow this down, the SSD does wear-leveling (writing to pages which are "fresher", regardless of where they are located in the SSD).

Persistent (resistive) RAM: like "normal" RAM except values are persistant even with no power. Can be used to improve the performance of secondary storage such as hard disks by caching disk data blocks and i-nodes. RAM can also do this caching but preferably should be caching address spaces, not stuff for disks. Roughly as performant as SSD.

8 File Systems

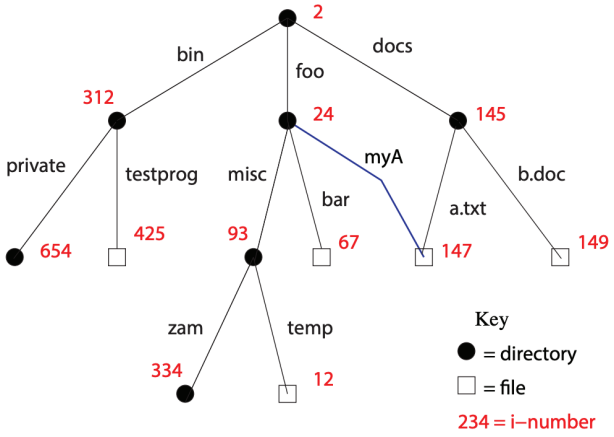
A **file** is a persistent, named data object where the data is a sequence of numbered bytes. Files have variable size and have meta-data (type, timestamp, etc) associated with them. The **file system** is the data structures and algorithms used to store, retrieve, and access files. We can consider three levels of a file system:

8.1 Logical file system

There are a few relevant high-level functions which abstract the hardware and low-level details away,

1. `open`: returns a file descriptor to identify the file. Each file descriptor has an associated file position, which starts at byte 0 when the file is opened.
2. `close`: invalidates a valid file descriptor, which the kernel keeps track of for each process in an "open file" table. Need to close files because sometimes writes do not occur as you call them but close will flush the write buffer so that the file actually gets written to.
3. `read and write`: reads and writes back and forth between a file with file descriptor `fd` and a virtual address space. Starts at the current file position, which is updated as bytes are read/written. This gives good sequential file I/O. An example is `read(fd, buf, 512)`, this also automatically advances the file position by 512 bytes.
4. `seek (lseek)`: enables non-sequential reading/writing, changes the file position associated with the file descriptor and the next read/write will use the new position. Does not check if the new file position is valid, the next read/write will error if it is invalid. `seek` does not even touch the file at all, so causes zero disk I/O operations. All it does it go into the process's file descriptor table and update the current position to whatever was given. An example is `lseek(fd, 100, SEEK_SET)` to set the offset to 100 bytes.
5. `fstat, chmod, ls`: get/set file meta-data

A **hard link** is an association between a path name (as a string) and the **i-number** of the file/directory associated with that path. The i-number is a unique identifier, used in the kernel. Each entry in a **directory** is a hard link. The hierarchical namespace can be naturally imagined as a tree, rooted at the directory `/`. Directories are themselves "files" but cannot be opened and written to. Hard links are created when the file is created and additional hard links can be made to existing files, such as `link(/docs/a.txt, /foo/myA)` creates a hard link `myA` in `/foo`, as seen below. Thus, each file has a unique i-number but can have multiple pathnames given by these hard links. It is not possible to `link` to a directory or else there would be a cycle.



In the tree, the leaves are either files or empty directories. Only the kernel can edit directories to preserve the i-number mappings in the kernel. Since there is a hard link, now `/foo/myA` "refers" to `/docs/a.txt`; so

the kernel must keep track of files using their unique i-number rather than the possibly not unique path. Hard links can be removed `unlink(/foo/myA)` and when the last hard link to a file is removed, the file is also removed since it is no longer accessible in any way. `rm` removes all hard links to the file (so the file is removed) but the data is not actually deleted. A **(symbolic) soft link** is a file which just contains the original path. You can `unlink` on the "original" path and the hard link will still work but soft links will not.

8.2 Virtual file system

This is optional and only needed if there is more than one physical file system which must be presented to the logical file system as one unified system (global file namespace). In Unix, `mount` will combine the namespaces of two file systems into one single hierarchical namespace (but the file systems themselves are not merged). For example, `mount(file system X, /x/a)` will create a namespace such that the "root" of `X` is at `/x/a`. A file `/test` in `X` now has the path `/x/a/test`. The namespace trees from above can be imagined as merged together. There might be i-number "collisions" from this but the kernel treats the two file systems as separate so it is aware of possible collisions and there is no real problem. Note that a file always has the same i-number, even when moved somewhere else.

8.3 Physical file system

The physical file system defines how files are actually stored on physical media. There are many different types, such as HFS, NTFS, Amazon S3, etc. Things which need to be stored persistently include: file data, meta-data, directories and links, file system meta-data. Things which do not: per process open file descriptor table (including the file position), system-wide open file table, cached copies of persistent data. Recall from earlier that disks are **sector addressable**. Group every 8 consecutive sectors and call it a block; this gives fewer seeks and fewer block pointers (see later). Physical file systems are **block addressable** which means you cannot read/write a partial block, you must at least read/write one block.

8.4 Very simple file system (VSFS)

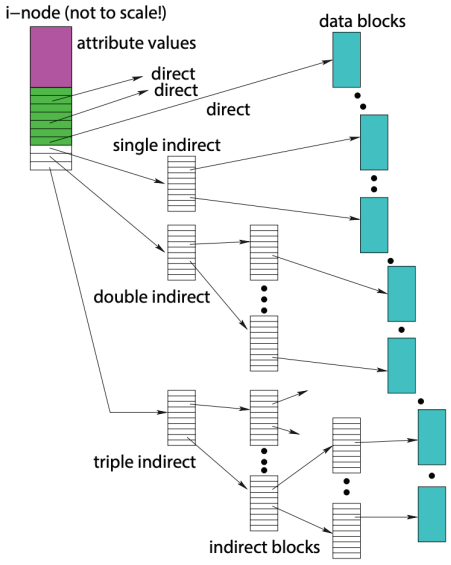
This is an example physical FS on a 256 KB disk with 512 byte sectors (and thus 4 KB blocks) for a total of 64 blocks. The last 56 blocks (blocks 8 to 63) are for user data (data in the files and links in the directories, note that two files cannot share a block, so we have at most 56 files in this file system). Although the drive can do sector based addressing, sectors are smaller than pages and the block size is conveniently equal to the page size in VSFS so we do block based addressing, which also has the added benefit of giving good sequential I/O performance. But, we do not yet know which blocks each file is in. We do not necessarily want consecutive blocks because it can cause external fragmentation.

Instead, blocks 3 to 7 contain an array of i-nodes (more on this later) where the index into this array is the file's i-number (index number). So, if each i-node is 256 bytes, since there are 5 blocks for this array, we can have 80 total i-nodes in the array, which is enough. Next, we need to know which i-nodes and blocks are unused so that we do not have to linearly search through the i-node array whenever we have a new file. We do this by reserving block 1 for a **bit map** for the i-nodes and block 2 for a bit map for the data blocks, where one bit is used to track each i-node or block and is 0 if the i-node/data block is free or 1 if in use. So, deleting a file involves writing a 0 into the corresponding bit maps to mark the data blocks used as free and the i-node as free. The data blocks themselves are not cleared. Empty files do not take up a data block but do have an i-number. Since blocks are 4 KB each, the bit maps can track 32 KB i-nodes and 32 KB data blocks (one bit for each i-node/block), which is enough. Lastly, block 0 is the **superblock** and contains meta-data about the entire file system such as how many i-nodes and blocks there are, where the i-node table begins, etc.

An **i-node** (index node) is a fixed size structure that holds file meta-data and pointers to data blocks: file type, permissions, length/size (note file size in bytes might be different from file size on disk due to block based addressing so a small file takes up a whole block), number of file blocks, last access time, last i-node

update, last file update, number of hard links to this file, direct data block pointers, single/double/triple indirect data block pointers.

A **direct** data block pointer is just a pointer which points to a data block where the file data is. The pointers are ordered (first pointer points to first block in file, etc). So, if an i-node has 12 direct pointers, then the largest file size from just these pointers is 48 KB, since each block is 4 KB. However, this might not be good enough for large files. A **single indirect** pointer points to a data block which itself is filled with direct pointers. A 4 KB block gives 1024 pointers (assuming 4 byte pointers). So the maximum file size for an i-node with one indirect pointer is 4096 KB. A **double indirect** points to a 4 KB data block of single indirect pointers (which each point to data blocks of direct pointers). For example, if an i-node contains 12 direct pointers, an indirect, and a double indirect pointer, then the largest file size from these is $(12 + 1024 + 1024^2) * 4$ KB. The i-node specification will say how many of each pointer it contains. These internal data blocks for indirect pointers are only allocated if needed, much like a multi-level page table. Note in the picture below that the internal "nodes" in each indirect tree are also data blocks, although not explicitly blue.



These are the steps to perform various disk operations on `/foo/bar`, assuming no i-node nor data block cache for now:

1. `open(/foo/bar)`: read root i-node, read root data block to get the i-number for `foo`, read `foo` i-node, read `foo` data block, read `bar` i-node to see permissions (whether we can open it) and return a file descriptor (the i-number) which is added to the process's file descriptor table and the file is also added to the kernel's list of open files. The file position is 0 to begin with.

2. `read(/foo/bar)`: to read the first block, read `bar`'s i-node for permissions then read `bar`'s first data block. Unlike opening the file, we do not have to traverse from the root because we already have the i-number for `bar` so can index into the i-node array directly. The last step is to write to `bar`'s i-node to update the access time. Same thing happens to read the second, third, etc block of `bar`. Also, even if the user only wanted a single byte out of the middle of the block, the entire block must be read because the file system only permits block-based addressing.

3. `create(/foo/bar)`: read root i-node, read root data, read `foo` i-node, read `foo` data, read i-node bitmap (to find a space for `bar`), write i-node bitmap, write `foo` data (to add an entry for `bar` in directory `foo`), read `bar` i-node, write `bar` i-node, write `foo` i-node (to update the modification time stamp).

4. `write(/foo/bar)`: read `bar`'s i-node, if writing to an entirely new data block, we then read the data bitmap, write to the data bitmap, write to the data block, then write to `bar`'s i-node to update the timestamp. If the data block already exists, there's no need to read/write from the data bitmap, we just write directly to the data block (if writing an entire block) or read first then write (if writing to a partial block).
5. deleting `/foo/bar`: read root i-node, read root data, read `foo` i-node, read `foo` data, read `bar` i-node, read data bitmap, write data bitmap (mask the data blocks as free), read i-node bitmap, write i-node bitmap, write `foo` data to remove the hard link to `bar`, write `foo` i-node to update the timestamp.

If there were an i-node or data block cache, then accessing cache would not be considered an I/O operation. However, writing to a value in cache would be because eventually that data must be written onto disk. Overall, here are the rules to follow,

1. A directory's latest modification timestamp needs to be updated when the directory is modified (add/remove/change hard link inside)
2. If you want to write in the i-node array, you first need to read the i-node (block) because the i-node is just a very small part of a block. Here, "reading" an i-node actually means reading the block which it is contained in.
3. Can assume that each directory fits in a single data block of hard links; no indirect pointers to a directory.
4. When reading or writing, we do not have to traverse the entire namespace tree from the root, we can directly index into the i-node array using the i-number (file descriptor).
5. When writing a partial block, that block must be read first. If writing an entire block, no read is required.
6. Reading from indirect pointers needs more disk I/O opeations because the indirect data blocks needs to be read first. While a normal read from a direct pointer may take x reads, a single indirect will require $x + 1$, a double indirect $x + 2$, etc.

Suppose an i-node has 9 direct pointers, 1 indirect pointer, and 1 double indirect pointer. In order, these give the file; the first byte of the file is pointed to by the first direct pointer, then when the file is big enough that the blocks pointed to by the 9 direct pointers cannot fit it, the 1 indirect pointer comes into play. The file data is always ordered this way amongst the blocks. For example, assuming a 1 KB block size and 4 byte pointer, the 9 direct pointers can "hold" a 9 KB (or 9216 byte) file. On the 9217 byte, the indirect pointer is needed. And when that is used up, the double indirect is used.

There are two alternatives to per-file indexing which uses direct/indirect pointers to point to each block in the file:

1. Chaining: each block includes a pointer to the next block in the file. If a directory, contains a table with the name of the file and each file's starting block. Okay for sequential access but very slow for random access in the file since it requires a linear search through the blocks. Still need i-nodes, each of which only need to point to the first block of the file.
2. External chaining: has a separate file access table which specifies the file chains rather than putting the chains in the data blocks themselves. So, each entry in this table has an address of the data block it corresponds to and an index to the next block in the chain. This is a significantly smaller table to go through than the data blocks themselves. Still a linked list like chaining but each node only has an address and index rather than an entire data block being the node.

These have less meta-data than VSFS and might be better for sequential I/O.

Notes regarding design, failure, fault tolerance:

1. For file system design, we might consider how many i-nodes it should have, how many direct/indirect blocks, the right block size (for example, if all files are large then choose a large block size and lots of indirection), etc. In general, most files are small even though the average file size is growing.
2. A single file operation such as deleting a memory can require several disk I/O operations but there might be a problem if the i-node entry in the bit map is marked as free but the data block bit map has not been updated yet and the power goes out. So, the file system should be crash consistent.
3. One way to deal with potentially inconsistent file system data structures is to run a consistency checker after a crash to try to fix it (eg, free space that is not marked as free). Another way is write-ahead journaling, in where you log the meta-data changes you want to make and periodically execute the changes that were logged and do not delete the entry from the journal until it has finished. So, if the power goes out, the journal is intact and you know which meta-data changes you were trying to do. But, if the power goes out while writing to the journal, you will still have problems.

9 Virtual Machines

A **virtual machine** is a simulated/emulated computer system and provides the ability for one machine to act as many. For example, SYS/161 is an emulation of a MIPS R3000. The OS and programs running on a virtual machine should not be aware that it is virtualized and should operate normally; the virtual machine should think it is the only one running.

The **guest OS** runs on the virtual CPU (virtual machine), the **host OS** runs on the physical (actual) machine. For example, OS/161 is the guest OS on top of a virtual machine (SYS/161, which is a type 2 hypervisor), which runs as a program in the host OS. A naive attempt for virtualization is to capture the instructions for the virtual CPU and translate them into instructions for the physical CPU. But, SYS/161 is an unprivileged program in the host OS so if the guest OS disables interrupts, this should not be able to disable interrupts on the physical CPU. Or, if a keyboard press gives an interrupt, which OS should deal with it? Lastly, what if the guest OS uses paging for virtual memory and maps a page to a frame that the host OS also maps a page to? So, this naive approach cannot deal with interrupts and privilege (and also virtual memory).

Hypervisor: a virtual machine manager that creates and manages virtual machines. There are two types:

1. Type 1: runs in privileged mode on bare hardware (the physical CPU) and VMs run on top of the hypervisor in less privileged modes (cannot disable interrupts, etc). Unprivileged instructions execute normally but privileged instructions are dealt with by the hypervisor. For example, if the VM wants to disable interrupts, the hypervisor can emulate this behavior without actually doing it. A type 1 does most of the job of a kernal, such as memory management but not all; it is a little like an OS and runs on bare metal so can do anything it wants. However, if the VM itself is in unprivileged mode and user programs running on the guest OS on the VM are also unprivileged, then for the hypervisor to distinguish between these, we need rings of privilege. The hypervisor has the most privilege, guest OS have the next highest, and user programs have the least. Thus, the hypervisor can now determine whether a privileged instruction came from the guest OS or a user program running in the guest OS.
2. Type 2: runs in unprivileged mode in the host OS which runs on top of bare hardware and VMs run in unprivileged mode on top of the hypervisor. Same with in type 1, unprivileged instructions execute normally and privileged ones are dealt with by the hypervisor and possibly emulated or make system calls to the host OS if necessary.

Dealing with virtual memory: each guest OS is unaware of each other and thinks it has the full range of physical memory to use. So, to prevent collisions in the page table where multiple guest OS are mapping pages to the same frame, type 1 hypervisors (which offer memory management and other kernel-like functions) have **shadow page tables** in where guest physical frames (physical address on the VM) is treated like another level in the page table and maps to a physical frame which is the actual physical address. So, the hypervisor prevents collision with these shadow page tables but now, every translation needs to go through the hypervisor rather than just the MMU. This can be slow, so translation is usually done in hardware; CPUs support these shadow/extended page tables and allow the MMU to directly translate between guest virtual addresses to actual (host) physical addresses. This is done by having the guest physical table entries point to the shadow page tables which contain host physical frames. And when the executing VM changes entirely, the hypervisor updates the MMU's shadow page table base register.

I/O and devices: If a guest OS wants to save data to a file, rather than partitioning the disk for each guest OS, the hypervisor creates a file on disk and presents it to the VM/guest OS as a file system (for example, as a magnetic tape). For devices required for I/O, they can be assigned to a specific VM (called **device pass through**, offers device isolation so that a keyboard press does not wake up VMs that it is not assigned to) and interrupts for devices are directed to their assigned VM. Hardware support is required for this.

Overall, VMs give safe sandbox environments to run sensitive applications without affecting the integrity of the real system or test/develop programs for different architectures or OS.

10 Other

volatile C keyword: tells the compiler to read and write to the vafriable in RAM rather than cache it in a register. The variable should not be optimized out by the compiler. Useful if the variable might be changed outside of the context of the code (for example, by another thread or memory mapped hardware).

goto C keyword: transfers control to a labelled statement and can be common in large codebases (especially OS) and can reduce code duplication and flag variables. Do not use goto backwards, into nested statements, or past important variable initializations.

C things:

1. If you don't close open files in C, you may run out of file descriptors.
2. Memory which is malloc'd but not freed won't be available for use until the program terminates (and the kernel re-claims the memory).
3. Free on NULL does nothing, free on bad address or already free'd pointer is bad.
4. Endianness defines the byte orders in memory. Little endian (eg Intel x86) is where the least significant bytes come first in the smallest address. Big endian (eg System/161) is the opposite and is probably what you are logically used to. Little endianness is a legacy of older processors, since it was slightly more efficient for some math operations and also required fewer instructions for casting values.

MIPS things:

1. Using 32 bit MIPS in this course
2. The stack grows downwards (lower addresses)
3. Stack space is handed out by the OS to threads and stack and heap space is uninitialized

Quick maffs:

1. $4096 = 2^{12}$
2. 512MB = 0x2000 0000 byte address. So, if a physical address is less than 0x2000 0000, it may possibly be from any of `kseg0, kseg1, kuseg` but if it is greater than or equal to 0x2000 0000, it must be a user address.
3. One hex digit corresponds to 4 bits and ranges in value from 0 to 15
4. There are $1024 = 2^{10}$ bytes in a KB and 2^{10} KB in a MB and 2^{10} MB in a GB
5. There are 1000 ms in a second

Random things:

1. If there is a fatal bug (eg segfault) in kernel code, the OS should panic and die. If there is a segfault in a user program, the process should be killed but the OS should not panic. Part of A3 is to implement this behavior.
2. It is safe to re-use the PID of a process if no process can call `waitpid` on it and this happens in two cases: the process is an orphan or `waitpid` has already been called on it (cannot `waitpid` on the same process more than once).
3. The system call `kill` terminates a process. Only the user which made the process, the process itself, or users with admin rights should be able to call `kill` on a process.
4. Remember KASSERTs in pseudo-code in exams.