Processes / Threads

- Returning from main calls exit syscall. Exit shuts down the entire process without waiting for threads to finish.
- When a thread returns from the function it began running due to pthread_create, pthread_exit is called implicitly.
- fork() uses copy-on-write, don't need to copy entire address space immediately
- Linux fork() spawns a new process and copies the entire address space and FDs of the parent process.
- Linux exec will override the current process image with a new image. No fork
- Pintos exec() spawns a new process, but loads an entirely new program/process image. Linux fork+exec
- A process's address space is in user space.
- The OS directly schedules processes, not threads. (False) -> Threads encapsulate concurrency and are scheduled by the OS.
- User-level threads incur lower overhead than kernel-level.
- Kernel threads: Each thread individually schedulable. Requires mode switch to switch threads
- Processes can share data via pipe, socket, open shared file, setup shared-memory segment.
- Advantages of every user thread matched with kernel thread in Pintos: 1) when user thread makes a syscall that must block, the thread can be blocked at any time by putting the kernel thread to sleep and waking another kernel thread from ready queue. 2) kernel thread helps to make sure that page table is constructed properly so that user thread's address space is protected from threads in other processes. 3) kernel gains safety because it doesn't have to rely on user's stack pointer register for correct behavior.
- Fork API: 1) allocate new PCB; 2) duplicate reg Vals, addr space, flags, reg state, open files; 3) allocate new pid; 4) mark READY state.
- Exec API: 1) replace code and data segment; 2) set EIP to start of new program. Reinitialize SP, FP; 3) push args onto stack
- Wait: wait blocks parent process until one of its children processes exits.
- ThreadFork(): 1) sanity check arg; 2) enter kernel mode and sanity check arg; 3) allocate new stack and PCB; 4) initialize TCB and place on READY list
- Syscall/Intr(U->K): PL 3->0; Iret(K->U): PL0->3
- Pintos: pure kernel threads have no corresponding user-mode thread

Kernel

- Device drivers, schedulers are part of the kernel
- Base and Bound registers (hardware) can only be modified by the kernel to ensure isolation and protection. Cannot share memory b/t processes, cannot relocate programs
- How does the Pintos kernel find the struct thread of the currently running kernel thread? The kernel rounds down %esp to the nearest page. Each struct thread is at the bottom of the page that its corresponding kernel stack is on.
- In Pintos, the virtual addr 0xc0000008 refers to the same physical address no matter which process we are in. (True): Addresses above PHYS_BASE are mapped to the same location in physical memory.
- Privileged instructions (hardware ensures): Cannot: set mode bit, change address space, disable interrupts, perform IOs, halt processor

User

- C std lib is linked to user program
- Shell is user program

IVT / Interrupt Handler / Interrupt

- What role does the IVT play in protecting the kernel? The IVT ensures that we only start executing kernel code at predefined entry points.
- User cannot execute arbitrary code after transition to kernel mode because entry into the kernel is through controlled entry points (IVT).
- The hardware saves the values for the stack pointer and program counter before jumping to the interrupt handler.
- Higher priority interrupts are able to preempt lower priority interrupts.
- Modern OS use a timer interrupt to regain control from an executing user thread. The timer interrupt handler will initiate a context switch to another thread.
- Interrupts are disabled when an interrupt handler is running.
- An interrupt handler must not sleep while waiting for another event, since it doesn't have a thread-control block (context) to put onto a wait queue and is operating with interrupts disabled.

Context Switch / Transfer Control / Syscall

- Synchronous events: syscall, exception; Asynchronous: interrupt
- Context switching is accomplished by swapping kernel stacks, not user stack. The user stack is in user memory and the user stack pointer is pushed on the kernel stack automatically on entry to the kernel. Consequently, swapping the kernel stack will implicitly swap out the user stack.
- Ways in which the processor transitions from user mode to kernel mode: 1) syscall, 2) processor encounters synchronous exception (page fault), 3) respond to interrupt, 4) signals (handled by kernel)
- During a system call, the kernel validates arguments on the user stack then copies them over to the kernel stack. (False) -> The kernel copies over the arguments onto the kernel stack before validating them to prevent against time-of-check to time-of-use (TOCTOU) attacks.
- What does the $0x30 represent in int $0x30 in Pintos? int $0x30 is used to trap to the kernel to make a system call. $0x30 represents the index number in the interrupt vector table that stores the handler for system calls (i.e. syscall_handler).

I/O

- The same file descriptor can correspond to different file descriptions across different processes. Each process has its own FDT stored in kernel memory.
- Open FDs are preserved across the linux exec syscall.
- lseek() within one process may be able to affect the writing position for another process (after fork).
- Calling lseek() on an open socket file descriptor results in error. This is because they are abstractions of network connections, not actual fields on disk.
- High-level file API buffers data in user memory. Low level IO -> kernel space.
- High-level API reads data in large blocks, often result in fewer syscalls.
- ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count) is faster because sendfile() copies data from in_fd to out_fd without transferring data to and from user space. Fewer copies make the transfer faster.
- IPC using shared memory is faster than using pipes. Each read/write only requires single memory operation for shared memory, whereas pipes require switching to kernel mode.
- Uniformity idea from POSIX covers open, close, read, write. Socket does not support lseek.
- Each client connects to the same server socket, but reads/writes to a separate connection socket after the connection is accepted.
- Each connection is uniquely identified by (sourceIP, destIP, source port, dest port, protocol)
- Steps needed to establish a pipe between a parent and a child process: call the pipe call, then call the fork syscall. Finally close one end of the pipe in the parent process and close the other end in the child process.
- After forking, open files are preserved, so closing a file descriptor in the child process would make the parent process unable to use its own file descriptor that refers to the same file object. (False) -> File descriptors index into file descriptions which are reference counted, so the parent can still use the same file descriptors even after a child closes them.
- Why high-level I/O uses a user space buffer: System calls to low-level I/O are much more expensive (25x) compared to or- dinary function calls. Byte-oriented kernel buffers exacerbate this problem and greatly reduce throughput. Moreover, low-level I/O lack functionality (e.g. reading until a newline using fgets or getline) that are useful to users.
- Pipes are implemented as a kernel buffer (not files) with two FDs, one for writing and one for reading pipe. FDs are copied, not shared. Block if the pipe is full. Block if the pipe is empty.
- Pipes Between Processes: After last write descriptor is closed, pipe is effectively closed. After last read descriptor is closed, writes generate SIGPIPE signals. If process ignores, then the write fails with EPIPE error.

Concurrency / Synchronization

- cond_wait() and cond_signal() can only be used when holding the lock associated with a conditional variable.
- disable/enable disrupt implementation of lock_acquire() requires that sleep() should trigger the scheduler and the next scheduled thread should enable interrupts.
    - Before put on queue: early release
    - After put on queue: deadlock, never wake up
- Sema = n means that some resource has up to n concurrent usages.
- It's possible for busy waiting to perform better than conventional blocking synchronization in certain cases. Putting a thread to sleep and waiting it up can take more CPU cycles.

- Semaphore: 1) mutual exclusion (init=0). 2) scheduling constraints (init=1)
- Monitors can be used to implement semaphores.
- Monitors = lock + ConVar *
- Can a user thread acquire a lock without entering the kernel? Yes. You can use an atomic read-modify-write instruction.
- Monitor programming paradigm requires only sleep on cond vars inside the critical section with the lock held. Never release the lock before waiting on a cond var.
- Hoare: context switch & transfer locks immediately, don't need to check predicate again upon waking up. (inefficient)
- Mesa: Need while loop because by the time the waiter gets scheduled, condition may be false again, so check again with while loop. Signaler keeps lock and processor. Waiter placed on ready queue with no special priority.
- Linux futex: fast userspace mutex. Interface to kernel sleep() function. Used within implementation of pthreads. Can be used to implement lock, semi, monitor. No busy waiting. Every unlock has to go thru kernel.

Misc

- Referee: fault isolation (process, dual mode), scheduling, pipes/socket. Glue: common services (File system, network, UI)

Pintos List

```
struct list* fdt = &thread_current()->pcb->fdt;
struct list_elem *e;
for (e = list_begin(fdt); e != list_end(fdt); e = list_next(e)) {
    struct fd* fd = list_entry(e, struct fd, elem);
    if (fd->num > max_fdnum) max_fdnum = fd->num;
}
```

I/O Read / Write

```
void copy_high(const char* src, const char* dest) {
    char buffer[100];
    FILE* rf = fopen(src, "r");
    int buf_size = fread(buffer, 1, sizeof(buffer), rf);
    fclose(rf);
    FILE* wf = fopen(dest, "w");
    fwrite(buffer, 1, buf_size, wf);
    fclose(wf);
}
void copy_low(const char* src, const char* dest) {
    char buffer[100];
    int rfd = open(src, O_RDONLY);
    int buf_size = 0;
    while ((bytes_read = read(rfd, &buffer[buf_size],
            sizeof(buffer) - buf_size)) > 0) {
        buf_size += bytes_read;
    }
    close(rfd);
    int bytes_written = 0;
    int wfd = open(dest, O_WRONLY);
    while (bytes_written < buf_size) {
        bytes_written += write(wfd, &buffer[bytes_written],
            buf_size - bytes_written);
    }
    close(wfd);
}
```

Semaphroe

```
typedef struct shared_data {
    sem_t sem;
    int data;
    int ref_cnt;
    pthread_mutex_t lock;
} shared_data_t;
void* save_data(void* shared_pg) {
    shared_data_t* shared_data = (shared_data_t*)shared_pg;
    shared_data->data = 162;
    sem_post(&shared_data->sem);
    pthread_mutex_lock(&shared_data->lock);
    int ref_cnt = --shared_data->ref_cnt;
    pthread_mutex_unlock(&shared_data->lock);
    if (ref_cnt == 0) free(shared_data);
    return NULL;
}
```

```
void initialize_shared_data(shared_data_t* shared_data) {
    sem_init(&shared_data->sem, 0, 0);
    shared_data->data = -1;
    shared_data->ref_cnt = 2; // two threads
    pthread_mutex_init(&shared_data->lock, NULL);
}
int wait_for_data(shared_data_t* shared_data) {
    sema_wait(&shared_data->sem);
    int data = shared_data->data;
    pthread_mutex_lock(&shared_data->lock);
    int ref_cnt = --shared_data->ref_cnt;
    pthread_mutex_unlock(&shared_data->lock);
    if (ref_cnt == 0) free(shared_data);
    return data;
}
```

Client Protocol

```
char *host_name, port_name;
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype, server->ai_protocol);
connect(sock_fd, server->ai_addr, server->ai_addrlen);
run_client(sock_fd);
close(sock_fd);
```

Server Protocol

```
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family, server->ai_socktype, server->ai_protocol);
bind(server_socket, server->ai_addr, server->ai_addrlen);
listen(server_socket, MAX_QUEUE);
while (1) {
    int conn_socket = accept(server_socket), NULL, NULL;
    server_client(conn_socket);
    close(conn_socket);
}
close(server_socket);
```

```
bool send(chan_t* chan, int value, bool wait) {
    pthread_mutex_lock(&chan->lock);
    while (circ_buf_len(&chan->buf) == circ_buf_capacity(&chan->buf)) {
        if (wait) {
            pthread_cond_wait(&chan->cond_send, &chan->lock);
        } else {
            pthread_mutex_unlock(&chan->lock);
            return false;
        }
    }
    circ_buf_put(&chan->buf, value);
    pthread_cond_signal(&chan->cond_recv);
    pthread_mutex_unlock(&chan->lock);
    return true;
}

bool recv(chan_t* chan, int* dst, bool wait) {
    pthread_mutex_lock(&chan->lock);
    while (circ_buf_len(&chan->buf) == 0) {
        if (wait) {
            pthread_cond_wait(&chan->cond_recv, &chan->lock);
        } else {
            pthread_mutex_unlock(&chan->lock);
            return false;
        }
    }
    *dst = circ_buf_get(&chan->buf);
    pthread_cond_signal(&chan->cond_send);
    pthread_mutex_unlock(&chan->lock);
    return true;
}

# If combine okToRead and okToWrite
if (AR == 0 && WW > 0) cond_braodcast(&okContinue); #read
if (WW > 0 || WR > 0) cond_broadcast(&okContinue); #write
```

```
Reader() {
    acquire(&lock);
    while ((AW+WW) > 0) {
        WR++;
        cond_wait(&okToRead, &lock);
        WR--;
    }
    AR++;
    release(&lock);
    // AcessDatabase
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
            cond_signal(&okToWrite);
    release(&lock);
}
Writer() {
    acquire(&lock);
    while ((AW+AR) > 0) {
        WW++;
        cond_wait(&okToWrite, &lock);
        WW--;
    }
    AW++;
    release(&lock);
    // AccessDatabase
    acquire(&lock);
    AW--;
    if (WW > 0) {
        cond_signal(&okToWrite);
    } else if (WR > 0) {
        cond_broadcast(&okToRead);
    }
    release(&lock);
}
```

```
# Lock Implementation: Test & Set + futex + maybe
# SYSCALL FREE
acquire(int *thelock, bool *maybe) {
    while (test&set(thelock)) {
        *maybe = true;
        futex(thelock, FUTEX_WAIT, 1);
        *maybe = true;
    }
}
```

```
release(int *thelock, bool *maybe) {
    thelock = 0;
    if (*maybe) {
        *maybe = false;
        futex(&value,
FUTEX_WAKE, 1)
    }
}
```