



CS 162 Midterm 1 Review

Fall 2024

Zoom Logistics

- This presentation will be recorded, so turn your video off if you do not want to be in the recording
- Use chat to answer or ask questions

Disclaimer

This is not an exhaustive review of all topics that are in scope for the midterm.

These are course materials that are all in scope:

- Lectures 1-10
- Project 0
- Project 1 Design
- Homework 0-2
- Discussions 0-3

Outline

- Dual-mode operation, context switching, and interrupts
- Processes and Threads
- File API, I/O
- Locks, semaphores, condition variables, synchronization

Operating System

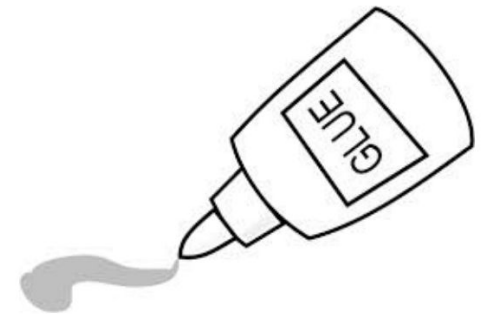
Layer of software that interfaces between (diverse) hardware resources and the (many) applications running on the machine



Referee



Illusionist



Glue

Dual Mode Operation

Kernel

- Process Management
- File System Management
- Memory Management
- Device Management
- System Calls
- Security and Protection

User

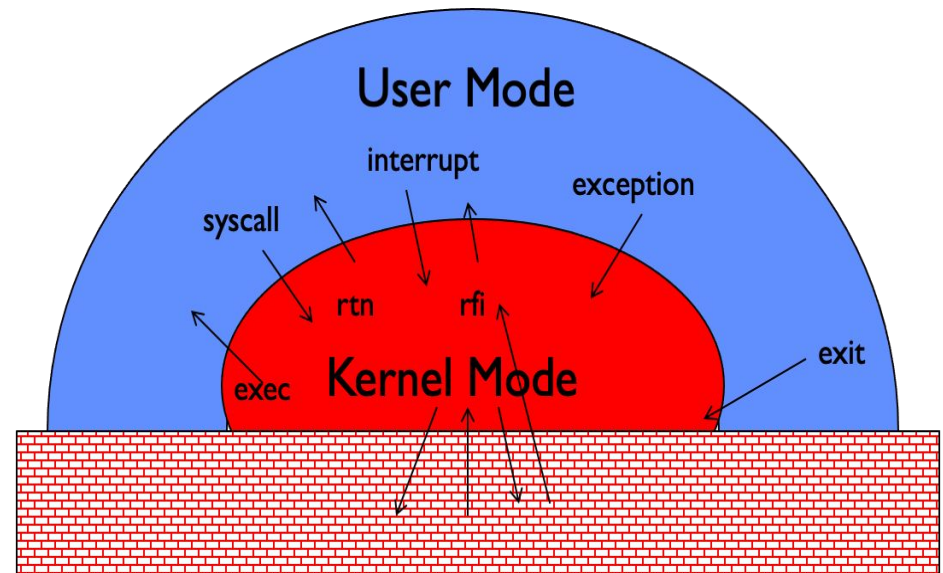
- User-level applications
- Standard user-level code within restricted virtual address spaces
- Most shell commands

What hardware support is necessary to enable protection?

- Privileged Instructions
 - Unsafe instructions cannot be executed in user mode
- Memory Isolation:
 - Memory accesses outside a process's address space prohibited
- Interrupts:
 - Ensure kernel can regain control from running process
- Safe Transfers:
 - Correctly transfer control from user-mode to kernel mode and back

How do you switch from kernel mode to user mode and back?

- **Syscalls** - used by processes to request certain services (e.g. exec, read, write)
- **Hardware Interrupts** - external asynchronous events (e.g. timer, I/O)
- **Traps** - software interrupts or exceptions, internal synchronous events (e.g. exceptions)



Interrupt Handling Roadmap





- 1) Processor detects interrupt
- 2) Suspend user program and switch to kernel stack
- 3) Identify interrupt type and invoke appropriate interrupt handler
- 4) Restore user program

Roadmap is nearly identical for syscalls and exceptions!!

Processes

- Instances of a running program
 - Process Control Block (PCB) in OS that stores necessary metadata such as process state, PID, open files list, memory limits, PTBR (1 page table per process), etc.
 - An executing program with restricted rights
- OS is in charge of:
 - Giving every process the illusion of running on a private CPU
 - Giving every process the illusion of running on private memory
 - Managing resources to allocate to each process
 - Isolating processes from all other processes and protect OS

Process Management API

- **exit** – terminate a process 
- **fork** – copy the current process  → 
- **exec** – change the program being run by the current process
- **wait** – wait for a process to finish 
- **kill** – send a signal (interrupt-like notification) to another process
- **sigaction** – set handlers for signals

Not an exhaustive list, but some of the more common ones



fork()

- Creates a new child process, a copy of the original parent process
 - Allocate a new PCB for the new process with duplicated memory context
- Makes complete copy of process state:
 - Address Space, Code/Data Segments, Registers, Stack, Open Files, Pointers to Files/IO
 - Uses **copy-on-write** to make duplication more efficient
- Allocates new PID and saves the same EIP as the parent process
- OS can schedule either child or parent process
- To distinguish, returns 0 for child and > 0 for parent



fork() ex.

- ```
int void main() {
 int data = 3;
 int success = fork();
 if (success == -1) {
 return 1;
 } else if (success == 0) {
 // Execute child logic
 } else {
 int child_pid = success; // pid > 0
 // Execute parent logic
 }
 return 0;
}
```

# exec()

- Call to Unix exec() replaces running program!
- exec() System Call handler will:
  - 1) Replace the code and data segment
  - 2) Set EIP to point to start of new program/reinitialize SP and FP
  - 3) Push arguments to program onto stack.

# sigaction()

- Signal: very short message that may be sent to a process or a group of processes. (ex. SIGKILL, SIGTSTOP, SIGCONT, ... see HW 2)
  - Each process has unique ID (pid) and a shared group id (pgid)
  - Used to make process aware that specific event has occurred
  - Allow process to execute a signal handler function when event has occurred
- Each signal has a default action
  - Can use sigaction() to override default action with signal handler
  - When signal is caught, program jumps to signal handler function
  - Control of the program resumes at the previously interrupted instructions

# Unix I/O and Files

- Uniformity
  - Same set of system calls open, read, write, close
- Explicit Open Before Use, Close After Use
  - Must explicitly open file/device/channel
  - Must explicitly close resource (or memory is leaked)
- Kernel Buffered Reads/Writes
  - Data is buffered in kernel to decouple internals from application

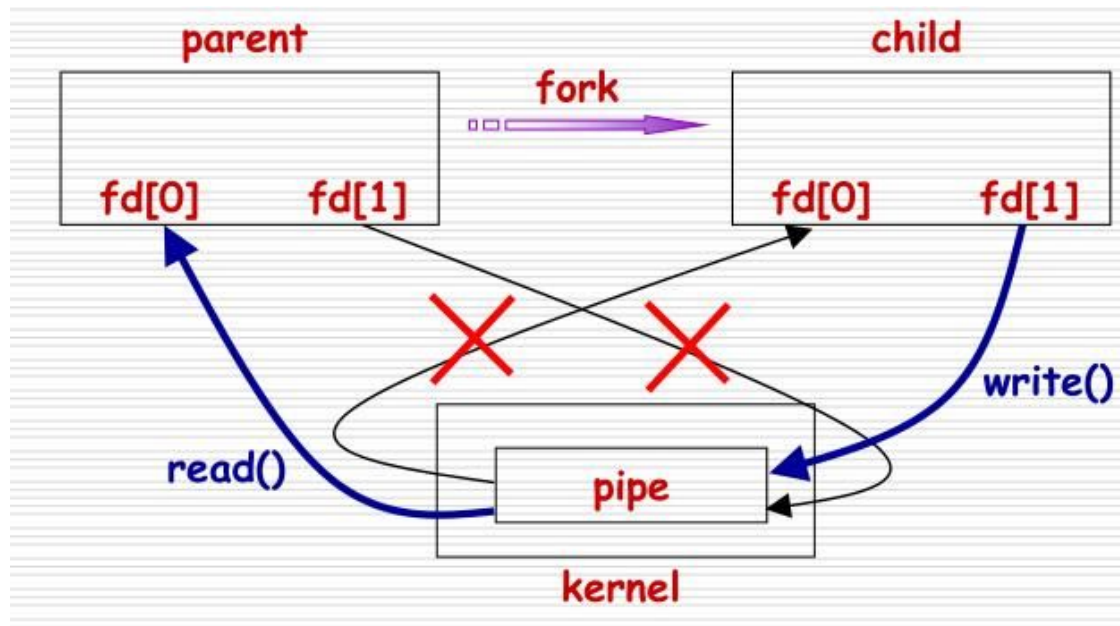


# File Descriptors

- Integer  $\geq 0$  that uniquely identifies an open IO resource in the OS
- Used to index into a per-process file descriptor table
- Each FD points to an open file description in a global table of open file descriptions
- Each file description in the table contains:
  - File offset
  - File access mode (from `open()`)
  - File status flags (from `open()`)
    - `ORD_ONLY`, `OWR_ONLY`, `O_RDWR` (see **man open**)
  - Reference to physical location
  - Number of times opened
- Can duplicate an FD to have multiple FDs point to the same file (**dup**)
  - Open file description remains alive until no file descriptors refer to it
  - **dup2** - can replace an already-existing fd

# Pipes

- `int pipe(int pipedfd[2])`
- One-way communication between two processes on the same physical machine
- **Read-only** end (`pipedfd[0]`) and **write-only** end (`pipedfd[1]`)
- Limited buffer space - Blocks if **write** is called on a full pipe. Blocks if **read** is called on empty pipe.
- After last “write” descriptor is closed, pipe is effectively closed and reads return only “EOF”



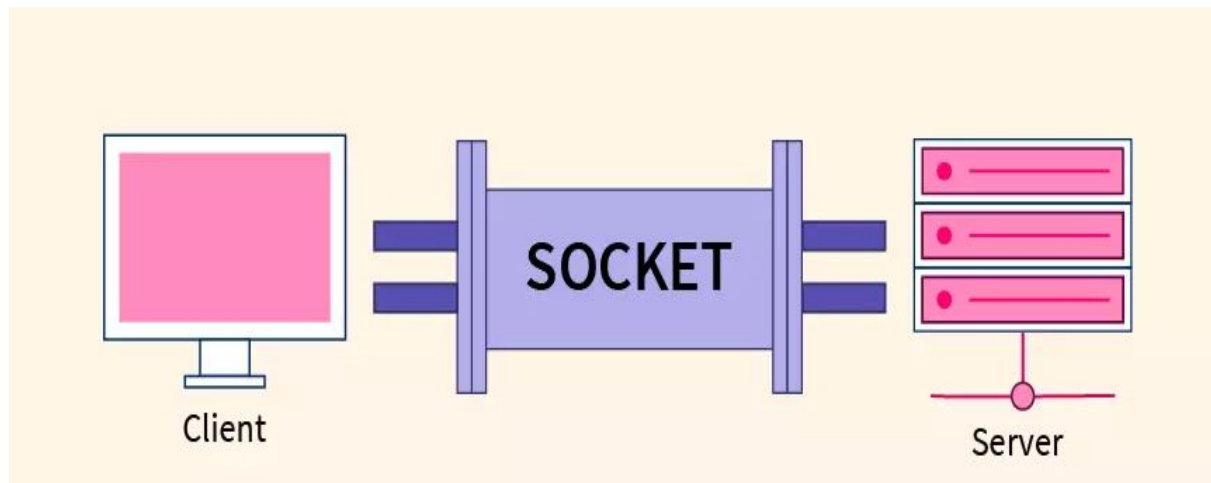
```

int fds[2];
pipe(fds); // Validation checking omitted for brevity
if (fork() != 0) {
 close(fds[1]); // Parent is read-only
 read(fds[0], outbuf, n); // Parent reads from pipe to outbuf
} else {
 close(fds[0]); // Child is write-only
 write(fds[1], inbuf, n); // Child writes inbuf to pipe
}

```

# Sockets

- Abstraction of two queues, one in each direction
- Can read or write to either end
- Used for communication between multiple processes on different machines
- File descriptors obtained via `socket/bind/connect/listen/accept`
- Still a file and uses same API/data structures as files and pipes



# From FDs to FILEs

- Instead of using integer file descriptors to describe files and relying on low-level I/O syscalls, high-level I/O API use “FILE” structures from the OS library
- FILE\* is an OS Library wrapper for manipulating **only** files, not pipes, sockets, etc.
- This FILE structure contains:
  - File descriptor (from call to open)
  - Buffer (array)
  - Lock (in case multiple threads use the FILE concurrently)
- FILE\* API operates on streams – unformatted sequences of bytes (text or binary data), with a position

`int fd = open(...);` vs `FILE* fptr = fopen(...);`

# Low-Level I/O API

- Low-level direct use of syscall interface:  
`open(), read(), write(), close()`
- Opening of file returns file descriptor:  
`int myfile = open(...);`
- File descriptor only meaningful to kernel
  - Index into process (PDB) which holds pointers to kernel-level structure (“file description”) describing file.
- Every `read()` or `write()` causes syscall no matter how small (could read a single byte)
- Consider loop to get 4 bytes at a time using `read()`:
  - Each iteration enters kernel for 4 bytes.

# High-Level I/O API

- High-level buffered access:  
`fopen(), fread(), fwrite(), fclose()`
- Opening of file returns ptr to FILE:  
`FILE *myfile = fopen(...);`
- FILE structure in user space contains:
  - a chunk of memory for a buffer
  - the file descriptor for the file (`fopen()` will call `open()` automatically)
- Every `fread()` or `fwrite()` filters through buffer and may not call `read()` or `write()` on every call.
- Consider loop to get 4 bytes at a time using `fread()`:
  - First call to `fread()` calls `read()` for block of bytes (say 1024). Puts in buffer and returns first 4 to user.
  - Subsequent `fread()` grab bytes from buffer

# FD API vs FILE\* API

- FILE\* is Buffered IO:
  - Maintains a per-file user-level buffer.
  - Write calls write to buffer.
  - System flushes buffer to disk when full (or on special character)
  - Read calls read from buffer. System reads from disk when buffer empty
- FD API is Immediate I/O:
  - Operations on file descriptors are unbuffered & visible immediately
- Biggest contrast is FD I/O immediately performs I/O whereas FILE\* operates on kernel buffer first
- FILE\* API calls don't have to go to disk as much as FD API!
  - Kernel just reads fixed size block from disk & buffer into user-space

# Threads

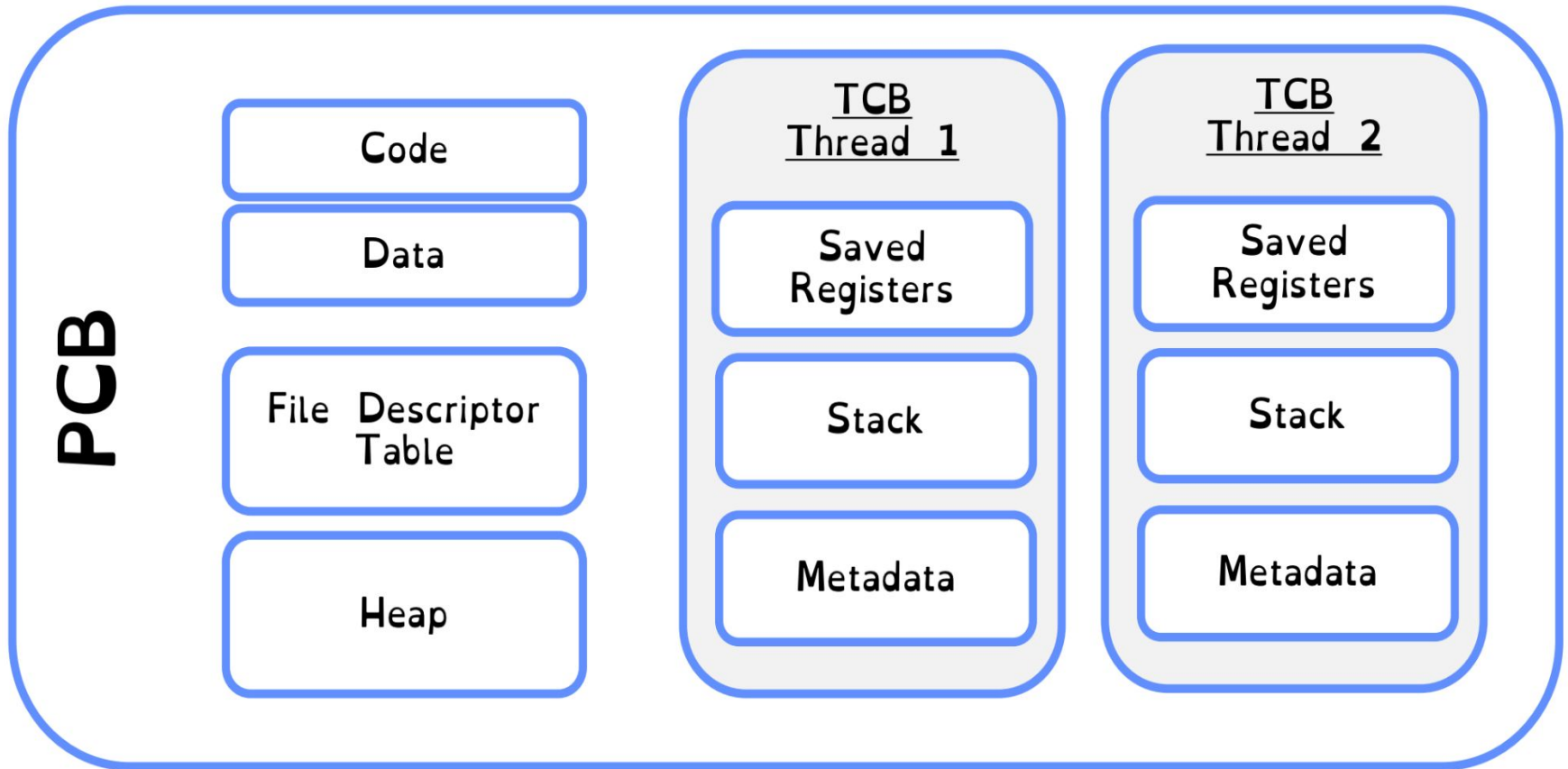
- A thread is a single execution sequence that represents a separately schedulable task
- Threads are really useful for:
  - Natural Program Structure:
    - Simultaneously update screen, fetch new data from network, receive keyboard input
  - Exploiting parallelism:
    - Split unit of work into  $n$  tasks and process tasks in parallel on multiple cores.
  - Responsiveness:
    - High priority work should not be delayed by low priority work. Schedule as separate threads for independence
  - Masking IO latency:
    - Continue to do useful work on separate thread while blocked on IO



# Thread Qualities

- No protection:
  - Threads inside the same process and are not isolated from each other
  - Share an address space
  - Share IO state (FDs)
- Individual execution:
  - Threads execute disjoint instruction streams.
  - Need own execution context
  - Given individual stack and register state (including EIP, ESP, EBP)

# Threads vs Processes Visual



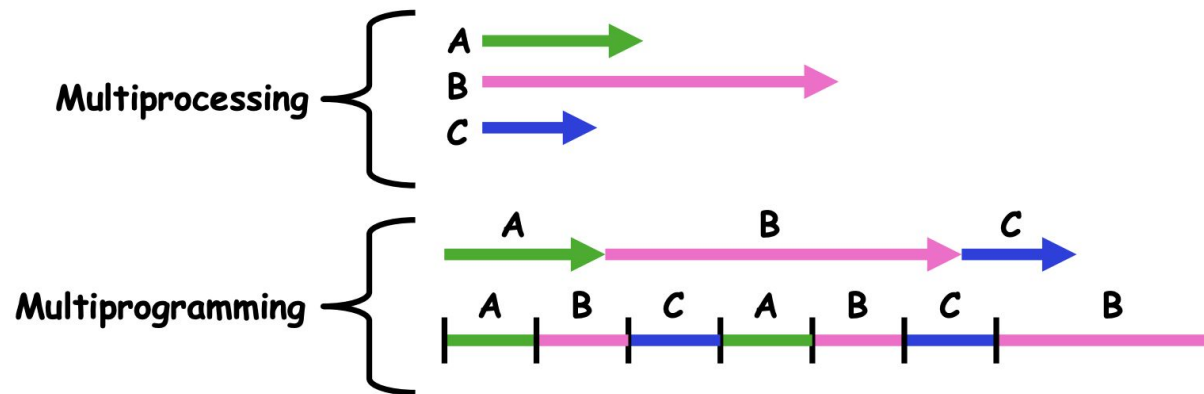
Processes are like the containers in which threads execute

# Multi-Everything

Multiprocessing  $\equiv$  Multiple CPUs

Multiprogramming  $\equiv$  Multiple Jobs or Processes

Multithreading  $\equiv$  Multiple Threads per Process



Multiple threads running “concurrently” can be scheduled in any order the OS wants

But threads can end up racing for shared data in the process

# Stacks for processes

```
void main() {
 int data = 0;
 int x = 1;
 pid_t pid = fork();
 if (pid != 0) {
 data = 1;
 printf("data is %d, &x is %x\n", data, &x);
 waitpid(pid, NULL, 0);
 } else {
 data = 2;
 printf("data is %d, &x is %x\n", data, &x);
 exit(0);
 }
}
```

What does this code print?

// Assume &x = DEADBEEF

# Stacks for processes (soln.)

```
void main() {
 int data = 0;
 int x = 1;
 pid_t pid = fork();
 if (pid != 0) {
 data = 1;
 printf("data is %d, &x is %x\n", data, &x);
 waitpid(pid, NULL, 0);
 } else {
 data = 2;
 printf("data is %d, &x is %x\n", data, &x);
 exit(0);
 }
}
```

data is 1, &x is DEADBEEF  
data is 2, &x is DEADBEEF

(either order is possible)

# Stacks for Threads

```
void helper (void * arg) {
 int *data = (int *) arg;
 *data = 1;
 printf("*data is %d, data is %x\n", *data, data);
 pthread_exit(0);
}

void main() {
 pthread_t thread;
 int data = 0;
 printf("data is %d, &data is %x\n", data, &data);
 pthread_create(&thread, NULL, &helper, &data);
 pthread_join(thread, NULL);
 printf("data is %d, &data is %x\n", data, &data);
}
```

What does this program print? Assume `&data = 0xDEADBEEF`.

# Stacks for threads (soln.)

```
void helper (void * arg) {
 int *data = (int *) arg;
 *data = 1;
 printf("*data is %d, data is %x\n", *data, data);
 pthread_exit(0);
}
void main() {
 pthread_t thread;
 int data = 0;
 printf("data is %d, &data is %x\n", data, &data);
 pthread_create(&thread, NULL, &helper, &data);
 pthread_join(thread, NULL);
 printf("data is %d, &data is %x\n", data, &data);
}
```

data is 0, &data is 0xDEADBEEF (main)  
\*data is 1, data is 0xDEADBEEF (helper)  
data is 1, &data is 0xDEADBEEF (main)

# Heaps for processes

```
void main() {
 char *x = malloc(100);
 strcpy(x, "temporary");
 pid_t pid = fork();
 if (pid != 0) {
 strcpy(x, "hello1");
 printf("%s address of x is %x\n", x, &x);
 waitpid(pid, NULL, 0);
 } else {
 strcpy(x, "hello2");
 printf("%s address of x is %x\n", x, &x);
 exit(0);
 }
}
```

What does this program print? Assume `&x = 0xDEADBEEF`.



# Heaps for processes (soln.)

```
void main() {
 char *x = malloc(100);
 strcpy(x, "temporary");
 pid_t pid = fork();
 if (pid != 0) {
 strcpy(x, "hello1");
 printf("%s address of x is %x\n", x, &x);
 waitpid(pid, NULL, 0);
 } else {
 strcpy(x, "hello2");
 printf("%s address of x is %x\n", x, &x);
 exit(0);
 }
}
```

hello2 address of x is 0xDEADBEEF  
hello1 address of x is 0xDEADBEEF

(either order is possible)

# Heap for threads

```
void helper(void * arg) {
 char *x = (char *) arg; // on the stack
 strcpy(x, "hello2");
 printf("%s address of x is %d\n", x, &x);
}
```

What does this program print?  
Assume &x = 0xDEADBEEF.

```
void main() {
 pthread_t thread;
 char *x = malloc(100); // on the heap
 strcpy(x, "hello1");
 printf("%s address of x is %d\n", x, &x);
 pthread_create(&thread, NULL, &helper, x);
 pthread_join(thread, NULL);
 printf("%s address of x is %d\n", x, &x);
}
```

# Heap for threads (soln.)

```
void helper(void * arg) {
 char *x = (char *) arg; // on the stack
 strcpy(x, "hello2");
 printf("%s address of x is %d\n", x, &x);
}
void main() {
 pthread_t thread;
 char *x = malloc(100); // on the heap
 strcpy(x, "hello1");
 printf("%s address of x is %d\n", x, &x);
 pthread_create(&thread, NULL, &helper, x);
 pthread_join(thread, NULL);
 printf("%s address of x is %d\n", x, &x);
}
```

hello1 address of x is 0xDEADBEEF (main)  
hello2 address of x is 0xDEADBEEF (helper)  
hello2 address of x is 0xDEADBEEF (main)

# File descriptors for processes

```
int fd;
void main() {
 fd = open("output", O_CREAT|O_WRONLY);
 pid_t pid = fork();
 if (pid != 0) {

 waitpid(pid, NULL, 0);
 printf("close fd %d returns %d\n", fd, close(fd));
 } else {
 printf("close fd %d returns %d\n", fd, close(fd));
 exit(0);
 }
}
```

What does this program print? Assume `fd == 3` after `open()` and `close(fd)` returns a close status (0 for success, -1 otherwise).

# File descriptors for processes (soln.)

```
int fd;
void main() {
 fd = open("output", O_CREAT|O_WRONLY);
 pid_t pid = fork();
 if (pid != 0) {

 waitpid(pid, NULL, 0);
 printf("close fd %d returns %d\n", fd, close(fd));
 } else {
 printf("close fd %d returns %d\n", fd, close(fd));
 exit(0);
 }
}
```

**close fd 3 returns 0** (child closes its copy first)

**close fd 3 returns 0** (parent then closes its fd)

Child process gets copies of parent's fd's! File isn't closed until all fd's for it are closed!

# File descriptors for threads

```
int fd;

void *helper (void *arg) {
 printf("close fd %d returns %d\n", fd, close(fd));
 pthread_exit(0);
}

void main() {
 fd = open("output", O_CREAT|O_TRUNC|O_WRONLY);
 pthread_t thread;
 pthread_create(&thread, NULL, &helper, NULL);
 pthread_join(thread, NULL);
 printf("close fd %d returns %d\n", fd, close(fd));
}
```

What does this program print? Assume `fd == 3` after `open()` and `close(fd)` returns a close status (0 for success, -1 otherwise).

# File descriptors for threads (soln.)

```
int fd;

void *helper (void *arg) {
 printf("close fd %d returns %d\n", fd, close(fd));
 pthread_exit(0);
}

void main() {
 fd = open("output", O_CREAT|O_TRUNC|O_WRONLY);
 pthread_t thread;
 pthread_create(&thread, NULL, &helper, NULL);
 pthread_join(thread, NULL);
 printf("close fd %d returns %d\n", fd, close(fd));
}

close fd 3 returns 0 (child closes fd first)
close fd 3 returns -1 (parent can't close it anymore)
Child threads share the same address space and don't
copy the parent thread's fd's!
```

# Non-determinism

(2 points) Which of the following could be the output of this program? Select **all** choices that correspond to possible results.

```
char* buffer;
void* ashitaka(void* arg) {
 buffer = "mononoke";
 printf("%s", buffer);
}
int main(int argc, char** argv) {
 pthread_t t;
 pthread_create(&t, NULL, ashitaka, NULL);
 buffer = "princess";
 printf("%s", buffer);
}
```

- ☐ princessmononoke    ☐ mononokeprincess    ☐ princessprincess  
☐ mononokemononoke    ☐ princess    ☐ mononoke    ☐ (program outputs nothing)



# Non-determinism (soln.)

(2 points) Which of the following could be the output of this program? Select **all** choices that correspond to possible results.

```
char* buffer;
void* ashitaka(void* arg) {
 buffer = "mononoke";
 printf("%s", buffer);
}
int main(int argc, char** argv) {
 pthread_t t;
 pthread_create(&t, NULL, ashitaka, NULL);
 buffer = "princess";
 printf("%s", buffer);
}
```

✓ princessmononoke    ✓ mononokeprincess    ✓ princessprincess  
✓ mononokemononoke    ✓ princess    ✓ mononoke    ☐ (program outputs nothing)

# Non-determinism (soln.)

- **princessmononoke**
  - The main thread sets `buffer = "princess"` and prints, then child thread sets `buffer = "mononoke"` and prints.
- **mononokeprincess**
  - The child thread sets `buffer = "mononoke"` and prints, then the main thread sets `buffer = "princess"` and prints.
- **princessprincess**
  - The child thread first sets `buffer = "mononoke"`. Then, the main thread sets `buffer = "princess"`. Then both threads print.
- **mononokemononoke**
  - The main thread first sets `buffer = "princess"`. Then, the child thread sets `buffer = "mononoke"`. Then both threads print.
- **princess**
  - The main thread sets `buffer = "princess"` and prints, and exits without giving the child thread the chance to execute.
- **mononoke**
  - The main thread sets `buffer = "princess"`. The child thread then sets `buffer = "mononoke"`. The main thread then prints and exits without giving the child a chance to continue.

## For Your Cheatsheet

| Comparison Categ.              | Across Processes          | Across Threads                       |
|--------------------------------|---------------------------|--------------------------------------|
| Creation                       | fork()                    | pthread_create()                     |
| Page table                     | Distinct                  | Same                                 |
| Registers, instruction pointer | Distinct                  | Distinct                             |
| Stack                          | Separate and inaccessible | Separate, but accessible             |
| Heap and static variables      | Separate                  | Shared                               |
| File descriptors               | Separate                  | Shared                               |
| Synchronization                | wait(), waitpid()         | pthread_join(),<br>semaphores, locks |
| Overhead                       | Higher                    | Lower                                |
| Protection                     | Higher                    | Lower                                |

# Concurrency Terminology

- Atomic Operation: An operation that always runs to completion or not at all
  - On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Synchronization:
  - Using atomic operations to ensure cooperation between threads
- Mutual Exclusion:
  - Ensuring that only one thread does a particular thing at a time
- Critical Section:
  - Piece of code that only one thread can execute at once

# Solution to Race Conditions

- Need to protect “critical sections”
- Lock() before entering critical section and before accessing shared data
- Unlock() when leaving, after accessing shared data
- Wait if locked
- Important idea: All synchronization involves waiting
- A few of ways to “lock” or synchronize a critical section:
  - Busy Wait (just rely on atomic loads and stores)
  - Use synchronization primitives that disable/enable interrupts (powerful, but could be dangerous if not done correctly)

# Busy Wait

While thread B is operating the refrigerator, thread A will continue to execute its while loop and consume CPU cycles until OS switches back to thread B – very inefficient! Don't do this on an exam!

## Thread A

```
leave note A;
while (note B) {\X
 do nothing;
}
if (noMilk) {
 buy milk;
}
remove note A;
```

## Thread B

```
leave note B;
if (noNote A) {\Y
 if (noMilk) {
 buy milk;
 }
}
remove note B;
```

Atomic loads and stores work, but bad on the CPU

# test&set() Lock Implementation

```
test&set (&address) {
 result = M[address];
 M[address] = 1;
 return result;
}

int mylock = 0; // Interface: acquire(&mylock);
 // release(&mylock);

acquire(int *thelock) {
 while (test&set(thelock)); // Atomic operation!
}

release(int *thelock) {
 *thelock = 0; // Atomic operation!
}
```

} Atomic Operations

# Disable & Enable Interrupts

```
LockAcquire { disable Ints; }
```

```
LockRelease { enable Ints; }
```

You need to be really careful disabling interrupts

What if you do this... 😬

```
LockAcquire ();
While (TRUE) { ; }
```



# Futex Solution

```
#include <linux/futex.h>
```

```
#include <sys/time.h>
```

```
int futex(int *uaddr, int futex_op, int val,
 const struct timespec *timeout);
```

- *uaddr* points to a 32-bit value in user space
- *futex\_op*
  - **FUTEX\_WAIT:**
    - if (*val* == \**uaddr*) → go to sleep until we hear a **FUTEX\_WAKE** operation
    - Atomic check that condition still holds after we disable interrupts
  - **FUTEX\_WAKE:**
    - Operation that wakes up at most *val* waiting threads

# Futex Applied

```
int mylock = 0;
```

```
acquire(int *thelock) {
 while (test&set(thelock)) {
 futex(thelock, FUTEX_WAIT, 1);
 }
}
```

Just put thread to sleep  
No more busy waiting!

```
release(int *thelock) {
 thelock = 0; // unlock
 futex(&thelock, FUTEX_WAKE, 1);
}
```

Wakes up **val** sleeping  
threads, even if there is none

# Semaphores

- Semaphores are like locks, but rely on the state of a non-negative integer value instead of a boolean/binary state
- When you initialize a semaphore, you initialize it with some non-negative integer value
- Semaphore will put a thread to sleep when it tries `down()` on a semaphore value of 0
- Multi-Purpose:
  - **Mutual Exclusion** (value is 0 or 1, essentially imitating a lock)
  - **Scheduling Constraints** (value can be  $\geq 0$ , used to schedule threads based on the constraints of your program, think of player/server problem from discussion!)


# Semaphore Operations

```
sema_init (semaphore* sema, value) {
 Initialize semaphore value to
 requested non-negative value;
}
```

```
sema_down (semaphore* sema) {
 Disable interrupts;
 While (semaphore value == 0) {
 Go to sleep and
 schedule another thread;
 }
 Decrement semaphore value by 1;
 Enable interrupts;
}
```

```
sema_up (semaphore* sema) {
 Disable interrupts;
 Wake up a sleeping thread;
 Increment semaphore value by 1;
 Enable interrupts;
}
```

```
typedef struct semaphore {
 unsigned value;
 struct list waiters;
} semaphore;
```

 When a thread gets put to sleep, the OS will re-enable interrupts as it schedules another thread to run!

# Locks

- Locks can also be quite neatly implemented using internal semaphores
- We would initialize the lock's semaphore to 1 and keep track of who acquires the lock
- Call `sema_down()` within `lock_acquire()` and `sema_up()` within `lock_release()`
- In this way, the lock's semaphore value will only alternate between 0 and 1
- Purpose: used so that threads can access/modify shared data synchronously

# Lock Operations

```
typedef struct lock {
 struct thread* holder;
 struct semaphore* sema;
} lock;

lock_init (lock* lock) {
 Initialize the holder of this
 lock to NULL;
 sema_init(&lock->sema, 1);
}

lock_acquire (lock* lock) {
 sema_down(&lock->sema);
 Set the holder of this lock
 to caller;
}

lock_release (lock* lock) {
 Reset lock holder back to NULL;
 sema_up(&lock->sema);
}
```

The internal semaphore operations are atomic, shown in the previous slide! Therefore, the actual acquisition of the lock relies on the state of the internal semaphore.

# Lock Operations (Spin-Locking)

```
typedef struct lock {
 struct thread* holder;
 int value;
} lock;

lock_init (lock* lock) {
 Initialize the holder of this
 lock to NULL;
 lock->value = 0;
}

lock_acquire (lock* lock) {
 while (test&set(&lock->value) == 1){
 // Busy waiting
 }
 Set the holder of this lock
 to caller;
}

lock_release (lock* lock) {
 Reset lock holder back to NULL;
 lock->value = 0;
}

test&set (void* address) {
 int result = *address;
 *address = 1;
 return result;
}
```

# Conditional Variables

- A queue of threads waiting for something inside a critical section
- In contrast to semaphores, conditional variables can allow sleeping inside the critical section by atomically releasing lock at time we go to sleep
- Require a lock to be held when performing certain conditional variable operations
- Purpose: used so that threads can access/modify shared data when a certain condition is met
- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data



# Conditional Variable Operations

```
typedef struct cond_var {
 struct list waiters;
} cond_var;
```

```
cond_init (cond_var* cond) {
 Initialize cond's list
 of waiters;
}
```

```
cond_wait (cond_var* cond, lock* lock) {
 Create a semaphore for this waiter;
 sema_init(&waiter_sema, 0);
 Push this waiter's semaphore
 into the waiters queue;
 lock_release(lock);
 sema_down(&waiter_sema);
 lock_acquire(lock);
}
```

Must have acquired the lock before you perform `cond_wait()`! Notice that we release the lock first in order to allow other threads to finish and then reacquire when we've woken up.

# Cond Signaling / Broadcasting To Waiters

```
cond_signal (cond_var* cond) {
 if (waiters queue is not empty) {
 Pop a waiter off and execute
 sema_up() on their semaphore;
 }
}
```

```
cond_broadcast (cond_var* cond) {
 while (waiters queue is not empty) {
 Pop a waiter off and execute
 sema_up() on their semaphore;
 }
}
```

`cond_signal()` will wake up only 1 waiting thread whereas `cond_broadcast()` will wake up all waiting threads!

`cond_broadcast()` can therefore be made up of several calls to `cond_signal()`.

Must acquire the lock before these operations and release afterwards appropriately in order ensure safe access to the shared data as threads get scheduled!

# Mesa vs Hoare Semantics

- How can we actually use conditional variables in practice?
- How do we know when to sleep or signal and how should we check our required conditions?
- Mesa and Hoare semantics offer two ways to implement a monitor

Let's assume our example from lecture and discussion:

Consider an infinite synchronized buffer problem of vending machines where there's a producer and consumer. "Infinite" refers to the fact that the machine has no limit on how much coke it can hold.

```
struct lock buffer_lock;
```

```
struct cond_var buffer_cond;
```

```
struct list coke_machine;
```

```
// Assume we initialized all of these structures
```

# Hoare Semantic

```
void producer(struct coke* coke) {
 lock_acquire(&buffer_lock);
 Insert coke into coke_machine;
 cond_signal(&buffer_cond);
 lock_release(&buffer_lock);
}
```

```
struct coke* consumer() {
 lock_acquire(&buffer_lock);

 if (list_empty(&coke_machine))
 cond_wait(&buffer_cond, &buffer_lock);

 Pop a coke from the coke_machine;
 lock_release(&buffer_lock);
 return coke;
}
```

Only does one check on the `coke_machine`'s status. Once the thread is awoken, it can proceed with the lock if it's available and try to take a coke out of the `coke_machine`.

But are you absolutely sure the `coke_machine` still has coke by the time the thread tries to pop one? 🤔

# Hoare Semantic

- When a waiting thread gets woken up, it's not guaranteed that it will be scheduled by the OS at a time that satisfies our condition
- Example Scenario (Assume `coke_machine` initially empty and some initial thread creates Thread A and Thread B):
  - Thread A runs `consumer()` and sleeps because the `coke_machine` is empty
  - Thread B runs `producer()`, puts 1 coke in the empty `coke_machine`, and wakes up Thread A
  - Before Thread A tries to re-acquire the `buffer_lock`, thread C gets created and runs `consumer()` to completion, emptying out the `coke_machine`
  - Our poor thread A will then be scheduled next only to find that it's trying to pop coke out of an empty `coke_machine` 😞
- For Hoare's semantic to be reliable for our system, it really depends on the OS scheduler

# Mesa Semantic

```
void producer(struct coke* coke) {
 lock_acquire(&buffer_lock);
 Insert coke into coke_machine;
 cond_signal(&buffer_cond);
 lock_release(&buffer_lock);
}

struct coke* consumer() {
 lock_acquire(&buffer_lock);
 while (list_empty(&coke_machine))
 cond_wait(&buffer_cond, &buffer_lock);
 Pop a coke from the coke_machine;
 lock_release(&buffer_lock);
 return coke;
}
```

In contrast to Hoare, this semantic will repeatedly check our `coke_machine` status to make sure that we definitely have coke available when we have the lock and try to pop a coke.

This semantic is much easier and safer to implement!

# Old Bridge Problem

An old bridge has only **one lane** and can only hold at most **3 cars** at a time without risking collapse. Traffic goes **both ways**.

- Fill in the functions `ArriveBridge(int direction)` and `ExitBridge()` so that at any given time, there are at most 3 cars on the bridge, and all of them are going the same direction.
- A car calls `ArriveBridge()` when it arrives at the bridge and wants to go in the specified direction (0 for left or 1 for right); `ArriveBridge()` should not return until the car is allowed to get on the bridge.
- A car calls `ExitBridge()` when it gets off the bridge, and allows other cars to get on.
- Whenever the bridge is not empty or full and a car is waiting to go the same direction as the cars on the bridge, that car should get on the bridge.
  - One direction of cars gets to flush out completely before the other direction is allowed. Don't worry about starvation/fairness.

# Example Picture of Scenario





# Starvation Explanation



In this case, all the waiting cars on the left will be allowed to cross the bridge before we allow the waiting car on the right to cross

# Old Bridge Problem Details

- We will be using the following data structures:
  - `struct cond_var cond_directions[2];`
    - Array of 2 conditional variables that represent a waiting queue for each direction
  - `struct lock cond_lock;`
    - A lock that you may use to synchronize access to necessary data
  - `int waiters[2];`
    - Array of 2 integers that track how many cars are waiting to travel in either direction
  - `int cars_on_bridge = 0;`
    - Number of cars currently on the bridge
  - `int curr_direction = 0;`
    - Direction of the cars that are currently on the bridge (default is 0)
- Assume we've initialized the conditional variables in `cond_directions` and filled `waiters` with 0s for each direction

# ArriveBridge() - Skeleton

```
// Remember, you can use struct cond_var cond_directions[2], int waiters[2],
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ArriveBridge(int direction) {

 while (cars_on_bridge == 3 ||
 (cars_on_bridge > 0 && curr_direction != direction)) {

 }

}
```

## ArriveBridge() - Soln.

```
// Remember, you can use struct cond_var cond_directions[2], int waiters[2],
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ArriveBridge(int direction) {

 while (cars_on_bridge == 3 ||
 (cars_on_bridge > 0 && curr_direction != direction)) {
 waiters[direction]++;
 cond_wait(&cond_directions[direction], &cond_lock);
 waiters[direction]--;
 }

}
```

## ArriveBridge() - Soln.

```
// Remember, you can use struct cond_var cond_directions[2], int waiters[2],
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ArriveBridge(int direction) {

 while (cars_on_bridge == 3 ||
 (cars_on_bridge > 0 && curr_direction != direction)) {
 waiters[direction]++;
 cond_wait(&cond_directions[direction], &cond_lock);
 waiters[direction]--;
 }

 cars_on_bridge++;
 curr_direction = direction;

}
```

## ArriveBridge() - Soln.

```
// Remember, you can use struct cond_var cond_directions[2], int waiters[2],
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ArriveBridge(int direction) {
 lock_acquire(&cond_lock);
 while (cars_on_bridge == 3 ||
 (cars_on_bridge > 0 && curr_direction != direction)) {
 waiters[direction]++;
 cond_wait(&cond_directions[direction], &cond_lock);
 waiters[direction]--;
 }
 cars_on_bridge++;
 curr_direction = direction;
 lock_release(&cond_lock);
}
```

# ExitBridge() - Skeleton

```
// Remember, you can use struct cond_var cond_directions[2], int
waiters[2],
```

```
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ExitBridge() {
```

```

```

```

```

```
 If (waiters[curr_direction] > 0) {
```

```

```

```
 } else if (cars_on_bridge == 0) {
```

```

```

```
 }
```

```

```

```
}
```

# ExitBridge() - Soln.

```
// Remember, you can use struct cond_var cond_directions[2], int
waiters[2],
```

```
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ExitBridge() {
```

```


```

```
 If (waiters[curr_direction] > 0) {
```

```
 cond_signal(&cond_directions[curr_direction]);
```

```
 } else if (cars_on_bridge == 0) {
```

```
 cond_broadcast(&cond_directions[1 - curr_direction]);
```

```
 }
```

```

```

```
}
```



# ExitBridge() - Soln.

```
// Remember, you can use struct cond_var cond_directions[2], int
waiters[2],
```

```
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ExitBridge() {
```

---

```
cars_on_bridge -= 1;
```

```
If (waiters[curr_direction] > 0) {
```

```
 cond_signal(&cond_directions[curr_direction]);
```

```
} else if (cars_on_bridge == 0) {
```

```
 cond_broadcast(&cond_directions[1 - curr_direction]);
```

```
}
```

---

```
}
```

# ExitBridge() Soln.

```
// Remember, you can use struct cond_var cond_directions[2], int
waiters[2],
```

```
// struct lock cond_lock, int curr_direction, int cars_on_bridge
```

```
int ExitBridge() {
 lock_acquire(&cond_lock);
 cars_on_bridge -= 1;
 If (waiters[curr_direction] > 0) {
 cond_signal(&cond_directions[curr_direction]);
 } else if (cars_on_bridge == 0) {
 cond_broadcast(&cond_directions[1 - curr_direction]);
 }
 lock_release(&cond_lock);
}
```

*Thank you!*