Threads

September 9th

Why processes
- Sharing
- isolation
- concurrency

Threads act like processes (light weight process)

Concurrency vs Parallel
└ switches tasks rapidly

Runs processes at same time

Threads

Kernel threads - kernel manages process threads
User threads - created and managed by user

User mode benefits
- Super fast, full scheduler controller
- Cons - Safety concerns, no parallelism

You cannot predict how long for a process to run
why? Multiple other things running + the scheduler may not prioritize your request

Creating Threads in C

```
pthread_t t1, t2
pthread_create(&t1, NULL, threadfunc(NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

- Causes Race condition - Depending which thread runs which code changes outputs

Mutex - locks

```
pthread_mutex_t thelock = PTHREAD_MUTEX_INITIALIZER
for(i=0; i<5; i++) {
    pthread_mutex_lock(&thelock);
    i++;
    pthread_mutex_lock(&theLock); }
```

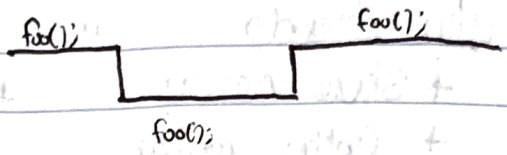
← Called critical section

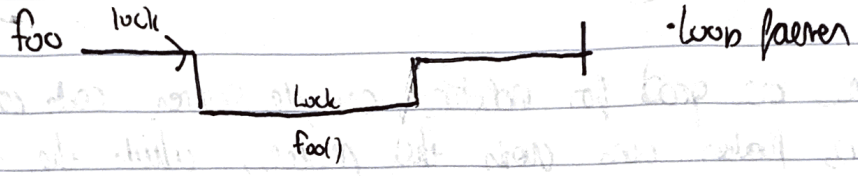
Extremely slow increase a loop

In order to cancel a thread (pthread_cancel)
the thread has to be in a cancel state (sleep is a cancel state)
* Amdahl's Law is important

4hr 40 min

Thread Safety - if you can run code in multithreads and it works then threadsafe
Reentrant function can interrupt foo() at any point then return foo() returns, then resume the original foo(), its threadsafe reentrant





Printf calls on malloc probably contain some information to know that the memory came from printf
→ Printf("Name");
maybe memory has "name" inside of it its allocated memory since we store our pointers to malloc

OOSP CH 3.1-3.2

Create Process
- Initialize process control block
- Initialize new address space
- Load program into address space
- Copy arguments into memory in address space
- Initialize hardware context to start execution of start
- Inform scheduler process is ready

Unix Exec
- Load program into current address space
- Copy arguments into memory address space
- Initialize hardware context to begin at start

cpu pairing Algorithm

PID	ARR (ARRIVAL TIME)	Length	Deadline
1	0	3	8
2	2	2	7
3	3	5	9
4	5	1	6

- Ratio of L/DL
- EDF (earliest deadline First)

Using EDF (4213)
- [can run into starvation if deadlines are too tight]

~~Execution Control~~
Least Slack
- How many Quanta can I waste while completing the task?

Chapter 4 Concurrency and Threads

4.1 Thread use cases
- Helpful for games when you're drawing graphics, receiving inputs, and updating player locations simultaneously

4.2 Thread Application
thread - a single execution sequence that represents a separately schedulable task
- Can run suspend or resume at any time

Threads are native mechanisms with unpredictable speed
thread_join is analogous to wait for fork and exec
thread_create does generally the same things fork does + exec.

4.4 Threads data structures and life cycle
threads have a per thread state and shared thread state
- Thread control block stores information about threads executing and meta data used to manage thread
each thread also has its own stack which is pointed to by TCB
- Thread meta data contains information such as thread ID, scheduling priority, and status

OS Proj 2
7-11

Thread data should be a linked list

Shared state - state shared between running threads
• Program code is shared between threads to a extent.
Information visible to each thread is the heap, and global variables

4.5 Thread Life Cycle
Initialize → Ready → wait → run → Finish
thread_create(), thread_join(), thread_yield(), thread_exit

Initialization places thread into init state, allocates per thread information, puts thread into ready state by placing thread onto ready priority Queue
Ready - At any time, the threads registers live on thread control block, can be taken off and copied to CPU registers once running
Running - thread is executing on cpu, registers are copied to CPU, can go back to ready by yielding, or copying registers back to thread control block, and switching cpu to run next thread

* A thread running is not on the ready list and vice versa
wait - thread is waiting for some event. Cannot run until some thread finishes and changes its state from waiting to running Thread is stored on waiting list of some synchronization variable awaited
FInished - Finished thread can never run again. OS can reclaim memory, but sometimes thread is placed on a finished waiting list. Once the exit value is read by thread_join, the thread is no longer used

4.6 Kernel Threads 3 Step creation
1 • Allocate per thread state 3. Put TCB onto ready list
2 • Initialize per thread state
- thread starts in dummy function, which then calls desired function, whose function returns, which without stub returns to whatever location at top of stack

Deleting Thread
- Remove thread from ready list
- Free per-thread state allocated for thread

* Interrupts could stop thread mid deletion and cause memory leak
* Thread cannot free its own state!
Instead, copy thread to finished state, one another thread has a moment clear the memory of threads in finished state

Thread Context Switch - Suspends current thread and resumes some other thread
- Saves current (running) thread registers to TCB and stack, restores new thread information to CPU registers

Caused by:
- Call to active thread library
- CPU interrupts or memory exception

Threads should not know a interrupt or memory exception occurred. To the thread time wont lag. Photo why thread running order is very unpredictable

Must disable interrupts during (context) switch or bugs can occur
* Page 62 contains useful explanation

Involuntary kernel Thread Context Switch
- Interrupts or traps signals kernel thread switch
- Save state
- Run kernel Handler
- Restore state

* Review worthwhile
Thread Safe Redflags
- Global shared memory
- Static values

4.7 Continued 4.8
Threads in a process contain
- User level stack
  - kernel interrupts stack for when thread comes interrupted
- kernel TCB for regs and settings per thread state

Scheduler activations - user level thread scheduler is notified for every kernel event that might affect user level threads
Asynchronous IO - much more practical in server applications

Chapter 5. Synchronizing Access to Shared Objects
Independent threads - threads operate on completely separate subsets of memory
Cooperating threads - read and write shared memory
Race condition - thread output depends on interleaving operations of different threads
Atomic operations - indivisible operations that cannot be interleaved or split by operations

The challenge with threads owning shared blobs of memory is it's extremely easy to modify something that should not have have been changed. Such as the too much milk problem. Someone checks the fridge, no milk, heads to the store another person sees the exact same thing, and buys milk. The solution is to throw a lock around the particular section handling it, then when you complete the task you pass it off to the next thread or DICA unlock it
Shared Objects - objects that can be safely owned by threads

5.2

Synchronization Variable - Data structure used for coordinating access to a shared state
- AKA locks and conditional variables

Locks 5.3
- Synchronization variable that provides mutual exclusion. When on thread holds a lock no other thread can.

Locks guard a shared objects state
- Generally a lock for each public method

Locks can be in one of two states
- Busy or free
- Lock is always initially free
- Aquire(); waits for lock to not be busy and by acquiring threads waits for lock to be released be continuation

Lock should ensure following properties
- At most one thread holds the lock
- if no thread has a lock, a thread calling acquire should eventually acquire the lock
- Bounded waiting - If T asks for the lock, then there is a bound on how many times a thread can grab the lock before T

5.3.2 Thread Safe Queues

Critical Section - area of shared memory that must be accessed atomically

- Threads should only share dynamically allocated memory

5.4 Condition Variables

- Provide a way for one thread to wait for another thread to take action
  - Very helpful when thread is waiting for a lock to open up.

wait(Lock* Lock) - releases lock of calling thread, and places thread on waitlist. When execution resumes is enabled, lock is owned lock before returning from wait

Review

Signal() - removes one thread and marks as ready to run to the scheduler. No effect if no threads are present

Broadcast() - Takes all threads of CV's waiting list and marks them as ready to run

* Page 130 is useful

- No threads can wait unless the thread has the lock in possession
- Wait must always be called inside a loop

Thread life cycle

- Running thread that calls wait is placed on waiting state
  - Copy TCB from running to waiting block
  - Calling signal copies TCB to ready state, and once the OS schedules the waiter then it runs

Generally you have as many conditional variables as unique ways a method must wait

Left off on page 139.

Project 2 Notes

User mode

There is cooperative and preemptive mode

Enable interrupts on autograder

Context Switch

- Save the state of a thread before switching

(get context)(First_context) • Save current context
set context (First_context) • Load current context
swap contexts (second, First) • Save one, load the other

* myContext.uc_stack.ss_sp = a_new_stack
* myContext.uc_stack.ss_size = STACK_SIZE

make context (&my_thread, void (*)(void) thread_func, 1, 1);

So, when I call
swap_context (&main_context, &my_thread);

- Saves current context into main_context and loads the current context of thread which points to the function thread_func

ThreadData should be a linked list and an array wrap function ptr into another function, that can swap the contexts when needed?

This is a linked list storing each threads contexts

Threads

Use Case:

Threads are useful when you want multiple things occurring at once. You want to render a scene while allowing the user to use a responsive main menu.
Or procedural generation while still allowing the player to move around

Thread Abstraction

- Basically threads act like mini virtual machines running simultaneously tasks. Although that's how they appear. Instead threads execute a sequence of instructions until a switch occurs. Then that thread is eventually "paused" while that thread is paused another one takes over and begins working on another section until it's switched off. Threads give the illusion of everything simultaneously being worked on

join, wait, yield

Thread API

join: Initializes a thread and joins it together with the desired function to run?
wait: Takes current running thread, saves context, lets next thread begin running. Once wait returns the saved thread resumes running
yield: thread gives up running on the machine to let another thread run. At any time the thread may be signaled to run again. But until then thread sits on waiting list

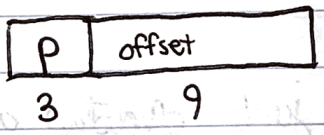
Thread yield
- Need idea of currently running thread
- Could have thread pointer, points at info that is known the thread that points to running thread

Memory

External Fragmentation - The memory being carved has a bunch of small memory pieces that are too small to satisfy the memory requests

Ways to avoid fragmentation
- First Fit - Find the hole that first fits
- Next Fit -
- Best Fit - Find smallest space it can fit into
- Worst Fit - Largest place it can fit into

- Instead of variable sized blocks, the O.S. uses some sized blocks
- Virtual Pages - some sized chunks in logical address
- Physical Frames - some sized chunks in physical address space



← Logical Address
P = Page #