Goals:

* How does an OS provide concurrency through threads?
  + High-level discussion of how stacks contribute to concurrency
* Introduce needs for synchronization
* Discussion of locks and semaphores

Recall: IPC

* Mechanism to create communication channel between distinct processes
  + Need to selectively punch holes in process protection
* Requires serialization format understood by both
* Failure in process isolated from the other
  + Sharing is done in a controlled way through IPC
  + Still have to be careful handling what is received by IPC

Recall: POSIX/Unix Pipe

* Data structure queue (of finite size) in the kernel
* Acts as a file descriptor (but its an in memory queue, no persistence) 🡺 read, write syscalls()
* Memory buffer is finite:
  + If Producer tries to write when buffer full, it blocks
  + If Consumer tried to read when buffer empty, it blocks
* int pipe(int fileds[2]) syscall
  + Writes to fileds[1], read from fileds[0]
  + Implemented as fixed size queue

Recall: Socket Endpoint for Communication

* Key Idea: Communication across the world looks like File I/O
* Bidirectional endpoint for Communication
  + Queues are to temporarily hold results
  + Queues are NOT pipes!
* Connection: Two sockets connected over network 🡺 IPC over network
* Connection Setup over TCP/IP 🡺 5-Tuple identifies each connection (Src IP/Port, Dest IP/Port, Protocol)

Multiplexing Processes: The Process Control Block

* Kernel represents each process as a process control block
  + Status (running, ready, blocked, …)
  + Register state (when not ready)
  + Process state
  + Process ID, User, Executable, Priority
  + Program counter
  + List of Open files
* Kernel scheduler maintains a data structure containing the PCBs
  + Give out CPU to different processes
  + This is a policy decision
  + Give out non-CPU resources
    - Memory/IO
* A diagram of a system

  Description automatically generated with low confidenceContext Switch
  + Execute P0 🡺 Interrupt or Sys Call 🡺 Save state into PCB0
  + Reload state from PCB1 🡺 Execute P1 🡺 Interrupt or Sys Call 🡺 Save state into PCB1
  + Reload State from PCB0

Launching too many processes will drastically reduce execution time of any process (trashing)

Saving/Reload is expensive (need to save, reload registers)

As a process executes, it changes state:

* Execute fork(), place process on scheduling queue
* new: The process/thread is being created
* A diagram of a process

  Description automatically generated with medium confidencePlaced on ready queue (from scheduling queue)
* ready: The process is waiting to run
* Scheduler pulls it off ready queue
* running: Instructions are being executed
* waiting: Process waiting for some event to occur (disk access)
* If we have more than one core, more than one thing can run 🡺 Scheduler has multiple waiting and ready queues
* terminated: The process has finished execution

(Terminated, process not available to run, but parent still needs to get the result)

Scheduling: All About Queues

* PCB’s move from queue to queue
* Scheduling: which order to remove the queue
* Scheduling 🡺 Which order/process remove from the ready queue gets CPU access

A diagram of a link

Description automatically generated with low confidence

Ready Queue + Various IO device queue

* Process is not running 🡺 PCB is in some scheduler queue
  + Separate queue for each device/signal/condition
  + Each queue can have different scheduler policy
  + Scheduler only interacts on ReadyQueue, other queues talk with device drivers

Scheduling: Mechanism for deciding which processes/threads receive the CPU

if (readyProcesses(PCBs)){

nextPCB = selectProcess(PCBs)

run(nextPCB)

else{

run\_idle\_process()

}

Scheduling: Mechanism for deciding which processes/threads receive the CPU

Lots of different scheduling policies

Recall: Single and multithreaded processes

* A picture containing text, screenshot, font, number

  Description automatically generatedThreads encapsulate concurrency: “Active” component
* Address spaces encapsulate protection: “Passive” part
  + Keeps buggy programs from trashing the system
* Why have multiple threads per address space?
  + For sharing

Core of Concurrency: the Dispatch Loop

* Dispatch Loop
* Loop {

RunThread()

ChooseNextThread()

SaveStateOfCPU(curTCB)

LoadStateOfCPU(newTCB)

}

This is an infinite loop

Should we ever exit this loop?

* Shutting down machine, panic, hardware failiure

Running a thread:

* Consider first portion: RunThread()
* How do I run a thread?
  + Load state (registers, PC, stack pointer) into CPU
  + Load environment (virtual memory, space, etc)
  + Jump to the PC
  + OS (which manages threads) and threads itself run on same CPU (and only one can run a time 🡺 Need to make sure proper transition between them 🡺 Loading PC means OS gives up control
* How does the dispatcher get control back?
  + Internal events: thread returns control voluntarily
  + External events: thread gets preempted

Internal Events

* Block on I/O
  + Act of requesting I/O implicitly yields the CPU
  + Waiting on a “signal” from the other thread
  + Thread executes yield()
    - Thread volunteers to give up CPU
    - ComputePI() {

While(TRUE){

ComputeNextDigit();

yield();

A picture containing text, font, screenshot, line

Description automatically generated}

}

* + Stack for yielding thread
  + yield() executes syscall which leaves user stack, enters kernel stack 🡺 run\_new\_thread 🡺 switch (blue user, red kernel)
  + For each user stack thread, there is a kernel stack (this distinction is for safe-guarding).
  + Don’t trust user, check to see what user gave you and make sure it’s okay, and then actually execute
  + How do we run a new thread?
    - run\_new\_thread(){

newThread = PickNewThread();

switch(curThread, newThread);

ThreadHouseKeeping;

}

* + How does a dispatcher switch to new thread?
    - Save anything next thread my trash: PC, registers, SP
    - Maintain isolation for each thread
  + What do the stack look like?

Proc A(){

B();

}

Proc B(){

While(TRUE)

yield();

}

A picture containing text, screenshot, font, diagram

Description automatically generatedA screenshot of a computer program

Description automatically generated with low confidenceSuppose we have two threads, S and T

Thread is complete, self-contained, snapshot of running state (TCB and two stacks)

Switch details:

* TCB + stacks (user/kernel) contains complete restartable state of thread
  + Can put in queue for later revival
* What if you make a mistake in implementing switch?
  + Suppose you forgot to save/restore register 32
  + Get intermittent failure depending on when context switch occurred and whether new thread uses register 32
  + System will give wrong result without warning
* Can you devise an exhaustive test to test switch code?
  + No! Too many combinations to test switch code
* Design for simplicity (Topaz Kernel story…), microptimization sometimes not worth it

Aren’t we still switching contexts?

* Yes, but much cheaper than switching processes
  + No need to change address space
* Some numbers for Linux
  + Context Switch: 10-100ms
  + Processes: 3-4us
  + Threads: 100ns
* Even cheaper: switch threads(using “yield”) in user-space!
* One-to-One threading model (each user thread has a kernel thread 🡺 Kernel thread is kernel stack that is one to one matched up with a user thread, such that users stack is switched out with kernel stack and then return back to user stack 🡺 Kernel stack suspended). Standard Linux model 🡺 If user thread does I/O, kernel thread goes to sleep queue and other kernel threads can run
* Many-to-One thread 🡺 Each kernel thread has more than one user thread 🡺 yield is a user level call (saves and restores registers w/ user threads w/o switching to kernel). Faster than one-to-one 🡺 If some thread is suspended due to I/O… kernel thread is suspended and hence all threads matching to kernel thread are suspended
* A picture containing text, line, screenshot, diagram

  Description automatically generatedMany-To-Many 🡺

Processes vs. Threads:

(Single Core)

* Switch overhead
  + Same process: low
  + Different proc: high
* Protection
  + Same proc: low
  + Different proc: high
* Sharing overhead
  + Same proc: low
  + Different proc: high
* Parallelism: no

(Multiple Core)

* Switch overhead
  + Same process: low
  + Different proc: high
* Protection
  + Same proc: low
  + Different proc: high
* Sharing overhead
  + Same proc: low
  + Different prov, simultaneous core: medium
  + Different proc, offloaded core: high
* A picture containing text, screenshot, font, line

  Description automatically generatedParallelism: yes

What happens when thread blocks in I/O?

* What happens when a thread requests a block of data from the file system?
  + User code invokes a system call
  + Read operation is initiated
  + Run new thread/switch
* Thread communication similar
  + Wait for Signal/Join
  + Networking

What happens if a thread never does any I/O, never waits, never yields control?

* Could the ComputePI program grab all the resources and never release the processor?
* Must find way that dispatcher can regain control!
* Answer: Utilize external events
  + Interrupts: Signals from hardware or software that stop the running code and jump back to kernel
  + Timer: Like an alarm clock that goes off every some milliseconds

If we can ensure that the external events occur frequently enough, can ensure dispatcher runs

Interrupt Controller

* Interrupts invoked with interrupt lines from devices
* Interrupt controller chooses interrupt request to honor
  + Interrupt identity specified with ID line
  + Maske enables/disables interrupts
  + Priority encoder picks highest enabled interrupt
  + Software Interrupt Set/Cleared by Software
* CPU can disable all interrupts with internal flag

A picture containing text, screenshot, font, line

Description automatically generated

Network Interrupt Example:

* An interrupt is a hardware-invoked context switch
  + No separate step to choose what to run next
  + Always run the interrupt handler immediately

A picture containing text, screenshot, font, line

Description automatically generated

Use of Timer to Interrupt to Return Control:

* Solution to our dispatcher problem
* Use of timer interrupt to force scheduling decision

TimerInterrupt(){

DoPeriodicHouseKeeping();

run\_new\_thread();

}

How do we initialize TCB and Stack?

* Initialize Register fields of TCB
  + Stack pointer made to point at stack
  + PC return address 🡺 OS (asm) routine ThreadRoot()
  + A screen shot of a computer

    Description automatically generated with low confidenceTwo arg registers (a0 and a1) initialized to fcnPtr and fcnArgPtr
* Initialize stack data?
  + No, important part of stack frame is in registers (ra)
  + Think of stack frame as just before body of ThreadRoot() really gets started
  + ThreadRoot stub

A picture containing text, screenshot, font, line

Description automatically generatedWhat does ThreadRoot look like?

Correctness with Concurrent Threads?

* Non-determinism
  + Scheduler can run threads at any order
  + Scheduler can switch threads at any time
  + This can make testing very difficult
* Independent Threads
  + No state shared with other threads
  + Deterministic, reproducible conditions
* Cooperating threads
  + Shared state between multiple threads
* GOAL: CORRECTNESS BY DESIGN

A screenshot of a computer code

Description automatically generated with low confidenceATM Bank Server example:

* Can build event driven version with a dispatch loop
  + Still like to overlap I/O with computation
  + Without threads, we would have to rewrite in event-driven style
  + A screenshot of a computer program

    Description automatically generated with low confidenceWhat if missed an I/O step?

Can threads make this easier?

* A picture containing text, receipt, font, screenshot

  Description automatically generatedThreads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments
  + One thread per request
* Requests proceeds to completion, blocking as required
* Unfortunately shared state can get corrupted

A picture containing text, font, screenshot

Description automatically generated

Atomic Operations:

* To understand a concurrent program, we need to know what the underlying indivisible operations are
* Atomic operation: an operation that always runs to completion or not at all
  + It is indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  + Fundamental building block – if no atomic operations, then no way for threads to work together

Recall: Locks

* Lock – prevents someone from doing something
  + Lock – before entering critical section and before accessing shared data
  + Unlock – when leaving, after accessing data
  + Wait – if locked
* Locks provide two atomic operations
  + Acquire(&lock) – wait until lock is free; then mark it as busy
    - Calling thread “holds” the lock
  + Release(&lock) – mark lock as free

A picture containing text, screenshot, font, diagram

Description automatically generated

Fixing banking problems:

* Identify critical sections and add locking: