Recall: Structure of a Mesa Monitor Program

lock

while (need to wait){

condvar.wait();

}

unlock

lock

condvar.signal();

unlock

Monitor 🡺 When you signal and no one is waiting 🡺 Nothing happens

\*Provides simulation of Reader/Writer solution\*

Can readers starve?

Yes 🡺 while loop condition is to have constant writes 🡺 Reader will spin forever

A picture containing text, screenshot, font, number

Description automatically generated

If we erase condition check in Reader exit 🡺 writer signaled when there are readers (will just simply go back to sleep) or there are no writers to signal to

If we turn signal into broadcast 🡺 all writers will wake up and all but one will go back to sleep since they don’t acquire lock

Can we construct Monitors from Semaphores?

* Locking is easy 🡺 Just use a mutex
* Can we implement condition variables this way?
  + Wait(Semaphore \*thesema) { semaP(thesema); }
  + Signal(Semaphore \*thesema) { semaV(thesema); }
  + Will deadlock 🡺 Wait() may sleep with lock held
* Does this work better?
  + Wait(Lock \*thelock, Semaphore \*thesema) {  
     release(thelock);  
     semaP(thesema);  
     acquire(thelock);  
    }
  + Signal(Semaphore \*thesema) {

semaV(thesema);   
}

* No 🡺 Condition vars have no history, semaphores have history
  + Normally if one thread signals and no one receives and later another thread waits, it will go to sleep
  + With semaphore implementation, if one thread signals, V increments, and then later another thread waits, it just decrements and continues
  + Problem: P and V are commutative 🡺 Wait and Signal are NOT
* Does this fix the problem?
  + Wait(Lock \*thelock, Semaphore \*thesema) {  
     release(thelock);  
     semaP(thesema);  
     acquire(thelock);  
    }
  + Signal(Semaphore \*thesema) {  
     if semaphore queue is not empty  
     semaV(thesema);  
    }
  + Not legal to look at contents of semaphore queue
  + There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
* In C, make sure to release lock if exception is thrown. Otherwise thread throws exception, breaks out of procedure, but lock is still held

A computer code on a white background

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with low confidence

A computer code on a white background

Description automatically generated with low confidence

Recall: Thread State in the Kernel

* For every thread in a process, the kernel maintains:
  + The thread’s TCB
  + A kernel stack used for syscalls/interrupts/traps
    - This kernel-state is sometimes called the “kernel thread”
    - The “kernel thread” is suspended (but ready to go) when thread is running in user-space
* Additionally, some threads just do work in the kernel
  + Still has TCB
  + Still has kernel stack
  + But not part of any process, and never executes in user mode

Pintos: Processes are single threaded

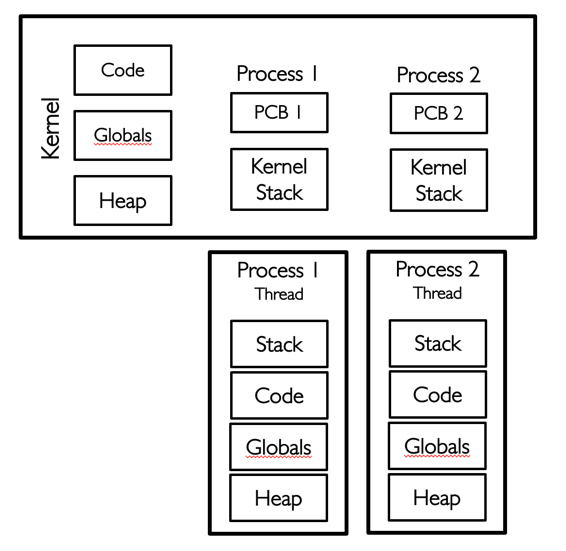
* TCB: Single page (4 KiB)
  + Stack growing from the top (high addresses\_
  + struct thread at the bottom
  + struct thread defines the TCB structure and PCB structure in Pintos

Multithreaded Processes:

* Traditional implementation strategy:
  + One PCB (process struct) per process
  + Each PCB (or stores pointers to) each thread’s TCB
* Linux’s strategy:
  + One task\_struct per thread
  + Threads belong to the same process happen to share some resources (address space, file desc table, etc)

Kernel Structure (remember kernel stack == kernel thread):

A picture containing text, screenshot, diagram, rectangle

Description automatically generated Single Thread per Process Multiple Threads per Process

Multiple Threads Per Process with Kernel A picture containing text, screenshot, diagram, font

Description automatically generated threads independent of any user threads (for kernel to perform its own computation)

Everything with a kernel thread is schedulable 🡺 Scheduler in kernel picks between different kernel threads 🡺 hence different user threads 🡺 How do schedule across these kernel threads?

Kernel thread is part of thread that get’s switched from switch syscall();

cs:eip, ss:esp associated with kernel thread of running thread is stored in tss (thread state structure) 🡺 when interrupt/syscall() executed or any transition to kernel 🡺 x86 grabs new stack pointer and inserts into stack portion of register

Each user process/thread associated with a kernel thread is described by a 4KB page object containing TCB and kernel stack for kernel thread

Interrupt vector table 🡺 How kernel ties in all interrupts to code that should run

Stubs.s 🡺 Generic handler 🡺 Know which handler from interrupt vector

A picture containing text, diagram, screenshot, font

Description automatically generatedUser 🡺 Enter interrupt 🡺 Save registers 🡺 Enter kernel via interrupt vector 🡺 interrupt vector table tells us where to start running 🡺 interrupt transfer control to code associated with interrupt

Timer may trigger thread switch

* thread\_tick
  + Updates thread counters
  + If quanta exhausted, sets yield flag
* thread\_yield
  + On path to rtn from interrupt
  + Sets current thread back to READY
  + Pushes it back to ready\_list
  + Calls schedule to select next tread to run upon iret
* Schedule
  + Selects next thread to run
  + Calls switch\_threads to change regs to point to stack for thread to resume
  + Sets its status to RUNNING
  + If user thread, activates the process
  + Returns back to intr\_handler

Timer interrupt

* Decide whether to schedule
* Pick new guy to schedule
* Put current kernel thread
* Load in kernel thread
* Return from kernel
* Now new user thread runs

Kernel Thread: Stack + State for independent execution in kernel

* Every user-level thread paired one-to-one with kernel thread
* Kernel thread associated with user thread is “suspended” (ready to go) when user-level thread is running