<https://docs.google.com/document/d/1x1GPOPXeWuED0wrKbLRMZDS4XP2tWwxWqCkbl4FkHWI/edit?usp=drivesdk>

|  |
| --- |
| //PAGEBREAK: 42  // Per-CPU process scheduler.  // Each CPU calls scheduler() after setting itself up.  // Scheduler never returns. It loops, doing:  // - choose a process to run  // - swtch to start running that process  // - eventually that process transfers control  // via swtch back to the scheduler.  void  scheduler(void)  {  struct proc \*p;  struct cpu \*c = mycpu();  c->proc = 0;    for(;;){  // Enable interrupts on this processor.  sti();  // Loop over process table looking for process to run.  acquire(&ptable.lock);  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  if(p->state != RUNNABLE)  continue;  // Switch to chosen process. It is the process's job  // to release ptable.lock and then reacquire it  // before jumping back to us.  c->proc = p;  switchuvm(p);  p->state = RUNNING;  swtch(&(c->scheduler), p->context);  switchkvm();  // Process is done running for now.  // It should have changed its p->state before coming back.  c->proc = 0;  }  release(&ptable.lock);  }  } |

xv6 schedules processes using the implementation of the Round-Robin scheduling algorithm. In this algorithm, each process is given a fixed time slice or quantum to execute, and then it is preempted and moved to the end of the queue. The scheduler then selects the next process in the queue and repeats the process.

During each iteration of the loop, the scheduler first disables interrupts on the CPU and acquires a lock on the process table to prevent concurrent access. It then searches the process table for a runnable process, and if one is found, switches to that process's virtual memory space, marks it as running, and transfers control to the process using swtch().

When the process's time slice expires, it will yield the CPU by calling yield() function which will change the process's state back to RUNNABLE, and move it to the end of the process queue. The scheduler then enables interrupts on the CPU, releases the process table lock, and continues the loop to select another process to run.

The per-CPU scheduler is called repeatedly in a loop to ensure that the CPU is always running a process, and to allow multiple CPUs to run different processes concurrently.

* **struct proc**: A struct that represents a process in the operating system. It contains information such as the process ID, program counter, stack pointer, etc.
* **struct cpu**: A struct that represents a CPU in the operating system. It contains information such as the current process being run by the CPU, the CPU's stack pointer, etc.
* **mycpu()**: A function that returns a pointer to the **struct cpu** representing the CPU that is currently executing the code.
* **sti()**: A function that enables interrupts on the CPU. Interrupts are used to handle external events, such as input/output operations.
* **acquire(&ptable.lock)**: A function that acquires a lock on the process table, preventing other CPUs from modifying it while the current CPU is working with it.
* **release(&ptable.lock)**: A function that releases the lock on the process table, allowing other CPUs to modify it again.
* **ptable.proc**: An array of **struct proc** representing all the processes in the system.
* **NPROC**: A constant representing the maximum number of processes allowed in the system.
* **p->state**: The current state of the process **p**. It can be **RUNNING**, **SLEEPING**, **RUNNABLE**, **ZOMBIE**, or **UNUSED**.
* **swtch(&(c->scheduler), p->context)**: A function that switches from the scheduler's stack to the stack of the selected process **p**. It saves the current context of the scheduler and restores the saved context of the process **p**.
* **switchuvm(p)**: A function that switches the current virtual memory layout to that of the process **p**. It sets up the page tables and enables paging for the process.
* **switchkvm()**: A function that switches the virtual memory layout back to the kernel's. It disables paging and restores the kernel's page tables.
* **c->proc = p**: Assigns the selected process **p** to the current CPU's **proc** field.
* **c->proc = 0**: Clears the current CPU's **proc** field to indicate that it is no longer running any process.
* **scheduler()**: The main function that implements the per-CPU process scheduler. It selects a runnable process from the process table and runs it using the **swtch()** function. After the process has finished running, it sets its state to **RUNNABLE** and continues looking for another process to run.

The sched() function in xv6 implements the scheduler and performs a context switch to the next process selected by the scheduler. It first declares an integer variable to temporarily store the interrupt enable flag and a pointer to the current process. The function then performs several checks, including whether the process table lock is held, whether only one lock is held by the current CPU, whether the current process is not already running, and whether interrupts are not already enabled. These checks ensure that the scheduler is called under the correct conditions. After saving the interrupt enable flag, the function calls the swtch() function to perform the context switch to the next process selected by the scheduler. The swtch() function saves the current process's state in its context and restores the state of the next process. Finally, the sched() function restores the interrupt enable flag for the current CPU. In xv6, the scheduler() function selects the next process to run based on the scheduling policy and is called by the trap() function when a process relinquishes the CPU. On the other hand, the sched() function performs the actual context switch between the current process and the selected process by saving the state of the current process and restoring the state of the selected process. The scheduler() and sched() functions together implement the core of the xv6 process scheduling mechanism.

Synchronization methods are techniques used to coordinate access to shared resources and prevent race conditions in concurrent systems. They can be implemented using various hardware instructions provided by modern processors.

Software instructions: These functions use standard software programming techniques, such as atomic operations and mutual exclusion, to ensure that shared resources are accessed in a controlled and coordinated way by multiple processes or threads.

Spinlocks: spinlock is a simple synchronization mechanism that busy-waits until a lock becomes available. Spinlocks are typically used for short-duration critical sections, where the overhead of putting a process to sleep and waking it up again is too high.

Sleeplocks: sleeplock is a more advanced synchronization mechanism that puts a process to sleep if a lock is not immediately available. Sleeplocks are typically used for longer-duration critical sections, where the overhead of busy-waiting is too high.

Semaphores: semaphore is a synchronization mechanism that allows processes to wait until a resource becomes available. Semaphores can be used to enforce mutual exclusion, to synchronize processes, or to signal events between processes.

Conditional variables: condvar is a synchronization mechanism that allows processes to wait for a specific condition to become true before proceeding. Conditional variables are typically used to implement synchronization patterns such as producer-consumer and reader-writer.

The implementation of semaphores and conditional variables can be found in the proc.c file, which includes the functions initlock(), initlock2(), initcondvar(), acquire(), release(), wait(), and wakeall(). These functions use the sleeplock implementation to ensure that access to shared resources is properly synchronized.

Hardware instructions: xv6 uses the xchg instruction provided by the hardware to implement atomic instructions like spinlocks and semaphores. The xchg instruction atomically exchanges the contents of a register or memory location with another register or memory location. This instruction is used to perform test-and-set and compare-and-swap operations.

|  |
| --- |
| // Enter scheduler. Must hold only ptable.lock  // and have changed proc->state. Saves and restores  // intena because intena is a property of this  // kernel thread, not this CPU. It should  // be proc->intena and proc->ncli, but that would  // break in the few places where a lock is held but  // there's no process.  void  sched(void)  {  int intena;  struct proc \*p = myproc();  if(!holding(&ptable.lock))  panic("sched ptable.lock");  if(mycpu()->ncli != 1)  panic("sched locks");  if(p->state == RUNNING)  panic("sched running");  if(readeflags()&FL\_IF)  panic("sched interruptible");  intena = mycpu()->intena;  swtch(&p->context, mycpu()->scheduler);  mycpu()->intena = intena;  } |

|  |
| --- |
| // Sleeping locks  #include "types.h"  #include "defs.h"  #include "param.h"  #include "x86.h"  #include "memlayout.h"  #include "mmu.h"  #include "proc.h"  #include "spinlock.h"  #include "sleeplock.h"  void  initsleeplock(struct sleeplock \*lk, char \*name)  {  initlock(&lk->lk, "sleep lock");  lk->name = name;  lk->locked = 0;  lk->pid = 0;  }  void  acquiresleep(struct sleeplock \*lk)  {  acquire(&lk->lk);  while (lk->locked) {  sleep(lk, &lk->lk);  }  lk->locked = 1;  lk->pid = myproc()->pid;  release(&lk->lk);  }  void  releasesleep(struct sleeplock \*lk)  {  acquire(&lk->lk);  lk->locked = 0;  lk->pid = 0;  wakeup(lk);  release(&lk->lk);  }  int  holdingsleep(struct sleeplock \*lk)  {  int r;    acquire(&lk->lk);  r = lk->locked && (lk->pid == myproc()->pid);  release(&lk->lk);  return r;  } |

|  |
| --- |
| // Mutual exclusion spin locks.  #include "types.h"  #include "defs.h"  #include "param.h"  #include "x86.h"  #include "memlayout.h"  #include "mmu.h"  #include "proc.h"  #include "spinlock.h"  void  initlock(struct spinlock \*lk, char \*name)  {  lk->name = name;  lk->locked = 0;  lk->cpu = 0;  }  // Acquire the lock.  // Loops (spins) until the lock is acquired.  // Holding a lock for a long time may cause  // other CPUs to waste time spinning to acquire it.  void  acquire(struct spinlock \*lk)  {  pushcli(); // disable interrupts to avoid deadlock.  if(holding(lk))  panic("acquire");  // The xchg is atomic.  while(xchg(&lk->locked, 1) != 0)  ;  // Tell the C compiler and the processor to not move loads or stores  // past this point, to ensure that the critical section's memory  // references happen after the lock is acquired.  \_\_sync\_synchronize();  // Record info about lock acquisition for debugging.  lk->cpu = mycpu();  getcallerpcs(&lk, lk->pcs);  }  // Release the lock.  void  release(struct spinlock \*lk)  {  if(!holding(lk))  panic("release");  lk->pcs[0] = 0;  lk->cpu = 0;  // Tell the C compiler and the processor to not move loads or stores  // past this point, to ensure that all the stores in the critical  // section are visible to other cores before the lock is released.  // Both the C compiler and the hardware may re-order loads and  // stores; \_\_sync\_synchronize() tells them both not to.  \_\_sync\_synchronize();  // Release the lock, equivalent to lk->locked = 0.  // This code can't use a C assignment, since it might  // not be atomic. A real OS would use C atomics here.  asm volatile("movl $0, %0" : "+m" (lk->locked) : );  popcli();  }  // Record the current call stack in pcs[] by following the %ebp chain.  void  getcallerpcs(void \*v, uint pcs[])  {  uint \*ebp;  int i;  ebp = (uint\*)v - 2;  for(i = 0; i < 10; i++){  if(ebp == 0 || ebp < (uint\*)KERNBASE || ebp == (uint\*)0xffffffff)  break;  pcs[i] = ebp[1]; // saved %eip  ebp = (uint\*)ebp[0]; // saved %ebp  }  for(; i < 10; i++)  pcs[i] = 0;  }  // Check whether this cpu is holding the lock.  int  holding(struct spinlock \*lock)  {  int r;  pushcli();  r = lock->locked && lock->cpu == mycpu();  popcli();  return r;  }  // Pushcli/popcli are like cli/sti except that they are matched:  // it takes two popcli to undo two pushcli. Also, if interrupts  // are off, then pushcli, popcli leaves them off.  void  pushcli(void)  {  int eflags;  eflags = readeflags();  cli();  if(mycpu()->ncli == 0)  mycpu()->intena = eflags & FL\_IF;  mycpu()->ncli += 1;  }  void  popcli(void)  {  if(readeflags()&FL\_IF)  panic("popcli - interruptible");  if(--mycpu()->ncli < 0)  panic("popcli");  if(mycpu()->ncli == 0 && mycpu()->intena)  sti();  } |