Tutorial: xv6 Process State Transitions & RR Scheduling Design

I. xv6 Processes & State Transitions

In operating systems, a **process** is the fundamental unit of resource allocation and scheduling. As a simplified Unix-like system, xv6's core functionality revolves around process state management and scheduling mechanisms.

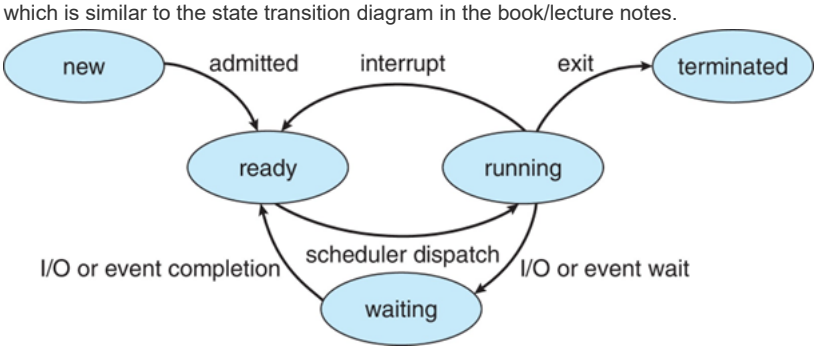
Today's topic to show how process state transitions are handled in xv6 and the Round Robin (RR) scheduling algorithm implementation.

1. xv6 Process States definition

xv6 defines 6 process states (via `enum procstate` in `kernel/proc.h`):

```
enum procstate {
  UNUSED,    // Unused (process control block/PCB is free)
  USED,      // process is being created and initialized
  SLEEPING,  // Sleeping (blocked, waiting for an event to complete)
  RUNNABLE,  // Runnable (ready to execute but not using the CPU)
  RUNNING,   // Running (currently executing on the CPU)
  ZOMBIE     // Zombie (process exited, waiting for parent to reclaim resources)
};
```

which is similar to the state transition diagram in the book/lecture notes.

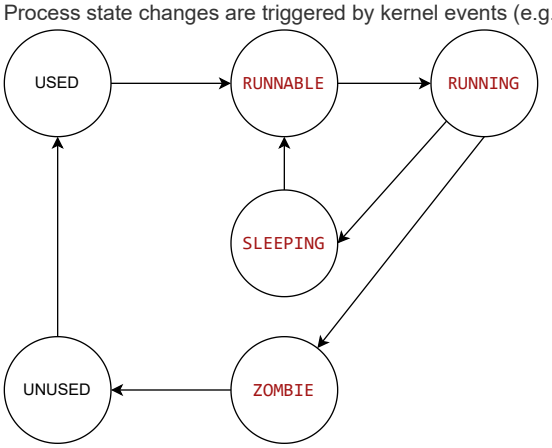


Each process's state info in xv6 is stored in a PCB (Process Control Block) represented by a `struct proc` (defined in `kernel/proc.h`), which not only stores the process's state but also stores other critical information such as the process ID, parent process, execution context (e.g., kernel stack, trap frame), memory space (e.g., page table), and file descriptors.

```
// Per-process state
struct proc {
  struct spinlock lock;        // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID
  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process
  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

2. Detailed State Transitions

Process state changes are triggered by kernel events (e.g., system calls, interrupts, exceptions). Below are the typical transition flows:



Typical state transitions in xv6 are as follows:

(1) Creation Phase: `UNUSED → USED → RUNNABLE`

- **Trigger**: The user calls the `fork()` system call to create a new process.
- In xv6, the complete process creation usually involves two steps:
  i. fork: creates a new process that is almost identical to the parent process.
  ii. exec: loads and executes a new program in the child process.

the shell typically first forks a child process, and then calls exec in the child process to execute the program specified by the user.

In this way, the memory image of the child process is completely replaced, but other attributes of the process (such as process ID, open file descriptors, etc.) are retained.

- **Flow**:
  i. The kernel `kfork` allocates a free PCB from the pool of idle processes (initially `UNUSED`).
  ii. It initializes the process (e.g., copies the parent's memory and file descriptors) and sets the state to `USED`.
  iii. Once initialization is complete, the state is updated to `RUNNABLE`, and the process waits for the scheduler to select it.

(2) Execution phase: `RUNNABLE ⇔ RUNNING`

- **Trigger**: The scheduler selects a process for execution(context switch)

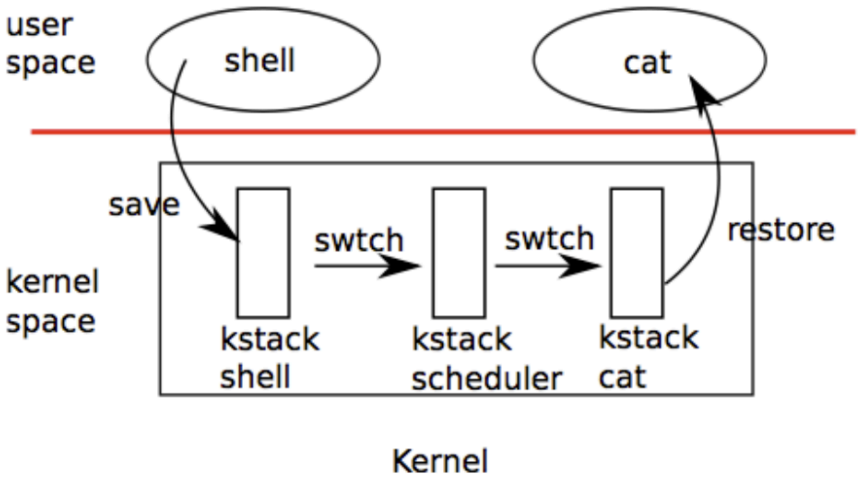


Figure 5-1. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

- **Flow**:
  i. The scheduler (in `scheduler()`) selects a `RUNNABLE` process, sets its state to `RUNNING`, and switches to its execution context using `swtch()`.
  ii. The states of child process reverts to `RUNNABLE` to wait for the next scheduling cycle.

We will show the codes of the execution phase in detail in the next section with the Round Robin (RR) scheduling algorithm.

(3) Blocking phase: `RUNNING → SLEEPING → RUNNABLE`

- **Trigger**: The process waits for an external event (e.g., I/O completion, child process exit, timed sleep).
  - xv6 use a channel mechanism to sleep a process and wait for an event to complete.
  - Implementation of sleep and wakeup in xv6. The overall idea is that sleep converts the current process to the `SLEEPING` state and calls sched to release the CPU, while wakeup finds a sleeping process and marks it as `RUNNABLE`.
- **Flow**:
  i. The process calls `sleep()` (e.g., to wait for I/O). The kernel sets its state to `SLEEPING` and yields the CPU.
  ii. When the awaited event completes (e.g., I/O interrupt, timer timeout interrupt), the kernel calls `wakeup()`, updating the process's state back to `RUNNABLE` so it can rejoin the scheduling queue.

(4) Exit phase: `RUNNING → ZOMBIE → UNUSED`

- **Trigger**: The process exits via `exit()` or is terminated by a signal `killed`. But in this section, we focus on the `exit()` system call.
- **Flow**:
  - **Parent Process Waiting**: The parent process invokes the `wait()` system call, which eventually enters the `kwait` function. If no child process has exited, the parent process calls `sleep(p, &wait_lock)` (inside `kwait` function) to enter the `SLEEPING` state, waiting on the pointer to its own process control block (denoted as `p`).
  - **Child Process Exiting**: After the child process finishes execution, it calls the `exit()` system call, which eventually enters the `kexit` function. After releasing resources, the child process invokes `wakeup(p->parent)` —here, `p->parent` refers to the pointer of the parent process's process control block, serving as the channel for wake-up.
  - **Parent Process Being Awakened**: The `wakeup` function iterates through all processes, finds the parent process that is in the `SLEEPING` state and waiting on the same channel (i.e., the parent process's `proc` pointer), and sets its state to `RUNNABLE`.
  - **Parent Process Resuming Execution**: Once the parent process is selected for execution by the scheduler, it resumes execution from the `sleep` call within the `kwait` function. At this point, it can locate the child process that has transitioned to the `ZOMBIE` state, processes the child's exit status, and returns the child process's PID.

II. Round-Robin (RR) Scheduling Design

RR scheduling is xv6's default scheduling algorithm. Its core principle is **"fair CPU time allocation"**: all runnable processes take turns using the CPU, each executing for a fixed **time slice**. When a process's time slice expires, it is preempted and moved to the end of the runnable queue.

1. Core Goals of RR Scheduling

- **Fairness**: Ensure every process (regardless of priority) receives roughly equal CPU time. keep no starvation.

A simple for loop is used to ensure that each process gets a chance to run for a fixed time slice.

2. Key Mechanisms of RR Scheduling in xv6

let's look at the code of `scheduler()` in xv6.

```
// kernel/proc.c/scheduler()
for(p = proc; p < &proc[NPROC]; p++){
  acquire(&p->lock);
  if(p->state == RUNNABLE){
    p->state = RUNNING;  // Runnable → Running
    c->proc = p;
    swtch(&c->context, &p->context);  // Switch execution context
    c->proc = 0;  // After process yield the CPU, state is back to RUNNABLE
  }
  release(&p->lock);
}
```

you can see that the scheduler iterates through all process slots `for(p = proc; p < &proc[NPROC]; p++)` to find those in the `RUNNABLE` state. If a process is found, it is set to `RUNNING`, and the scheduler switches to its execution context. After the process yields the CPU (either by time slice expiration or preemption), its state is set back to `RUNNABLE` to wait for the next scheduling cycle.

In This way, all processes get a chance to run for a fixed time slice, ensuring fairness.

P0	P1	P2	P3	P0	P1	P2	P3	P0

Time Slice to ensure fairness

From the scheduler's codes, you can see that there only has one line of code that switches the execution context to the next process.

```
        swtch(&c->context, &p->context);  // Switch execution context
```

How does the scheduler get right back for next process after time slice expires?

In xv6, the RR time slice is typically **1 timer tick** (≈100ms, determined by the hardware timer frequency). Each process can execute for at most 1 tick per scheduling cycle before being preempted.

Clock Interrupts & Preemption

- **Trigger**: A hardware timer generates a clock interrupt every 1/10 of a second.

```
void
clockintr()
{
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }

  // ask for the next timer interrupt. this also clears
  // the interrupt request. 1000000 is about a tenth
  // of a second.
  w_stimecmp(r_time() + 1000000);
}
```

- **Preemption Logic**: The interrupt handler (in `usertrap()`) checks if the currently running process has exhausted its time slice. If so, it sets the process's state from `RUNNING` to `RUNNABLE` and triggers a reschedule.

```
    // give up the CPU if this is a timer interrupt.
    if(which_dev == 2){
      yield();
    }
```

In yield(), the process is set to `RUNNABLE` state and call `sched()` which let the scheduler is invoked to select the next process to run.

```
void yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

In sched(), there will be some sanity check to ensure that the process is in the right state. and then switch the cpu context back to scheduler() and continue the scheduling process.

```
void sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&p->lock))
    panic("sched p->lock");
  if(mycpu()->noff != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched RUNNING");
  if(intr_get())
    panic("sched interruptible");

  intena = mycpu()->intena;
  swtch(&p->context, &mycpu()->context); // switch context back to scheduler process
  mycpu()->intena = intena;
}
```

Execution Flow of RR Scheduling

`scheduler() acquire lock → check is runnable process → if yes, child process run (via context switch) → time up → yield () → sched() → return to sc`

3. Idea of implementing priority scheduling

- **Priority**: Each process has its own PCB, and each PCB has its own priority value.
  - Priority value (e.g., `p->priority`)
  - Remaining time (e.g., `p->remain_time`). to record the remaining time slice for the process.
- These info need to be updated in each scheduling cycle.
- These info can be stored in the pcb struct.
- **Scheduling Logic**:
  i. In `scheduler()` , before selecting the next runnable process, sort the queue by priority (lower values first).
  ii. Inside each priority Queue, using RR scheduling to select the next runnable process. (basicly the same as RR scheduling in standard XV6 design)
  iii. repeat process 1 and 2 for each priority queue.
- **Priority Demotion**:
  every times when a process is running and interrupted by timer, you can check if the process has remaining time.
  i. If the selected process has remaining time, just update its remaining time and yeild.
  ii. If the selected process has no remaining time, demote its priority by 1, reset its remaining time according to its new priority.

III. Summary

1. **Process State Transitions**: xv6 processes follow a lifecycle of `UNUSED → USED → RUNNABLE ⇔ RUNNING → SLEEPING/ZOMBIE → UNUSED` . The key lies in understanding the triggers for state changes (e.g., system calls, interrupts).
2. **RR Scheduling**: By using fixed time slices and clock-interrupt preemption, RR ensures fair CPU rotation for all processes. It is xv6's default and simplest scheduling strategy.

Some Frequently Asked Questions

1. How to add Q3_test.c into qemu environment?
   i. Copy The Q3_test.c into `xv6-rsicv/user/`
   ii. Add the Q3_test.c into Makefile's user program Compile list `UPROGS=`

```
135  UPROGS=\
136      $U/_cat\
137      $U/_echo\
138      $U/_forktest\
139      $U/_grep\
140      $U/_init\
141      $U/_kill\
142      $U/_ln\
143      $U/_ls\
144      $U/_mkdir\
145      $U/_rm\
146      $U/_sh\
147      $U/_stressfs\
148      $U/_usertests\
149      $U/_grind\
150      $U/_wc\
151      $U/_zombie\
152      $U/_logstress\
153      $U/_forphan\
154      $U/_dorphan\
155      $U/_Q3_test\
```

2. How to run Q3_test.c in qemu environment?
    i. In xv6-rsicv directory, run `make qemu` to start qemu environment.
    ii. use pure `English input` to input `Q3_test` to run Q3_test.c (case sensitive, do not press direction key or space key before or after the command)
    a.

```
csc3150@csc3150:~/hw3/xv6-riscv$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kerne
l/kernel -m 128M -smp 3 -nographic -global virtio-mmio.for
ce-legacy=false -drive file=fs.img,if=none,format=raw,id=x
0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ Q3_test
```