hey. can you expalin how the scheduler works ?

Sure, let's break down the scheduler function step-by-step.

Pseudocode Explanation

1. **Initialization**:
    - Declare a pointer `p` to iterate over processes.
    - Get the current CPU using `mycpu()` and set its `proc` field to `0`.
2. **Infinite Loop**:
    - Print a comma and newline for debugging.
    - Enable interrupts using `intr_on()` to avoid deadlocks.
    - Initialize a flag `found` to `0` to track if a runnable process is found.
3. **Process Iteration**:
    - Iterate over all processes in the `proc` array.
    - Acquire the lock for each process.
    - Check if the process state is `RUNNABLE`.
        - If yes, set the process state to `RUNNING`.
        - Set the current CPU's `proc` field to this process.
        - Switch context to the chosen process using `swtch()`.
        - After the process is done running, set the CPU's `proc` field back to `0`.
        - Set `found` to `1`.
    - Release the lock for the process.
4. **Idle State**:
    - If no runnable process is found (`found == 0`), enable interrupts and wait for an interrupt using `wfi`.

Code Explanation

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    printf(",\n");

    // Enable interrupts to avoid deadlock
    intr_on();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
```

```c
        acquire(&p->lock);
        if(p->state == RUNNABLE) {
          // Switch to chosen process
          p->state = RUNNING;
          c->proc = p;
          swtch(&c->context, &p->context);

          // Process is done running for now
          c->proc = 0;
          found = 1;
        }
        release(&p->lock);
      }

      if(found == 0) {
        // No runnable process found, wait for an interrupt
        intr_on();
        asm volatile("wfi");
      }
    }
  }
```

Key Functions and Concepts

- `mycpu()`: Returns the current CPU.
- `intr_on()`: Enables interrupts.
- `acquire(&p->lock)`: Acquires the lock for process `p`.
- `release(&p->lock)`: Releases the lock for process `p`.
- `swtch(&c->context, &p->context)`: Switches context from the scheduler to the chosen process.
- `wfi`: Assembly instruction to wait for an interrupt.

Summary

The scheduler continuously loops, looking for runnable processes. When it finds one, it switches context to that process. If no runnable process is found, it waits for an interrupt to avoid busy-waiting. This ensures efficient CPU utilization and process management.

So the issue have is the scheduler keeps on looping without switching to a runnable process after i run printhell.

To diagnose why the scheduler keeps looping without switching to a runnable process, we need to ensure that:

1. There are runnable processes in the process table.
2. The `swtch` function is correctly switching contexts.
3. The process states are being correctly managed.

Steps to Diagnose the Issue

1. **Check if there are runnable processes**:
    - Ensure that there are processes in the `RUNNABLE` state in the process table.
    - Add debug prints to verify the state of each process.
2. **Verify `swtch` function**:
    - Ensure that the `swtch` function is correctly saving and restoring contexts.
    - Add debug prints before and after the `swtch` call to verify if the context switch is happening.
3. **Check process state transitions**:
    - Ensure that processes are correctly transitioning from `RUNNABLE` to `RUNNING` and back to `RUNNABLE` or other states.
    - Add debug prints to verify state transitions.

Adding Debug Prints

Let's add debug prints to the scheduler to verify the state of each process and the context switch.

Modified `scheduler` Function with Debug Prints


```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    printf("Scheduler loop start\n");

    // Enable interrupts to avoid deadlock
    intr_on();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      printf("Checking process %d, state: %d\n", p->pid, p->state);
      if(p->state == RUNNABLE) {
        printf("Switching to process %d\n", p->pid);
        // Switch to chosen process
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        // Process is done running for now
        // It should have changed its p->state before coming back
        c->proc = 0;
        found = 1;
      }
      release(&p->lock);
    }
```

```
    if(found == 0) {
      // No runnable process found, wait for an interrupt
      printf("No runnable process found, waiting for interrupt\n");
      intr_on();
      asm volatile("wfi");
    }
  }
}
```

Next Steps

1. **Run the modified scheduler**:
   - Rebuild and run your kernel with the modified scheduler.
   - Observe the debug prints to see if there are any runnable processes and if the context switch is happening.
2. **Check process creation and state transitions**:
   - Ensure that processes are being created and their states are being correctly set to `RUNNABLE`.
   - Verify that processes are transitioning back to `RUNNABLE` or other states after running.
3. **Verify `swtch` function**:
   - Ensure that the `swtch` function is correctly implemented and switching contexts.
   - Add debug prints in the `swtch` function to verify if the context switch is happening.

By following these steps, you should be able to diagnose why the scheduler is not switching to a runnable process and fix the issue.

I think something is wroung with sigalarm syscall

Let's investigate the `sigalarm` syscall implementation and its interaction with the scheduler. The `sigalarm` syscall sets up an alarm handler that is triggered after a specified interval. If the alarm handler or the syscall itself is not functioning correctly, it could cause issues with process scheduling.

Steps to Diagnose the Issue

1. **Verify `sigalarm` syscall implementation**:
   - Ensure that the `sigalarm` syscall correctly sets the alarm interval and handler.
   - Check if the alarm handler is being called as expected.
2. **Check process state transitions**:
   - Ensure that the process state transitions correctly when the alarm handler is triggered.
   - Verify that the process returns to the `RUNNABLE` state after the alarm handler executes.
3. **Add debug prints to `sigalarm` syscall**:

- Add debug prints to verify the values of the interval and handler being set.
- Add debug prints to check if the alarm handler is being called.

Modified `sys_sigalarm` Function with Debug Prints

Let's add debug prints to the `sys_sigalarm` function to verify its behavior.

```
uint64
sys_sigalarm(void)
{
  int interval;
  uint64 handler;
  argint(0, &interval);
  argaddr(1, &handler);
  struct proc *p = myproc();

  printf("Setting sigalarm: interval=%d, handler=%p\n", interval,
(void*)handler);

  if(p->alarm_trapframe) {
    kfree(p->alarm_trapframe);
    p->alarm_trapframe = 0;
  }

  p->alarm_interval = interval;
  p->alarm_handler = (void(*)())handler;
  p->alarm_on = (interval > 0);

  if(p->trapframe == 0){
    p->alarm_trapframe = kalloc();
  }

  printf("Sigalarm set: interval=%d, handler=%p\n", p->alarm_interval,
(void*)p->alarm_handler);

  return 0;
}
```


Verify Alarm Handler Execution

Ensure that the alarm handler is being called and the process state transitions correctly. Add debug prints in the `handler` function and check the process state transitions.

Modified `handler` Function with Debug Prints

```
void handler() {
  printf("_________________________Alarm!_________________________\n");
```

```
    sigreturn();
  }
```

Verify Process State Transitions

Ensure that the process transitions back to the `RUNNABLE` state after the alarm handler executes. Add debug prints in the scheduler to verify this.

Modified `scheduler` Function with Debug Prints

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    printf("Scheduler loop start\n");

    // Enable interrupts to avoid deadlock
    intr_on();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      printf("Checking process %d, state: %d\n", p->pid, p->state);
      if(p->state == RUNNABLE) {
        printf("Switching to process %d\n", p->pid);
        // Switch to chosen process
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        // Process is done running for now
        // It should have changed its p->state before coming back
        c->proc = 0;
        found = 1;
      }
      release(&p->lock);
    }

    if(found == 0) {
      // No runnable process found, wait for an interrupt
      printf("No runnable process found, waiting for interrupt\n");
      intr_on();
      asm volatile("wfi");
    }
  }
}
```

Next Steps

1. **Rebuild and run your kernel**:
    - Rebuild and run your kernel with the modified `sys_sigalarm` and `scheduler` functions.
    - Observe the debug prints to verify the behavior of the `sigalarm` syscall and the scheduler.
2. **Check the alarm handler execution**:
    - Ensure that the alarm handler is being called as expected.
    - Verify that the process transitions back to the `RUNNABLE` state after the alarm handler executes.
3. **Verify process state transitions**:
    - Ensure that processes are correctly transitioning between states (`RUNNABLE`, `RUNNING`, etc.).
    - Check if the scheduler is correctly switching to runnable processes.

By following these steps, you should be able to diagnose and fix any issues with the `sigalarm` syscall and the scheduler.

---

also look at the function sys_sigreturn

Let's review the `sys_sigreturn` function and ensure it correctly restores the process state after the alarm handler executes. This function is crucial for returning the process to its previous state, allowing it to continue execution.

`sys_sigreturn` Function

Here is the `sys_sigreturn` function from

:


```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  if(p->alarm_trapframe) {
    memmove(p->trapframe, p->alarm_trapframe, PGSIZE);
    kfree(p->alarm_trapframe);
    p->alarm_trapframe = 0;
  }
  return 0;
}
```

Explanation

1. **Retrieve the current process**:

    - `struct proc *p = myproc();` retrieves the current process.

2. **Check if `alarm_trapframe` is set**:
    - `if(p->alarm_trapframe) {` checks if the `alarm_trapframe` is set.

3. **Restore the process state**:
    - `memmove(p->trapframe, p->alarm_trapframe, PGSIZE);` copies the saved trapframe back to the current trapframe, restoring the process state to what it was before the alarm handler was called.

4. **Free the `alarm_trapframe`**:
    - `kfree(p->alarm_trapframe);` frees the memory allocated for the `alarm_trapframe`.
    - `p->alarm_trapframe = 0;` sets the `alarm_trapframe` pointer to `0`.

Adding Debug Prints

To diagnose any issues with the `sys_sigreturn` function, let's add debug prints to verify its behavior.

Modified `sys_sigreturn` Function with Debug Prints

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  printf("sys_sigreturn called for process %d\n", p->pid);
  if(p->alarm_trapframe) {
    printf("Restoring trapframe for process %d\n", p->pid);
    memmove(p->trapframe, p->alarm_trapframe, PGSIZE);
    kfree(p->alarm_trapframe);
    p->alarm_trapframe = 0;
    printf("Trapframe restored for process %d\n", p->pid);
  } else {
    printf("No alarm_trapframe to restore for process %d\n", p->pid);
  }
  return 0;
}
```


Next Steps

1. **Rebuild and run your kernel**:
    - Rebuild and run your kernel with the modified `sys_sigreturn` function.
    - Observe the debug prints to verify the behavior of the `sys_sigreturn` function.

2. **Check the alarm handler execution**:
    - Ensure that the alarm handler is being called as expected.

- Verify that the process transitions back to the `RUNNABLE` state after the alarm handler executes.

3. **Verify process state transitions**:
    - Ensure that processes are correctly transitioning between states (`RUNNABLE`, `RUNNING`, etc.).
    - Check if the scheduler is correctly switching to runnable processes.

By following these steps, you should be able to diagnose and fix any issues with the `sys_sigreturn` function and the scheduler.

---

where does sys_sigreturn fo after returning ?

The `sys_sigreturn` function is a system call handler. When it returns, control is handed back to the kernel's syscall handling mechanism, which then resumes the execution of the user process that made the system call.

Flow of Execution

1. **User Process Makes a System Call**:
    - The user process calls `sigreturn()`, which triggers a system call interrupt (`ecall`).
2. **Kernel Handles the System Call**:
    - The kernel's syscall handler (`syscall()`) is invoked.
    - The syscall handler determines which system call was made by examining the value in the `a7` register (which holds the syscall number).
3. **Kernel Executes the System Call Handler**:
    - The syscall handler (`syscall()`) calls the appropriate system call function based on the syscall number. In this case, it calls `sys_sigreturn()`.

4. **`sys_sigreturn` Executes**:
    - `sys_sigreturn` restores the process's trapframe from the saved `alarm_trapframe` and frees the `alarm_trapframe`.
5. **Return to Syscall Handler**:
    - After `sys_sigreturn` returns, control goes back to the syscall handler (`syscall()`).
6. **Resume User Process Execution**:
    - The syscall handler (`syscall()`) completes, and the kernel resumes execution of the user process from the point where it was interrupted to handle the system call.

Relevant Code in `syscall.c`

Here is the relevant part of the syscall handling mechanism in `syscall.c`:

```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();
  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // increment count of syscall
    syscall_counts[num]++;
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

Summary

- The `sys_sigreturn` function restores the process's state and returns to the syscall handler.
- The syscall handler then resumes the execution of the user process from the point where it was interrupted.

By understanding this flow, you can see how the `sys_sigreturn` function fits into the overall system call handling mechanism and how it allows the user process to continue execution after handling an alarm.

---

where does sys_sigreturn fo after returning ?

After `sys_sigreturn` returns, control is handed back to the syscall handler, which then resumes the execution of the user process from the point where it was interrupted to handle the system call.

Detailed Flow of Execution

1. **User Process Makes a System Call**:
    - The user process calls `sigreturn()`, which triggers a system call interrupt (`ecall`).
2. **Kernel Handles the System Call**:
    - The kernel's syscall handler (`syscall()`) is invoked.
    - The syscall handler determines which system call was made by examining the value in the `a7` register (which holds the syscall number).
3. **Kernel Executes the System Call Handler**:

   - The syscall handler (`syscall()`) calls the appropriate system call function based on the syscall number. In this case, it calls `sys_sigreturn()`.

4. **`sys_sigreturn` Executes**:

   - `sys_sigreturn` restores the process's trapframe from the saved `alarm_trapframe` and frees the `alarm_trapframe`.

5. **Return to Syscall Handler**:

   - After `sys_sigreturn` returns, control goes back to the syscall handler (`syscall()`).

6. **Resume User Process Execution**:

   - The syscall handler (`syscall()`) completes, and the kernel resumes execution of the user process from the point where it was interrupted to handle the system call.

Relevant Code in `syscall.c`

Here is the relevant part of the syscall handling mechanism in `syscall.c`:

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();
  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // increment count of syscall
    syscall_counts[num]++;
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```


Summary

- The `sys_sigreturn` function restores the process's state and returns to the syscall handler.
- The syscall handler then resumes the execution of the user process from the point where it was interrupted.

Example Flow with `sys_sigreturn`

1. **User Process**:

- Calls `sigreturn()`.

2. **System Call Interrupt**:

    - Triggers `ecall`.

3. **Kernel**:

    - Enters `syscall()` function.
    - Determines the syscall number is `SYS_sigreturn`.
    - Calls `sys_sigreturn()`.

4. **`sys_sigreturn`**:

    - Restores the trapframe.
    - Frees the `alarm_trapframe`.
    - Returns to `syscall()`.

5. **Kernel**:

    - `syscall()` completes.
    - Resumes user process execution.

By understanding this flow, you can see how the `sys_sigreturn` function fits into the overall system call handling mechanism and how it allows the user process to continue execution after handling an alarm.

**who calls syscall ? **

The `syscall` function is called by the trap handling mechanism when a user process makes a system call. This typically happens in response to an `ecall` (environment call) instruction executed by the user process. The `ecall` instruction triggers a trap, which is handled by the kernel.

Flow of Execution

1. **User Process Executes `ecall`**:

    - The user process executes an `ecall` instruction to request a system call.

2. **Trap Handling**:

    - The `ecall` instruction causes a trap, which is handled by the kernel's trap handling mechanism.

3. **Trap Handler**:

    - The trap handler determines the cause of the trap and calls the appropriate handler function. For system calls, it calls the `syscall` function.

Relevant Code in `trap.c`

Here is the relevant part of the trap handling mechanism in `trap.c`:

```c
void
usertrap(void)
{
  int which_dev = 0;
  struct proc *p = myproc();

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap().
  w_stvec((uint64)kernelvec);

  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid);
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }

  if(p->killed)
    exit(-1);

  if(which_dev == 2)
    yield();

  usertrapret();
}
```


Summary

- The `syscall` function is called by the `usertrap` function in `trap.c` when a user process makes a system call using the `ecall` instruction.

- The `usertrap` function handles traps from user mode, determines the cause of the trap, and calls `syscall` if the trap was caused by an `ecall` instruction.

By understanding this flow, you can see how the `syscall` function is invoked in response to system calls made by user processes.

---

where is the shell code that reads and runs user command ?

The shell code that reads and runs user commands is located in the

file. The relevant functions are `getcmd` and `runcmd`.

`getcmd` Function

The `getcmd` function reads a command from the user input:

```
int
getcmd(char *buf, int nbuf)
{
  write(2, "$ ", 2);
  memset(buf, 0, nbuf);
  gets(buf, nbuf);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}
```

`runcmd` Function

The `runcmd` function executes the parsed command:


```
void
runcmd(struct cmd *cmd)
{
  int p[2];
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    exit(1);

  switch(cmd->type){
  default:
    panic("runcmd");
  case EXEC:
```

```
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit(1);
    exec(ecmd->argv[0], ecmd->argv);
    fprintf(2, "exec %s failed\n", ecmd->argv[0]);
    break;
  case REDIR:
    rcmd = (struct redircmd*)cmd;
    close(rcmd->fd);
    if(open(rcmd->file, rcmd->mode) < 0){
      fprintf(2, "open %s failed\n", rcmd->file);
      exit(1);
    }
    runcmd(rcmd->cmd);
    break;
  case LIST:
    lcmd = (struct listcmd*)cmd;
    if(fork1() == 0)
      runcmd(lcmd->left);
    wait(0);
    runcmd(lcmd->right);
    break;
  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");
    if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->left);
    }
    if(fork1() == 0){
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
    wait(0);
    wait(0);
    break;
  case BACK:
    bcmd = (struct backcmd*)cmd;
    if(fork1() == 0)
      runcmd(bcmd->cmd);
    break;
  }
  exit(0);
}
```

Main Function

The `main` function initializes the shell and enters a loop to read and execute commands:

```c
int
main(void)
{
  static char buf[100];
  int fd;

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Chdir must be called by the parent, not the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        fprintf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait(0);
  }
  exit(0);
}
```


Summary

- The `getcmd` function reads a command from the user input.
- The `runcmd` function executes the parsed command.
- The `main` function initializes the shell, ensures necessary file descriptors are open, and enters a loop to read and execute commands.

I have tracked the issue after sigalarm/sigreturn to the acquire function in kernel/spinlock.c
But I cannot debug any further since printf function cannot be used within it.

To debug the `acquire` function in

without using `printf`, you can use other debugging techniques such as:

1. **Using a Debugger**: Use a debugger like GDB to step through the code and inspect the state of variables.
2. **Using LED or GPIO Pins**: If you are working on embedded systems, you can toggle GPIO pins to indicate the flow of execution.
3. **Using a Custom Logging Mechanism**: Implement a lightweight logging mechanism that writes to a memory buffer, which you can inspect later.
4. **Using Assertions**: Use assertions to check for conditions and halt execution if they are not met.

Example: Using a Custom Logging Mechanism

You can implement a simple logging mechanism that writes to a circular buffer in memory. This buffer can then be inspected to understand the flow of execution.

Step 1: Define the Logging Buffer

In

or a common header file:

```
#define LOG_SIZE 1024
char log_buffer[LOG_SIZE];
int log_index = 0;

void log_message(const char *msg) {
  int i = 0;
  while (msg[i] != '\0' && log_index < LOG_SIZE - 1) {
    log_buffer[log_index++] = msg[i++];
  }
  log_buffer[log_index++] = '\n';
}
```

Step 2: Use the Logging Function in `acquire`

Modify the `acquire` function to use the `log_message` function:

```
void
acquire(struct spinlock *lk)
{
  log_message("Attempting to acquire lock");
  push_off(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
    ;

  __sync_synchronize();
```

```
  lk->cpu = mycpu();
  log_message("Lock acquired");
}
```

Step 3: Inspect the Log Buffer

After running your code, inspect the `log_buffer` to see the logged messages. You can add a function to print the log buffer when needed:

```
void print_log_buffer(void) {
  for (int i = 0; i < log_index; i++) {
    uartputc_sync(log_buffer[i]);
  }
}
```

Call `print_log_buffer` at a convenient point in your code to output the log messages.

Example: Using Assertions

You can use assertions to check for conditions and halt execution if they are not met:

```
#include <assert.h>

void
acquire(struct spinlock *lk)
{
  assert(lk != NULL);
  push_off(); // disable interrupts to avoid deadlock.
  assert(!holding(lk));

  while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
    ;

  __sync_synchronize();

  lk->cpu = mycpu();
  assert(lk->cpu == mycpu());
}
```


Summary

By using a custom logging mechanism or assertions, you can debug the `acquire` function without relying on `printf`. This approach allows you to gather information about the execution flow and identify issues related to `sigalarm` and `sigreturn`.

---

seems like something is wrong with sigreturn. can you look into it >

Let's analyze the `sys_sigreturn` function and its context to identify potential issues.

`sys_sigreturn` Function

Here is the `sys_sigreturn` function from

:

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  printf("sys_sigreturn called for process %d\n", p->pid);
  if(p->alarm_trapframe) {
    printf("Restoring trapframe for process %d\n", p->pid);
    memmove(p->trapframe, p->alarm_trapframe, PGSIZE);
    kfree(p->alarm_trapframe);
    p->alarm_trapframe = 0;
    printf("Trapframe restored for process %d\n", p->pid);
  } else {
    printf("No alarm_trapframe to restore for process %d\n", p->pid);
  }
  return 0;
}
```

Context in `trap.c`

The `sys_sigreturn` function is called when the user process returns from a signal handler. The `usertrap` function in `trap.c` handles traps from user space, including system calls like `sys_sigreturn`.

Potential Issues

1. **Trapframe Restoration**: The function restores the process's trapframe from `alarm_trapframe`. If `alarm_trapframe` is not correctly set or if the memory is corrupted, this could cause issues.
2. **Memory Management**: The function uses `kfree` to free the `alarm_trapframe`. If `alarm_trapframe` is already freed or not allocated, this could cause a double-free or invalid free error.
3. **State Management**: The function does not check if the process is in a valid state to restore the trapframe. If the process state is not correctly managed, this could cause inconsistencies.

Debugging Steps

1. **Check Trapframe Allocation**: Ensure that `alarm_trapframe` is correctly allocated and set before `sys_sigreturn` is called.
2. **Check Memory Management**: Ensure that `alarm_trapframe` is not freed multiple times and is correctly managed.

3. **Check Process State**: Ensure that the process state is correctly managed before and after `sys_sigreturn` is called.

Adding Debugging Information

You can add more detailed debugging information to `sys_sigreturn` to help identify the issue:

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  printf("sys_sigreturn called for process %d\n", p->pid);
  if(p->alarm_trapframe) {
    printf("Restoring trapframe for process %d\n", p->pid);
    printf("Current trapframe: epc=%p, sp=%p\n", (void*)p->trapframe->epc,
(void*)p->trapframe->sp);
    printf("Alarm trapframe: epc=%p, sp=%p\n", (void*)p->alarm_trapframe->epc,
(void*)p->alarm_trapframe->sp);
    memmove(p->trapframe, p->alarm_trapframe, PGSIZE);
    kfree(p->alarm_trapframe);
    p->alarm_trapframe = 0;
    printf("Trapframe restored for process %d\n", p->pid);
    printf("Restored trapframe: epc=%p, sp=%p\n", (void*)p->trapframe->epc,
(void*)p->trapframe->sp);
  } else {
    printf("No alarm_trapframe to restore for process %d\n", p->pid);
  }
  return 0;
}
```

Summary

- Ensure that `alarm_trapframe` is correctly allocated and set before `sys_sigreturn` is called.
- Ensure that `alarm_trapframe` is not freed multiple times and is correctly managed.
- Ensure that the process state is correctly managed before and after `sys_sigreturn` is called.
- Add detailed debugging information to `sys_sigreturn` to help identify the issue.

By following these steps, you should be able to identify and resolve the issue with `sys_sigreturn`.

**I am getting this error when trying to run printhell (the user command that tests my alarm implementation in xv6)

scause=0xf sepc=0x80000d52 stval=0x0
panic: kerneltrap**

The error message indicates that a trap occurred in the kernel, which is unexpected and causes a panic. The `scause` value `0xf` indicates a specific type of trap, and `sepc` points to the program counter where the trap occurred.

Steps to Diagnose and Fix the Issue

1. **Identify the Trap Type**: The `scause` value `0xf` needs to be decoded to understand the type of trap.
2. **Check the `sepc` Value**: The `sepc` value `0x80000d52` points to the instruction that caused the trap. You can use this to locate the exact line in the code.
3. **Review the Code Around the Trap**: Check the code around the `sepc` value to understand what might be causing the trap.

Step-by-Step Diagnosis

1. Identify the Trap Type

The `scause` value `0xf` corresponds to a specific trap type. In RISC-V, the `scause` register encodes the cause of the trap. The lower bits indicate the trap type:

- `0x0` to `0xf`: Synchronous exceptions (e.g., illegal instruction, page fault)
- `0x8000000000000000` bit set: Asynchronous interrupts

The value `0xf` indicates a synchronous exception. Specifically, `0xf` corresponds to an "Environment call from M-mode" (Machine mode), which is unexpected in supervisor mode.

2. Check the `sepc` Value

The `sepc` value `0x80000d52` points to the instruction that caused the trap. You can use the disassembly of the kernel to locate this instruction.

3. Review the Code Around the Trap

Locate the instruction at `0x80000d52` in the disassembly or source code to understand what might be causing the trap.

Potential Issues

1. **Invalid Memory Access**: The trap could be caused by an invalid memory access, such as dereferencing a null or invalid pointer.
2. **Illegal Instruction**: The trap could be caused by executing an illegal instruction.
3. **Environment Call from M-mode**: The trap could be caused by an environment call from machine mode, which is unexpected.

Debugging Steps

1. **Add Debugging Information**: Add more detailed debugging information to the code to understand the state of the process and the trapframe.
2. **Check Memory Accesses**: Ensure that all memory accesses are valid and that pointers are correctly initialized.
3. **Review Environment Calls**: Ensure that environment calls are made from the correct privilege level.

Example Debugging Code

Add detailed debugging information to the `usertrap` and `kerneltrap` functions:

```
void
usertrap(void)
{
  int which_dev = 0;
  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");
  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);
  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();
  printf("usertrap: scause=%lx sepc=%lx stval=%lx\n", r_scause(), r_sepc(),
r_stval());

  if(r_scause() == 8){
    // system call
    if(killed(p))
      exit(-1);
    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;
    // an interrupt will change sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();
    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    printf("usertrap(): unexpected scause %lx pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%lx stval=%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }
  if(killed(p))
    exit(-1);
  if(which_dev == 2)
    yield();
  usertrapret();
}

void
kerneltrap()
{
  int which_dev = 0;
  uint64 sepc = r_sepc();
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();
```

```
  printf("kerneltrap: scause=%lx sepc=%lx stval=%lx\n", scause, sepc,
r_stval());

  if((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if(intr_get() != 0)
    panic("kerneltrap: interrupts enabled");
  if((which_dev = devintr()) == 0){
    // interrupt or trap from an unknown source
    printf("scause=0x%lx sepc=0x%lx stval=0x%lx\n", scause, r_sepc(),
r_stval());
    panic("kerneltrap");
  }
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2 && myproc() != 0)
    yield();
  // the yield() may have caused some traps to occur,
  // so restore trap registers for use by kernelvec.S's sepc instruction.
  w_sepc(sepc);
  w_sstatus(sstatus);
}
```

Summary

- The `scause` value `0xf` indicates a synchronous exception, specifically an "Environment call from M-mode".
- The `sepc` value `0x80000d52` points to the instruction that caused the trap.
- Add detailed debugging information to the `usertrap` and `kerneltrap` functions to understand the state of the process and the trapframe.
- Ensure that all memory accesses are valid and that pointers are correctly initialized.
- Ensure that environment calls are made from the correct privilege level.

By following these steps, you should be able to identify and resolve the issue causing the kernel trap.

---

*Explain this enqueue function in proc.c
void enqueue(int priority, struct proc p) {
int rear = mlfq[priority].rear;
mlfq[priority].queue[rear] = p;
mlfq[priority].rear = (rear + 1) % NPROC;
printf("Enqueued process %d to queue %d\n", p->pid, priority); // debug line
print_all_queues(); // debug line
}

**

The `enqueue` function in

is responsible for adding a process to a specific priority queue in a Multi-Level Feedback Queue (MLFQ) scheduler. Here's a step-by-step explanation of how it works:

Function Signature

```c
void enqueue(int priority, struct proc* p)
```

- **Parameters**:
  - `priority`: The priority level of the queue to which the process should be added.
  - `p`: A pointer to the process (`struct proc`) that needs to be enqueued.

Function Body

1. **Retrieve the Rear Index**:

   ```c
   int rear = mlfq[priority].rear;
   ```

   - The `rear` variable is assigned the current rear index of the queue at the specified priority level. This index indicates where the next process will be added in the queue.

2. **Add the Process to the Queue**:

   ```c
   mlfq[priority].queue[rear] = p;
   ```

   - The process `p` is added to the queue at the position indicated by the `rear` index.

3. **Update the Rear Index**:

   ```c
   mlfq[priority].rear = (rear + 1) % NPROC;
   ```


   - The `rear` index is incremented by 1 and then taken modulo `NPROC` (the maximum number of processes). This ensures that the index wraps around if it reaches the end of the queue array, maintaining a circular queue.

4. **Debugging Information**:

   ```c
   printf("Enqueued process %d to queue %d\n", p->pid, priority); // debug line
   print_all_queues();   // debug line
   ```

   - A debug message is printed to indicate that the process has been enqueued, showing the process ID (`p->pid`) and the priority level of the queue.

- The `print_all_queues` function is called to print the state of all queues, which is useful for debugging.

Summary

The `enqueue` function adds a process to the appropriate priority queue in the MLFQ scheduler. It updates the rear index of the queue to ensure that the next process is added to the correct position. The function also includes debug lines to print information about the enqueued process and the state of all queues, which helps in tracking the behavior of the scheduler during development and debugging.

**write a script to count the number of numbers over here :
1 2 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 2 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56
56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 56 2 56 56 56 56 56 56 56 56 2
**

Here's a simple Python script to count the occurrences of each number in the given sequence:

```
numbers = [1, 2, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56,
56, 56, 2, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56,
56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 56, 2, 56,
56, 56, 56, 56, 56, 56, 56, 2]

# Create a dictionary to store the count of each number
count_dict = {}

# Iterate over each number in the list
for number in numbers:
    if number in count_dict:
        count_dict[number] += 1
    else:
        count_dict[number] = 1

# Print the count of each number
for number, count in count_dict.items():
    print(f"Number {number} occurs {count} times.")
```


Explanation

1. **Initialize the List**: The list `numbers` contains the sequence of numbers.
2. **Create a Dictionary**: `count_dict` is used to store the count of each number.
3. **Iterate Over the List**: For each number in the list, check if it is already in the dictionary:
   - If it is, increment its count.
   - If it is not, add it to the dictionary with a count of 1.
4. **Print the Results**: Iterate over the dictionary and print the count of each number.

---

how to run qemu with 1 cpu ?
MLFQ must be tested with a single cpu

To run QEMU with a single CPU, you can use the `-smp` option to specify the number of CPUs. For testing the Multi-Level Feedback Queue (MLFQ) scheduler with a single CPU, you should set this option to 1.

Here is the command to run QEMU with a single CPU:

```
qemu-system-riscv64 -smp 1 -machine virt -kernel kernel -nographic -bios none
```

Explanation

- `qemu-system-riscv64`: The QEMU system emulator for RISC-V 64-bit architecture.
- `-smp 1`: Specifies that QEMU should use a single CPU.
- `-machine virt`: Specifies the machine type as `virt`, which is a virtual machine.
- `-kernel kernel`: Specifies the kernel image to boot.
- `-nographic`: Disables graphical output and redirects the serial I/O to the terminal.
- `-bios none`: Specifies that no BIOS should be used.

Make sure to adjust the paths and options according to your specific setup and requirements.

---

using make qemu

To run QEMU with a single CPU using `make qemu`, you can modify the

to include the `-smp 1` option. Here are the steps to do this:

1. **Open the

**: Locate and open the

in your project directory.

2. **Modify the QEMU Command**: Find the line that starts with `QEMUOPTS` or the line that runs QEMU, and add the `-smp 1` option to it.

Example Modification

Here is an example of how you might modify the

:

```
QEMUOPTS = -machine virt -kernel kernel -nographic -bios none -smp 1
```

If your

has a specific target for running QEMU, it might look something like this:

```
qemu:
	qemu-system-riscv64 $(QEMUOPTS)
```

Full Example

Here is a full example of what the relevant part of your

might look like after modification:

```
QEMUOPTS = -machine virt -kernel kernel -nographic -bios none -smp 1

qemu:
	qemu-system-riscv64 $(QEMUOPTS)
```

3. **Run QEMU**: Use the `make qemu` command to run QEMU with the specified options.

```
make qemu
```

This will start QEMU with a single CPU, allowing you to test the Multi-Level Feedback Queue (MLFQ) scheduler in a single-CPU environment.

**how to run make qemu with 1 cpu ? MLFQ must be tested with a single cpu **

To run `make qemu` with a single CPU, you can modify the `CPUS` variable in the Makefile or override it directly from the command line. Since you want to test the Multi-Level Feedback Queue (MLFQ) scheduler with a single CPU, you should set `CPUS` to 1.

Option 1: Modify the Makefile

You can set the `CPUS` variable to 1 directly in the Makefile. Locate the section where `CPUS` is defined and change its value to 1.

```
ifndef CPUS
CPUS := 1
endif
```

Option 2: Override from the Command Line

You can override the `CPUS` variable directly from the command line when running `make qemu`.

```
make qemu CPUS=1
```

Example Makefile Modification

Here is the relevant part of the Makefile with the `CPUS` variable set to 1:

```
# try to generate a unique GDB port
GDBPORT = $(shell expr `id -u` % 5000 + 25000)
# QEMU's gdb stub command line changed in 0.11
QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \
	then echo "-gdb tcp::$(GDBPORT)"; \
	else echo "-s -p $(GDBPORT)"; fi)
ifndef CPUS
CPUS := 1
endif

QEMUOPTS = -machine virt -bios none -kernel $K/kernel -m 128M -smp $(CPUS) -nographic
QEMUOPTS += -global virtio-mmio.force-legacy=false
QEMUOPTS += -drive file=fs.img,if=none,format=raw,id=x0
QEMUOPTS += -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

qemu: $K/kernel fs.img
	$(QEMU) $(QEMUOPTS)
```

Running the Command

After making the changes, you can run `make qemu` to start QEMU with a single CPU:

```
make qemu
```


Alternatively, if you chose not to modify the Makefile, you can run:

```
make qemu CPUS=1
```

This will ensure that QEMU is started with a single CPU, allowing you to test the MLFQ scheduler in a single-CPU environment.

MLFQ scheduler cannot find any processes

If the MLFQ scheduler cannot find any processes, it could be due to several reasons. Here are some steps to debug and potentially fix the issue:

1. **Ensure Processes are Enqueued**: Verify that processes are being enqueued into the MLFQ queues correctly. Check the `enqueue` function and ensure that it is being called appropriately when processes become runnable.
2. **Check Process States**: Ensure that processes are transitioning to the `RUNNABLE` state correctly. If processes are stuck in other states (e.g., `SLEEPING`, `ZOMBIE`), they won't be picked up by the scheduler.
3. **Debugging Output**: Add more debugging output to trace the state transitions of processes and the actions of the scheduler.
4. **Initialization**: Ensure that the MLFQ data structures are initialized correctly.

Steps to Debug

1. **Verify Enqueueing**:
    - Ensure that processes are being enqueued into the MLFQ queues when they become runnable.
    - Add debug statements in the `enqueue` function to confirm that processes are being added to the queues.
2. **Check Process States**:
    - Add debug statements in the `scheduler` function to print the state of each process as it is checked.
    - Ensure that processes are transitioning to the `RUNNABLE` state when they should be.
3. **Initialization**:
    - Ensure that the MLFQ data structures are initialized correctly in the `procinit` function or wherever appropriate.

Example Debugging Output

Here is an example of how you might add debugging output to the `scheduler` function and the `enqueue` function:

`scheduler` Function

```
void scheduler(void) { // MLFQ
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    printf("Scheduler loop start\n"); // debug line
    intr_on();
```

```
    int found = 0;
    for (int i = 0; i < NUM_QUEUES; i++) {
      printf("Checking queue %d\n", i); // debug lines
      while ((p = dequeue(i)) != 0) {
        printf("Dequeued process %d from queue %d\n", p->pid, i); // debug line
        acquire(&p->lock);
        if (p->state == RUNNABLE) {
          printf("Switching to process %d\n", p->pid); // debug line
          p->state = RUNNING;
          c->proc = p;
          swtch(&c->context, &p->context);
          c->proc = 0;

          if (p->state == RUNNABLE) {
            if (p->total_ticks >= p->time_slice) {
              printf("Moving process %d to lower queue\n", p->pid); // debug line
              move_to_lower_queue(p);
            } else {
              printf("Re-enqueueing process %d to queue %d\n", p->pid, p->priority); // debug line
              enqueue(p->priority, p);
            }
          }
          found = 1;
        } else {
          printf("Process %d is not runnable, state: %d\n", p->pid, p->state); // debug line
        }
        release(&p->lock);
        if (found) break;
      }
      if (found) break;
    }

    if (!found) {
      printf("No runnable process found, waiting for interrupt\n"); // debug line
      intr_on();
      asm volatile("wfi");
    }

    global_ticks++;
    if (global_ticks >= PRIORITY_BOOST_TICKS) {
      printf("Performing priority boost\n"); // debug line
      priority_boost();
    }
  }
}
```


`enqueue` **Function**

```c
void enqueue(int priority, struct proc* p) {
  int rear = mlfq[priority].rear;
  mlfq[priority].queue[rear] = p;
  mlfq[priority].rear = (rear + 1) % NPROC;
  printf("Enqueued process %d to queue %d\n", p->pid, priority); // debug line
  print_all_queues();   // debug line
}
```

Verify Initialization

Ensure that the MLFQ data structures are initialized correctly. For example, in the `procinit` function:

```c
void procinit(void) {
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  initlock(&wait_lock, "wait_lock");
  for(p = proc; p < &proc[NPROC]; p++) {
    initlock(&p->lock, "proc");
    p->state = UNUSED;
    p->kstack = KSTACK((int) (p - proc));
  }

  // Initialize MLFQ data structures
  for (int i = 0; i < NUM_QUEUES; i++) {
    mlfq[i].front = 0;
    mlfq[i].rear = 0;
  }
}
```


Conclusion

By adding these debug statements and ensuring proper initialization, you should be able to trace the issue and determine why the MLFQ scheduler is not finding any processes. Look for any discrepancies in the process states and the enqueueing logic.

---

explain the procinit function

The `procinit` function is responsible for initializing the process table in the operating system. This function sets up the necessary data structures and locks for managing processes. Here's a detailed explanation of how it works:

Function Signature

```c
void procinit(void)
```

- This function does not take any parameters and does not return any value.

Function Body

1. **Variable Declaration**:

```c
struct proc *p;
```

   - A pointer `p` of type `struct proc` is declared. This will be used to iterate over the process table.

2. **Initialize Locks**:

```c
initlock(&pid_lock, "nextpid");
initlock(&wait_lock, "wait_lock");
```

   - Two locks are initialized:
     - `pid_lock`: This lock is used to protect the allocation of process IDs.
     - `wait_lock`: This lock is used to protect the process waiting mechanism.

3. **Initialize Process Table**:

```c
for(p = proc; p < &proc[NPROC]; p++) {
    initlock(&p->lock, "proc");
    p->state = UNUSED;
    p->kstack = KSTACK((int) (p - proc));
}
```


   - The function iterates over the process table, which is an array of `struct proc` of size `NPROC`.
   - For each process `p` in the process table:
     - `initlock(&p->lock, "proc")`: Initializes the lock for the process. This lock will be used to protect the process's state and other critical sections.
     - `p->state = UNUSED`: Sets the initial state of the process to `UNUSED`, indicating that the process slot is available.
     - `p->kstack = KSTACK((int) (p - proc))`: Sets the kernel stack pointer for the process. The `KSTACK` macro calculates the virtual address of the kernel stack for the process based on its position in the process table.

Summary

The `procinit` function initializes the process table by setting up locks and setting the initial state and kernel stack for each process. This function ensures that the process management data structures are ready for use by the operating system.

---

insteaad of having a literal queue to indicate priority, why not have priority as a field in the struct proc ?

Using a priority field in the `struct proc` instead of maintaining separate queues for each priority level can simplify the code and reduce the overhead of managing multiple queues. Here's how you can modify the code to use a priority field in the `struct proc`:

Step-by-Step Plan

1. **Remove the MLFQ Data Structures**:
    - Remove the `mlfq` array and related functions like `enqueue` and `dequeue`.
2. **Modify the `struct proc`**:
    - Ensure that the `priority` field is present in the `struct proc`.
3. **Update the Scheduler**:
    - Modify the scheduler to iterate over all processes and select the one with the highest priority.
4. **Update Process Management Functions**:
    - Ensure that functions like `fork`, `exit`, and `wait` correctly handle the priority field.

Code Changes

1. Remove MLFQ Data Structures

Remove the MLFQ-related data structures and functions from

:


```
// Remove these definitions and functions
/*
#define NUM_QUEUES 4
#define PRIORITY_BOOST_TICKS 48
struct {
  struct proc* queue[NPROC];
  int front;
  int rear;
} mlfq[NUM_QUEUES];
int time_slices[NUM_QUEUES] = {1, 4, 8, 16};
int global_ticks = 0;

void print_queue(int priority) { ... }
void print_all_queues() { ... }
void enqueue(int priority, struct proc* p) { ... }
```

```
struct proc* dequeue(int priority) { ... }
void move_to_lower_queue(struct proc* p) { ... }
void priority_boost() { ... }
*/
```

2. Ensure `priority` Field in `struct proc`

Ensure that the `priority` field is present in `struct proc` in `proc.h`:

```
struct proc {
  // ... other fields ...
  int priority;           // Current priority (0-3)
  int time_slice;         // Remaining time slice
  int total_ticks;        // Total ticks spent in the current queue
  // ... other fields ...
};
```

3. Update the Scheduler

Modify the `scheduler` function to select the process with the highest priority:

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    printf("Scheduler loop start\n"); // debug line
    intr_on();

    struct proc *highest_priority_proc = 0;
    int highest_priority = -1;

    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      p->state = RUNNING;
```

```
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          p->priority = (p->priority < 3) ? p->priority + 1 : 3; // Lower priority
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      printf("No runnable process found, waiting for interrupt\n"); // debug line
      intr_on();
      asm volatile("wfi");
    }
  }
}
```

4. Update Process Management Functions

Ensure that functions like `fork`, `exit`, and `wait` correctly handle the priority field. For example, in the `fork` function:

```
if((np = allocproc()) == 0){
  return -1;
}
// Copy user memory from parent to child.
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
  freeproc(np);
  release(&np->lock);
  return -1;
}
np->sz = p->sz;

// Copy saved user registers.
*(np->trapframe) = *(p->trapframe);
// Cause fork to return 0 in the child.
np->trapframe->a0 = 0;
// Increment reference counts on open file descriptors.
for(i = 0; i < NOFILE; i++)
  if(p->ofile[i])
    np->ofile[i] = filedup(p->ofile[i]);
np->cwd = idup(p->cwd);
safestrcpy(np->name, p->name, sizeof(p->name));
pid = np->pid;
release(&np->lock);
```

```
acquire(&wait_lock);
np->parent = p;
release(&wait_lock);
acquire(&np->lock);
np->state = RUNNABLE;
np->priority = 0; // Start with the highest priority
np->time_slice = time_slices[0];
np->total_ticks = 0;
release(&np->lock);
return pid;
```

Conclusion

By using a priority field in the `struct proc`, you can simplify the code and reduce the overhead of managing multiple queues. The scheduler can iterate over all processes and select the one with the highest priority, making the scheduling logic more straightforward.

---

give the modifies exit =, wait functions

Here are the modified `exit` and `wait` functions to handle the priority field in the `struct proc`:

Modified `exit` Function

The `exit` function is responsible for terminating the current process. It sets the process state to `ZOMBIE` and wakes up the parent process if it is waiting.

```
void exit(int status) {
  struct proc *p = myproc();
  if(p == initproc)
    panic("init exiting");

  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(p->cwd);
  end_op();
  p->cwd = 0;

  acquire(&wait_lock);

  // Give any children to init.
  reparent(p);
```

```
  // Parent might be sleeping in wait().
  wakeup(p->parent);

  acquire(&p->lock);
  p->xstate = status;
  p->state = ZOMBIE;
  release(&wait_lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```

Modified `wait` Function

The `wait` function waits for a child process to exit and returns its PID. If no children are present, it returns -1.

```
int wait(uint64 addr) {
  struct proc *pp;
  int havekids, pid;
  struct proc *p = myproc();
  acquire(&wait_lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(pp = proc; pp < &proc[NPROC]; pp++){
      if(pp->parent == p){
        // make sure the child isn't still in exit() or swtch().
        acquire(&pp->lock);
        havekids = 1;
        if(pp->state == ZOMBIE){
          // Found one.
          pid = pp->pid;
          if(addr != 0 && copyout(p->pagetable, addr, (char *)&pp->xstate,
                                  sizeof(pp->xstate)) < 0) {
            release(&pp->lock);
            release(&wait_lock);
            return -1;
          }
          freeproc(pp);
          release(&pp->lock);
          release(&wait_lock);
          return pid;
        }
        release(&pp->lock);
      }
    }
```

```
    // No point waiting if we don't have any children.
    if(!havekids || killed(p)){
      release(&wait_lock);
      return -1;
    }

    // Wait for a child to exit.
    sleep(p, &wait_lock);  // DOC: wait-sleep
  }
}
```

Summary of Changes

- `exit` **Function**:
  - Closes all open files.
  - Reparents any children to the `init` process.
  - Sets the process state to `ZOMBIE`.
  - Wakes up the parent process if it is waiting.
  - Calls `sched()` to switch to the scheduler.

- `wait` **Function**:
  - Scans the process table for exited children.
  - If a child is found in the `ZOMBIE` state, it retrieves the exit status and frees the process.
  - If no children are found, it waits for a child to exit.

These modifications ensure that the priority field is correctly handled during process termination and waiting.

**apply nessacary change for moving to lower queue **

To apply the necessary changes for moving a process to a lower priority queue without using literal queues, we need to update the `yield` function and remove the `move_to_lower_queue` function. Instead, we will directly manipulate the `priority` field in the `struct proc`.

Updated `yield` Function

The `yield` function will now directly adjust the `priority` field of the process if it has used its entire time slice.

```
// Give up the CPU for one scheduling round.
void yield(void) {
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  if (p->time_slice > 0) {
    // Process is yielding voluntarily (e.g., for I/O)
```

```
    // No need to change priority, just reset the time slice
    p->time_slice = time_slices[p->priority];
  } else {
    // Process used its entire time slice, move to lower priority
    if (p->priority < NUM_QUEUES - 1) {
      p->priority++;
    }
    p->time_slice = time_slices[p->priority];
    p->total_ticks = 0;
  }
  sched();
  release(&p->lock);
}
```

Remove `move_to_lower_queue` Function

Since we are directly manipulating the `priority` field in the `yield` function, we can remove the `move_to_lower_queue` function.

```
// Remove this function
/*
void move_to_lower_queue(struct proc* p) {
  if (p->priority < NUM_QUEUES - 1) {
    p->priority++;
  }
  p->time_slice = time_slices[p->priority];
}
*/
```

Updated `scheduler` Function

Ensure the `scheduler` function correctly handles the priority and time slice of processes.

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    printf("Scheduler loop start\n"); // debug line
    intr_on();

    struct proc *highest_priority_proc = 0;
    int highest_priority = -1;

    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
```

```
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          if (p->priority < NUM_QUEUES - 1) {
            p->priority++;
          }
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      printf("No runnable process found, waiting for interrupt\n"); // debug
line
      intr_on();
      asm volatile("wfi");
    }
  }
}
```


Summary of Changes

1. **Updated `yield` Function**:
    - Directly adjusts the `priority` field if the process has used its entire time slice.
    - Resets the `time_slice` and `total_ticks` fields accordingly.
2. **Removed `move_to_lower_queue` Function**:
    - This function is no longer needed as the `yield` function now handles priority adjustments directly.
3. **Updated `scheduler` Function**:
    - Ensures that the `priority` and `time_slice` fields are correctly handled when a process is selected to run.

These changes simplify the code by removing the need for separate queue management functions and directly using the `priority` field in the `struct proc`.

---

apply nessacary change for moving to enqueue

To apply the necessary changes for removing the `enqueue` function and directly setting the priority and time slice in the `struct proc`, we need to update the code where `enqueue` is called. Specifically, we will set the `priority`, `time_slice`, and `total_ticks` fields directly.

Updated Code

1. Remove `enqueue` Function Call

Replace the `enqueue` function call with direct assignments to the `priority`, `time_slice`, and `total_ticks` fields in the `struct proc`.

```
// Set up new context to start executing at forkret,
// which returns to user space.
memset(&p->context, 0, sizeof(p->context));
p->context.ra = (uint64)forkret;
p->context.sp = p->kstack + PGSIZE;

// Initialize process priority and time slice
p->priority = 0;  // Start in the highest priority
p->time_slice = time_slices[0];
p->total_ticks = 0;

// Return the process
return p;
```

Summary of Changes

1. **Removed `enqueue` Function Call**:
   - Directly set the `priority`, `time_slice`, and `total_ticks` fields in the `struct proc`.
2. **Updated Process Initialization**:
   - Ensure that new processes are initialized with the correct priority and time slice.

These changes simplify the code by removing the need for the `enqueue` function and directly using the `priority` field in the `struct proc`.

---

infifnite loop of No runnable process found, waiting for interrupt

The infinite loop of "No runnable process found, waiting for interrupt" suggests that the scheduler is not finding any processes in the `RUNNABLE` state. This could be due to several reasons, such as processes not being correctly set to `RUNNABLE` or an issue with the process state transitions.

Let's ensure that processes are correctly set to `RUNNABLE` and that the scheduler is properly handling process states.

Steps to Diagnose and Fix

1. **Ensure Processes are Set to `RUNNABLE`**:
   - Verify that processes are correctly set to `RUNNABLE` when they should be.
2. **Check Scheduler Logic**:
   - Ensure the scheduler correctly identifies `RUNNABLE` processes and switches to them.
3. **Debugging Output**:
   - Add debugging output to track process states and transitions.

Updated Code

1. Ensure Processes are Set to `RUNNABLE`

Check the `yield` and `exit` functions to ensure processes are correctly set to `RUNNABLE`.

```
// Give up the CPU for one scheduling round.
void yield(void) {
  struct proc *p = myproc();
  acquire(&p->lock);
  p->state = RUNNABLE;
  if (p->time_slice > 0) {
    // Process is yielding voluntarily (e.g., for I/O)
    p->time_slice = time_slices[p->priority];
  } else {
    // Process used its entire time slice, move to lower priority
    if (p->priority < NUM_QUEUES - 1) {
      p->priority++;
    }
    p->time_slice = time_slices[p->priority];
    p->total_ticks = 0;
  }
  sched();
  release(&p->lock);
}

// Exit the current process. Does not return.
void exit(int status) {
  struct proc *p = myproc();
  if(p == initproc)
    panic("init exiting");
```

```
  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(p->cwd);
  end_op();
  p->cwd = 0;

  acquire(&wait_lock);

  // Give any children to init.
  reparent(p);

  // Parent might be sleeping in wait().
  wakeup(p->parent);

  acquire(&p->lock);
  p->xstate = status;
  p->state = ZOMBIE;
  release(&wait_lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```

2. Check Scheduler Logic

Ensure the scheduler correctly identifies `RUNNABLE` processes and switches to them.

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    printf("Scheduler loop start\n"); // debug line
    intr_on();

    struct proc *highest_priority_proc = 0;
    int highest_priority = -1;

    for (p = proc; p < &proc[NPROC]; p++) {
```

```
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          if (p->priority < NUM_QUEUES - 1) {
            p->priority++;
          }
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      printf("No runnable process found, waiting for interrupt\n"); // debug line
      intr_on();
      asm volatile("wfi");
    }
  }
}
```


3. Add Debugging Output

Add debugging output to track process states and transitions.

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    printf("Scheduler loop start\n"); // debug line
    intr_on();

    struct proc *highest_priority_proc = 0;
```

```
    int highest_priority = -1;

    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      printf("Switching to process %d with priority %d\n", p->pid, p->priority); // debug line
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          if (p->priority < NUM_QUEUES - 1) {
            p->priority++;
          }
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      printf("No runnable process found, waiting for interrupt\n"); // debug line
      intr_on();
      asm volatile("wfi");
    }
  }
}
```


Summary

1. **Ensure Processes are Set to `RUNNABLE`**:
   - Verify that processes are correctly set to `RUNNABLE` in the `yield` and `exit` functions.
2. **Check Scheduler Logic**:
   - Ensure the scheduler correctly identifies `RUNNABLE` processes and switches to them.
3. **Add Debugging Output**:

- Add debugging output to track process states and transitions.

These changes should help diagnose and fix the issue with the scheduler not finding any runnable processes.

MLFQ Analysis [5 points] (part of report) Create a timeline graph that shows which queue a process is in over time. Vary the length of time that each process consumes the CPU before willingly quitting using schedulertest. The graph should be a timeline/scatter plot between queue_id on the y-axis and time elapsed on the x-axis from start with color-coded processes. Ensure that the priority boost is visible in this graph. Below is a reference graph (note that it does not implement priority boost but aging, this is just for you to understand the format of the graph):

To create a timeline graph that shows which queue a process is in over time, you can follow these steps:

1. **Collect Data**: Modify the scheduler to log the queue (priority) of each process at each time step.
2. **Run Tests**: Use `schedulertest` to vary the length of time each process consumes the CPU.
3. **Generate Graph**: Use a plotting library like `matplotlib` in Python to create the timeline/scatter plot.

Step 1: Modify the Scheduler to Log Data

Modify the `scheduler` function to log the queue (priority) of each process at each time step. You can use a global array to store this data.


```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "spinlock.h"
#include "proc.h"
#include "defs.h"
#include "number_syscall.h"

#define MAX_LOG_ENTRIES 10000

struct log_entry {
  int time;
  int pid;
  int priority;
};

struct log_entry log[MAX_LOG_ENTRIES];
int log_index = 0;
int current_time = 0;

void log_process(int pid, int priority) {
  if (log_index < MAX_LOG_ENTRIES) {
    log[log_index].time = current_time;
    log[log_index].pid = pid;
```

```c
    log[log_index].priority = priority;
    log_index++;
  }
}

void scheduler(void) {
  printf("MLFQ SCHEDULER\n");
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    intr_on();

    struct proc *highest_priority_proc = 0;
    int highest_priority = -1;

    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      p->state = RUNNING;
      c->proc = p;
      log_process(p->pid, p->priority); // Log the process
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          if (p->priority < NUM_QUEUES - 1) {
            p->priority++;
          }
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      printf("No runnable process found, waiting for interrupt\n"); // debug line
      intr_on();
      asm volatile("wfi");
    }
```

```
        current_time++;
      }
    }
```

Step 2: Run Tests

Use `schedulertest` to vary the length of time each process consumes the CPU. Ensure that the priority boost is visible in the log data.

Step 3: Generate Graph

Use Python and `matplotlib` to create the timeline/scatter plot.

```python
import matplotlib.pyplot as plt
import pandas as pd

# Load log data
log_data = [
    # Replace this with the actual log data collected from the scheduler
    {"time": 0, "pid": 1, "priority": 0},
    {"time": 1, "pid": 1, "priority": 0},
    {"time": 2, "pid": 2, "priority": 1},
    # ...
]

# Convert to DataFrame
df = pd.DataFrame(log_data)

# Plot
fig, ax = plt.subplots()
for pid in df['pid'].unique():
    df_pid = df[df['pid'] == pid]
    ax.scatter(df_pid['time'], df_pid['priority'], label=f'PID {pid}', s=10)

ax.set_xlabel('Time')
ax.set_ylabel('Queue (Priority)')
ax.set_title('Process Queue Over Time')
ax.legend()
plt.show()
```


Summary

1. **Modify the Scheduler**: Log the queue (priority) of each process at each time step.
2. **Run Tests**: Use `schedulertest` to vary the length of time each process consumes the CPU.
3. **Generate Graph**: Use Python and `matplotlib` to create the timeline/scatter plot.

This approach will help you visualize the behavior of the MLFQ scheduler, including the priority boost mechanism.

there is no schedulertest in this latest version of xv6

If `schedulertest` is not available in your version of xv6, you can create a custom test to simulate different process behaviors and collect the necessary data for the timeline graph.

Step 1: Modify the Scheduler to Log Data

First, modify the scheduler to log the queue (priority) of each process at each time step.

```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "spinlock.h"
#include "proc.h"
#include "defs.h"
#include "number_syscall.h"

#define MAX_LOG_ENTRIES 10000

struct log_entry {
  int time;
  int pid;
  int priority;
};

struct log_entry log[MAX_LOG_ENTRIES];
int log_index = 0;
int current_time = 0;

void log_process(int pid, int priority) {
  if (log_index < MAX_LOG_ENTRIES) {
    log[log_index].time = current_time;
    log[log_index].pid = pid;
    log[log_index].priority = priority;
    log_index++;
  }
}

void scheduler(void) {
  printf("MLFQ SCHEDULER\n");
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    intr_on();

    struct proc *highest_priority_proc = 0;
```

```
    int highest_priority = -1;

    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      p->state = RUNNING;
      c->proc = p;
      log_process(p->pid, p->priority); // Log the process
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          if (p->priority < NUM_QUEUES - 1) {
            p->priority++;
          }
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      printf("No runnable process found, waiting for interrupt\n"); // debug line
      intr_on();
      asm volatile("wfi");
    }

    current_time++;
  }
}
```


Step 2: Create a Custom Test

Create a custom test program that simulates different process behaviors. This program should create multiple processes that consume the CPU for varying lengths of time before yielding.

Create a new file `schedulertest.c` in the

directory:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void busy_wait(int ticks) {
  int start = uptime();
  while (uptime() - start < ticks) {
    // Busy wait
  }
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(2, "Usage: schedulertest <num_processes>\n");
    exit(1);
  }

  int num_processes = atoi(argv[1]);
  for (int i = 0; i < num_processes; i++) {
    if (fork() == 0) {
      // Child process
      int ticks = (i + 1) * 10; // Vary the length of time
      while (1) {
        busy_wait(ticks);
        yield();
      }
    }
  }

  // Parent process waits for a while
  sleep(1000);
  exit(0);
}
```


Step 3: Compile and Run the Test

Add the new test to the

:

```
UPROGS=\
  _cat\
  _echo\
  _forktest\
  _grep\
  _init\
  _kill\
  _ln\
  _ls\
  _mkdir\
```

```
  _rm\
  _sh\
  _stressfs\
  _usertests\
  _wc\
  _zombie\
  _schedulertest\  # Add this line
```

Compile and run the test:

```
make qemu
```

In the xv6 shell, run the test:

```
schedulertest 5
```

Step 4: Extract and Plot the Data

After running the test, extract the log data from the kernel and save it to a file. You can add a system call to retrieve the log data or print it directly from the kernel.

For simplicity, let's print the log data directly from the kernel:

```
void print_log(void) {
  for (int i = 0; i < log_index; i++) {
    printf("Time: %d, PID: %d, Priority: %d\n", log[i].time, log[i].pid,
log[i].priority);
  }
}
```


Call `print_log` at the end of the test:

```
int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(2, "Usage: schedulertest <num_processes>\n");
    exit(1);
  }

  int num_processes = atoi(argv[1]);
  for (int i = 0; i < num_processes; i++) {
    if (fork() == 0) {
      // Child process
      int ticks = (i + 1) * 10; // Vary the length of time
      while (1) {
```

```
        busy_wait(ticks);
        yield();
      }
    }
  }

  // Parent process waits for a while
  sleep(1000);
  print_log(); // Print the log data
  exit(0);
}
```

Step 5: Generate the Graph

Save the log data to a file and use Python and `matplotlib` to create the timeline/scatter plot.

```python
import matplotlib.pyplot as plt
import pandas as pd

# Load log data from file
log_data = []
with open('log.txt', 'r') as f:
    for line in f:
        parts = line.strip().split(',')
        log_data.append({
            'time': int(parts[0].split(':')[1].strip()),
            'pid': int(parts[1].split(':')[1].strip()),
            'priority': int(parts[2].split(':')[1].strip())
        })

# Convert to DataFrame
df = pd.DataFrame(log_data)

# Plot
fig, ax = plt.subplots()
for pid in df['pid'].unique():
    df_pid = df[df['pid'] == pid]
    ax.scatter(df_pid['time'], df_pid['priority'], label=f'PID {pid}', s=10)

ax.set_xlabel('Time')
ax.set_ylabel('Queue (Priority)')
ax.set_title('Process Queue Over Time')
ax.legend()
plt.show()
```


Summary

1. **Modify the Scheduler**: Log the queue (priority) of each process at each time step.
2. **Create a Custom Test**: Simulate different process behaviors.

3. **Compile and Run the Test**: Add the test to the

and run it.
4. **Extract and Plot the Data**: Print the log data and use Python to create the timeline/scatter plot.

This approach will help you visualize the behavior of the MLFQ scheduler, including the priority boost mechanism.

---

support both Round Robin and MLFQ schedures using compile filags
Modify the makefile to support the SCHEDULER macro to compile the specified scheduling algorithm. Use the flags for compilation:-
Your compilation process should look something like this: make clean; make qemu SCHEDULER=MLFQ.

To support both Round Robin (RR) and Multi-Level Feedback Queue (MLFQ) schedulers using compile flags, you need to:

1. Modify the Makefile to accept the `SCHEDULER` macro.
2. Use conditional compilation in your code to switch between the two scheduling algorithms based on the `SCHEDULER` macro.

Step 1: Modify the Makefile

Update the Makefile to accept the `SCHEDULER` macro and set the appropriate compile flags.

```
K=kernel
U=user

OBJS = \
  $K/entry.o \
  $K/start.o \
  $K/console.o \
  $K/printf.o \
  $K/uart.o \
  $K/kalloc.o \
  $K/spinlock.o \
  $K/string.o \
  $K/main.o \
  $K/vm.o \
  $K/proc.o \
  $K/swtch.o \
  $K/trampoline.o \
  $K/trap.o \
  $K/syscall.o \
  $K/sysproc.o \
  $K/bio.o \
  $K/fs.o \
  $K/log.o \
  $K/sleeplock.o \
```

```
  $K/file.o \
  $K/pipe.o \
  $K/exec.o \
  $K/sysfile.o \
  $K/kernelvec.o \
  $K/plic.o \
  $K/virtio_disk.o

# check scheduler
SCHEDULER ?= RR

ifeq ($(SCHEDULER),MLFQ)
    CFLAGS += -DMLFQ
else ifeq ($(SCHEDULER),RR)
    CFLAGS += -DRR
endif

# riscv64-unknown-elf- or riscv64-linux-gnu-
# perhaps in /opt/riscv/bin
#TOOLPREFIX =

# Try to infer the correct TOOLPREFIX if not set
ifndef TOOLPREFIX
TOOLPREFIX := $(shell if riscv64-unknown-elf-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
    then echo 'riscv64-unknown-elf-'; \
    elif riscv64-linux-gnu-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
    then echo 'riscv64-linux-gnu-'; \
    elif riscv64-unknown-linux-gnu-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
    then echo 'riscv64-unknown-linux-gnu-'; \
    else echo "***" 1>&2; \
    echo "*** Error: Couldn't find a riscv64 version of GCC/binutils." 1>&2; \
    echo "*** To turn off this error, run 'gmake TOOLPREFIX= ...'." 1>&2; \
    echo "***" 1>&2; exit 1; fi)
endif

QEMU = qemu-system-riscv64

CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
OBJCOPY = $(TOOLPREFIX)objcopy
OBJDUMP = $(TOOLPREFIX)objdump

CFLAGS = -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2
CFLAGS += -MD
CFLAGS += -mcmodel=medany
# CFLAGS += -ffreestanding -fno-common -nostdlib -mno-relax
CFLAGS += -fno-common -nostdlib
CFLAGS += -fno-builtin-strncpy -fno-builtin-strncmp -fno-builtin-strlen -fno-builtin-memset
```

```
CFLAGS += -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-
builtin-bzero
CFLAGS += -fno-builtin-strchr -fno-builtin-exit -fno-builtin-malloc -fno-
builtin-putc
CFLAGS += -fno-builtin-free
CFLAGS += -fno-builtin-memcpy -Wno-main
CFLAGS += -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf
CFLAGS += -I.
CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1
&& echo -fno-stack-protector)

# Disable PIE when possible (for Ubuntu 16.10 toolchain)
ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]no-pie'),)
CFLAGS += -fno-pie -no-pie
endif
ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]nopie'),)
CFLAGS += -fno-pie -nopie
endif

LDFLAGS = -z max-page-size=4096

$K/kernel: $(OBJS) $K/kernel.ld $U/initcode
    $(LD) $(LDFLAGS) -T $K/kernel.ld -o $K/kernel $(OBJS)
    $(OBJDUMP) -S $K/kernel > $K/kernel.asm
    $(OBJDUMP) -t $K/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
$K/kernel.sym

$U/initcode: $U/initcode.S
    $(CC) $(CFLAGS) -march=rv64g -nostdinc -I. -Ikernel -c $U/initcode.S -o
$U/initcode.o
    $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o $U/initcode.out $U/initcode.o
    $(OBJCOPY) -S -O binary $U/initcode.out $U/initcode
    $(OBJDUMP) -S $U/initcode.o > $U/initcode.asm

tags: $(OBJS) _init
    etags *.S *.c

ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o

_%: %.o $(ULIB)
    $(LD) $(LDFLAGS) -T $U/user.ld -o $@ $^
    $(OBJDUMP) -S $@ > $*.asm
    $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym

$U/usys.S : $U/usys.pl
    perl $U/usys.pl > $U/usys.S

$U/usys.o : $U/usys.S
    $(CC) $(CFLAGS) -c -o $U/usys.o $U/usys.S

$U/_forktest: $U/forktest.o $(ULIB)
    # forktest has less library code linked in - needs to be small
    # in order to be able to max out the proc table.
```

```
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $U/_forktest $U/forktest.o
$U/ulib.o $U/usys.o
	$(OBJDUMP) -S $U/_forktest > $U/forktest.asm

mkfs/mkfs: mkfs/mkfs.c $K/fs.h $K/param.h
	gcc -Werror -Wall -I. -o mkfs/mkfs mkfs/mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_syscount\
	$U/_printhell\

fs.img: mkfs/mkfs README $(UPROGS)
	mkfs/mkfs fs.img README $(UPROGS)

-include kernel/*.d user/*.d

clean:
	rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
	*/*.o */*.d */*.asm */*.sym \
	$U/initcode $U/initcode.out $K/kernel fs.img \
	mkfs/mkfs .gdbinit \
		$U/usys.S \
	$(UPROGS)

# try to generate a unique GDB port
GDBPORT = $(shell expr `id -u` % 5000 + 25000)
# QEMU's gdb stub command line changed in 0.11
QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \
	then echo "-gdb tcp::$(GDBPORT)"; \
	else echo "-s -p $(GDBPORT)"; fi)
ifndef CPUS
```

```
CPUS := 3
endif

QEMUOPTS = -machine virt -bios none -kernel $K/kernel -m 128M -smp $(CPUS) -
nographic
QEMUOPTS += -global virtio-mmio.force-legacy=false
QEMUOPTS += -drive file=fs.img,if=none,format=raw,id=x0
QEMUOPTS += -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

qemu: $K/kernel fs.img
    $(QEMU) $(QEMUOPTS)

.gdbinit: .gdbinit.tmpl-riscv
    sed "s/:1234/:$(GDBPORT)/" < $^ > $@

qemu-gdb: $K/kernel .gdbinit fs.img
    @echo "*** Now run 'gdb' in another window." 1>&2
    $(QEMU) $(QEMUOPTS) -S $(QEMUGDB)
```

Step 2: Use Conditional Compilation in Your Code

Modify the

file to use conditional compilation to switch between the RR and MLFQ schedulers.

```c
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "spinlock.h"
#include "proc.h"
#include "defs.h"
#include "number_syscall.h"

struct cpu cpus[NCPU];
struct proc proc[NPROC];

void print_proc_array() {
  static char *states[] = {
    [UNUSED]    "unused",
    [USED]      "used",
    [SLEEPING]  "sleeping",
    [RUNNABLE]  "runnable",
    [RUNNING]   "running",
    [ZOMBIE]    "zombie"
  };
  for (int i = 0; i < NPROC; i++) {
    char *state = (proc[i].state >= 0 && proc[i].state < NELEM(states) &&
states[proc[i].state]) ? states[proc[i].state] : "???";
    printf("Process %d: PID=%d, State=%s, Priority=%d\n", i, proc[i].pid,
```

```
state, proc[i].priority);
  }
  printf("\n\n");
}

struct proc *initproc;
int nextpid = 1;
struct spinlock pid_lock;
extern void forkret(void);
static void freeproc(struct proc *p);
extern char trampoline[]; // trampoline.S

// helps ensure that wakeups of wait()ing
// parents are not lost. helps obey the
// memory model when using p->parent.
// must be acquired before any p->lock.
struct spinlock wait_lock;

#define NUM_QUEUES 4
#define PRIORITY_BOOST_TICKS 48

int time_slices[NUM_QUEUES] = {1, 4, 8, 16};
int global_ticks = 0;

void RR_scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    intr_on();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
        found = 1;
      }
      release(&p->lock);
    }

    if(found == 0) {
      // No runnable process found, wait for an interrupt
      intr_on();
      asm volatile("wfi");
    }
  }
}
```

```c
void MLFQ_scheduler(void) {
  printf("MLFQ SCHEDULER\n");
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    intr_on();

    struct proc *highest_priority_proc = 0;
    int highest_priority = -1;

    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          if (p->priority < NUM_QUEUES - 1) {
            p->priority++;
          }
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      intr_on();
      asm volatile("wfi");
    }
  }
}

void scheduler(void) {
#ifdef MLFQ
  MLFQ_scheduler();
#elif defined(RR)
  RR_scheduler();
```

```
#else
#error "No scheduler defined"
#endif
}
```

Step 3: Compile and Run

To compile and run with the desired scheduler, use the following commands:

For Round Robin (RR) scheduler:

```
make clean
make qemu SCHEDULER=RR
```

For Multi-Level Feedback Queue (MLFQ) scheduler:

```
make clean
make qemu SCHEDULER=MLFQ
```

Summary

1. **Modify the Makefile**: Add conditional compilation flags based on the `SCHEDULER` macro.
2. **Use Conditional Compilation in Code**: Switch between the RR and MLFQ schedulers using `#ifdef` directives.
3. **Compile and Run**: Use the `SCHEDULER` macro to specify the desired scheduler during compilation.

This setup allows you to easily switch between different scheduling algorithms by specifying the `SCHEDULER` macro during the build process.

PROFI

Makefile:93: * missing separator. Stop.**
Makefile:93: * missing separator. Stop.**

The error "missing separator" typically indicates that there is an issue with the indentation in the Makefile. Make sure that all commands are indented with a tab character, not spaces.

Let's review the relevant sections of the Makefile and ensure proper indentation. Here is the corrected Makefile with the necessary changes to support the `SCHEDULER` macro:

```
K=kernel
U=user

OBJS = \
  $K/entry.o \
  $K/start.o \
```

```
  $K/console.o \
  $K/printf.o \
  $K/uart.o \
  $K/kalloc.o \
  $K/spinlock.o \
  $K/string.o \
  $K/main.o \
  $K/vm.o \
  $K/proc.o \
  $K/swtch.o \
  $K/trampoline.o \
  $K/trap.o \
  $K/syscall.o \
  $K/sysproc.o \
  $K/bio.o \
  $K/fs.o \
  $K/log.o \
  $K/sleeplock.o \
  $K/file.o \
  $K/pipe.o \
  $K/exec.o \
  $K/sysfile.o \
  $K/kernelvec.o \
  $K/plic.o \
  $K/virtio_disk.o

# check scheduler
SCHEDULER ?= RR

ifeq ($(SCHEDULER),MLFQ)
    CFLAGS += -DMLFQ
else ifeq ($(SCHEDULER),RR)
    CFLAGS += -DRR
endif

# riscv64-unknown-elf- or riscv64-linux-gnu-
# perhaps in /opt/riscv/bin
#TOOLPREFIX =

# Try to infer the correct TOOLPREFIX if not set
ifndef TOOLPREFIX
TOOLPREFIX := $(shell if riscv64-unknown-elf-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
    then echo 'riscv64-unknown-elf-'; \
    elif riscv64-linux-gnu-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
    then echo 'riscv64-linux-gnu-'; \
    elif riscv64-unknown-linux-gnu-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1; \
    then echo 'riscv64-unknown-linux-gnu-'; \
    else echo "***" 1>&2; \
    echo "*** Error: Couldn't find a riscv64 version of GCC/binutils." 1>&2; \
    echo "*** To turn off this error, run 'gmake TOOLPREFIX= ...'." 1>&2; \
```

```
    echo "***" 1>&2; exit 1; fi)
endif

QEMU = qemu-system-riscv64

CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
OBJCOPY = $(TOOLPREFIX)objcopy
OBJDUMP = $(TOOLPREFIX)objdump

CFLAGS = -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2
CFLAGS += -MD
CFLAGS += -mcmodel=medany
# CFLAGS += -ffreestanding -fno-common -nostdlib -mno-relax
CFLAGS += -fno-common -nostdlib
CFLAGS += -fno-builtin-strncpy -fno-builtin-strncmp -fno-builtin-strlen -fno-builtin-memset
CFLAGS += -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero
CFLAGS += -fno-builtin-strchr -fno-builtin-exit -fno-builtin-malloc -fno-builtin-putc
CFLAGS += -fno-builtin-free
CFLAGS += -fno-builtin-memcpy -Wno-main
CFLAGS += -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf
CFLAGS += -I.
CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)

# Disable PIE when possible (for Ubuntu 16.10 toolchain)
ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]no-pie'),)
CFLAGS += -fno-pie -no-pie
endif
ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]nopie'),)
CFLAGS += -fno-pie -nopie
endif

LDFLAGS = -z max-page-size=4096

$K/kernel: $(OBJS) $K/kernel.ld $U/initcode
    $(LD) $(LDFLAGS) -T $K/kernel.ld -o $K/kernel $(OBJS)
    $(OBJDUMP) -S $K/kernel > $K/kernel.asm
    $(OBJDUMP) -t $K/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $K/kernel.sym

$U/initcode: $U/initcode.S
    $(CC) $(CFLAGS) -march=rv64g -nostdinc -I. -Ikernel -c $U/initcode.S -o $U/initcode.o
    $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o $U/initcode.out $U/initcode.o
    $(OBJCOPY) -S -O binary $U/initcode.out $U/initcode
    $(OBJDUMP) -S $U/initcode.o > $U/initcode.asm

tags: $(OBJS) _init
```

```
	etags *.S *.c

ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o

_%: %.o $(ULIB)
	$(LD) $(LDFLAGS) -T $U/user.ld -o $@ $^
	$(OBJDUMP) -S $@ > $*.asm
	$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym

$U/usys.S : $U/usys.pl
	perl $U/usys.pl > $U/usys.S

$U/usys.o : $U/usys.S
	$(CC) $(CFLAGS) -c -o $U/usys.o $U/usys.S

$U/_forktest: $U/forktest.o $(ULIB)
	# forktest has less library code linked in - needs to be small
	# in order to be able to max out the proc table.
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $U/_forktest $U/forktest.o
$U/ulib.o $U/usys.o
	$(OBJDUMP) -S $U/_forktest > $U/forktest.asm

mkfs/mkfs: mkfs/mkfs.c $K/fs.h $K/param.h
	gcc -Werror -Wall -I. -o mkfs/mkfs mkfs/mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_syscount\
	$U/_printhell\

fs.img: mkfs/mkfs README $(UPROGS)
	mkfs/mkfs fs.img README $(UPROGS)
```

```
-include kernel/*.d user/*.d

clean:
    rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
    */*.o */*.d */*.asm */*.sym \
    $U/initcode $U/initcode.out $K/kernel fs.img \
    mkfs/mkfs .gdbinit \
        $U/usys.S \
    $(UPROGS)

# try to generate a unique GDB port
GDBPORT = $(shell expr `id -u` % 5000 + 25000)
# QEMU's gdb stub command line changed in 0.11
QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \
    then echo "-gdb tcp::$(GDBPORT)"; \
    else echo "-s -p $(GDBPORT)"; fi)
ifndef CPUS
CPUS := 3
endif

QEMUOPTS = -machine virt -bios none -kernel $K/kernel -m 128M -smp $(CPUS) -
nographic
QEMUOPTS += -global virtio-mmio.force-legacy=false
QEMUOPTS += -drive file=fs.img,if=none,format=raw,id=x0
QEMUOPTS += -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

qemu: $K/kernel fs.img
    $(QEMU) $(QEMUOPTS)

.gdbinit: .gdbinit.tmpl-riscv
    sed "s/:1234/:$(GDBPORT)/" < $^ > $@

qemu-gdb: $K/kernel .gdbinit fs.img
    @echo "*** Now run 'gdb' in another window." 1>&2
    $(QEMU) $(QEMUOPTS) -S $(QEMUGDB)
```


Summary

1. **Ensure Proper Indentation**: Make sure all commands in the Makefile are indented with a tab character.
2. **Add Conditional Compilation Flags**: Use the `SCHEDULER` macro to switch between RR and MLFQ schedulers.
3. **Compile and Run**: Use the `SCHEDULER` macro to specify the desired scheduler during compilation.

This setup allows you to easily switch between different scheduling algorithms by specifying the `SCHEDULER` macro during the build process.

**kernel/proc.c: In function 'scheduler':
kernel/proc.c:659:1: error: 'noreturn' function does return [-Werror]

**

The error indicates that the `scheduler` function, which is marked as `noreturn`, is returning. This is because the `MLFQ_scheduler` and `RR_scheduler` functions are not marked as `noreturn`, and the compiler cannot guarantee that they will not return.

To fix this, you should mark both `MLFQ_scheduler` and `RR_scheduler` as `noreturn` to indicate that they will not return. This will ensure that the compiler understands that the `scheduler` function will not return.

Step 1: Mark `MLFQ_scheduler` and `RR_scheduler` as `noreturn`

Add the `__attribute__((noreturn))` attribute to both `MLFQ_scheduler` and `RR_scheduler` functions.

```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "spinlock.h"
#include "proc.h"
#include "defs.h"
#include "number_syscall.h"

struct cpu cpus[NCPU];
struct proc proc[NPROC];

void print_proc_array() {
  static char *states[] = {
    [UNUSED]    "unused",
    [USED]      "used",
    [SLEEPING]  "sleeping",
    [RUNNABLE]  "runnable",
    [RUNNING]   "running",
    [ZOMBIE]    "zombie"
  };
  for (int i = 0; i < NPROC; i++) {
    char *state = (proc[i].state >= 0 && proc[i].state < NELEM(states) &&
states[proc[i].state]) ? states[proc[i].state] : "???";
    printf("Process %d: PID=%d, State=%s, Priority=%d\n", i, proc[i].pid,
state, proc[i].priority);
  }
  printf("\n\n");
}

struct proc *initproc;
int nextpid = 1;
struct spinlock pid_lock;
extern void forkret(void);
static void freeproc(struct proc *p);
extern char trampoline[]; // trampoline.S

// helps ensure that wakeups of wait()ing
```

```
// parents are not lost. helps obey the
// memory model when using p->parent.
// must be acquired before any p->lock.
struct spinlock wait_lock;

#define NUM_QUEUES 4
#define PRIORITY_BOOST_TICKS 48

int time_slices[NUM_QUEUES] = {1, 4, 8, 16};
int global_ticks = 0;

__attribute__((noreturn)) void RR_scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    intr_on();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
        found = 1;
      }
      release(&p->lock);
    }

    if(found == 0) {
      // No runnable process found, wait for an interrupt
      intr_on();
      asm volatile("wfi");
    }
  }
}

__attribute__((noreturn)) void MLFQ_scheduler(void) {
  printf("MLFQ SCHEDULER\n");
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {
    // Enable interrupts to avoid deadlock
    intr_on();

    struct proc *highest_priority_proc = 0;
    int highest_priority = -1;
```

```
    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->priority > highest_priority) {
        highest_priority = p->priority;
        highest_priority_proc = p;
      }
      release(&p->lock);
    }

    if (highest_priority_proc) {
      p = highest_priority_proc;
      acquire(&p->lock);
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;

      if (p->state == RUNNABLE) {
        if (p->total_ticks >= p->time_slice) {
          if (p->priority < NUM_QUEUES - 1) {
            p->priority++;
          }
          p->time_slice = time_slices[p->priority];
          p->total_ticks = 0;
        }
      }
      release(&p->lock);
    } else {
      intr_on();
      asm volatile("wfi");
    }
  }
}

__attribute__((noreturn)) void scheduler(void) {
#ifdef MLFQ
  MLFQ_scheduler();
#else
  RR_scheduler();
#endif
}
```


Step 2: Compile and Run

To compile and run with the desired scheduler, use the following commands:

For Round Robin (RR) scheduler:

```
make clean
make qemu SCHEDULER=RR
```

For Multi-Level Feedback Queue (MLFQ) scheduler:

```
make clean
make qemu SCHEDULER=MLFQ
```

Summary

1. **Mark Functions as `noreturn`**: Add the `__attribute__((noreturn))` attribute to `MLFQ_scheduler` and `RR_scheduler` functions.
2. **Compile and Run**: Use the `SCHEDULER` macro to specify the desired scheduler during compilation.

This setup ensures that the compiler understands that the `scheduler` function will not return, resolving the warning and allowing the code to compile correctly.

---

always runs Round Robin even tho i run make clean ; make qemu SCHEDULER=MLFQ

The issue might be due to the way the preprocessor directives are used. The `#if` directive checks if the macro is defined and evaluates to a non-zero value. However, in your Makefile, you are defining the macro using `-DMLFQ`, which means you should use `#ifdef` instead of `#if`.

Let's correct the code in

to use `#ifdef` and ensure that the `SCHEDULER` macro is properly defined in the Makefile.

Step 1: Modify

Update the `scheduler` function to use `#ifdef` for checking the `MLFQ` macro.

```
void scheduler(void) {
  while(1){
    #ifdef MLFQ
      MLFQ_scheduler();
    #else
      RR_scheduler();
    #endif
  }
}
```

Step 2: Ensure Proper Macro Definition in Makefile

Ensure that the `SCHEDULER` macro is properly defined in the Makefile. Here is the relevant section of the Makefile:

```
# check scheduler
SCHEDULER ?= RR

ifeq ($(SCHEDULER),MLFQ)
    CFLAGS += -DMLFQ
else ifeq ($(SCHEDULER),RR)
    CFLAGS += -DRR
endif
```

Step 3: Compile and Run

To compile and run with the desired scheduler, use the following commands:

For Round Robin (RR) scheduler:

```
make clean
make qemu SCHEDULER=RR
```

For Multi-Level Feedback Queue (MLFQ) scheduler:

```
make clean
make qemu SCHEDULER=MLFQ
```

Summary

1. **Modify

**: Use `#ifdef` instead of `#if` to check if the `MLFQ` macro is defined.
2. **Ensure Proper Macro Definition in Makefile**: Ensure that the `SCHEDULER` macro is properly defined in the Makefile.
3. **Compile and Run**: Use the `SCHEDULER` macro to specify the desired scheduler during compilation.

This setup ensures that the correct scheduler is selected based on the `SCHEDULER` macro during the build process.

If mlfq is defined, then number of cpu must be 1

To ensure that the number of CPUs is set to 1 when `MLFQ` is defined, you can modify the Makefile to enforce this condition. Specifically, you can add a check to set the `CPUS` variable to 1 if `SCHEDULER` is set to `MLFQ`.

Step 1: Modify the Makefile

Add a check to set `CPUS` to 1 if `SCHEDULER` is `MLFQ`.

```
K=kernel
U=user

OBJS = \
  $K/entry.o \
  $K/start.o \
  $K/console.o \
  $K/printf.o \
  $K/uart.o \
  $K/kalloc.o \
  $K/spinlock.o \
  $K/string.o \
  $K/main.o \
  $K/vm.o \
  $K/proc.o \
  $K/swtch.o \
  $K/trampoline.o \
  $K/trap.o \
  $K/syscall.o \
  $K/sysproc.o \
  $K/bio.o \
  $K/fs.o \
  $K/log.o \
  $K/sleeplock.o \
  $K/file.o \
  $K/pipe.o \
  $K/exec.o \
  $K/sysfile.o \
  $K/kernelvec.o \
  $K/plic.o \
  $K/virtio_disk.o

# check scheduler
SCHEDULER ?= RR

ifeq ($(SCHEDULER),MLFQ)
    CFLAGS += -DMLFQ
    CPUS := 1
else ifeq ($(SCHEDULER),RR)
    CFLAGS += -DRR
endif

# riscv64-unknown-elf- or riscv64-linux-gnu-
# perhaps in /opt/riscv/bin
#TOOLPREFIX =

# Try to infer the correct TOOLPREFIX if not set
```

```
ifndef TOOLPREFIX
TOOLPREFIX := $(shell if riscv64-unknown-elf-objdump -i 2>&1 | grep 'elf64-big'
>/dev/null 2>&1; \
    then echo 'riscv64-unknown-elf-'; \
    elif riscv64-linux-gnu-objdump -i 2>&1 | grep 'elf64-big' >/dev/null 2>&1;
\
    then echo 'riscv64-linux-gnu-'; \
    elif riscv64-unknown-linux-gnu-objdump -i 2>&1 | grep 'elf64-big'
>/dev/null 2>&1; \
    then echo 'riscv64-unknown-linux-gnu-'; \
    else echo "***" 1>&2; \
    echo "*** Error: Couldn't find a riscv64 version of GCC/binutils." 1>&2; \
    echo "*** To turn off this error, run 'gmake TOOLPREFIX= ...'." 1>&2; \
    echo "***" 1>&2; exit 1; fi)
endif

QEMU = qemu-system-riscv64

CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
OBJCOPY = $(TOOLPREFIX)objcopy
OBJDUMP = $(TOOLPREFIX)objdump

CFLAGS = -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2
CFLAGS += -MD
CFLAGS += -mcmodel=medany
# CFLAGS += -ffreestanding -fno-common -nostdlib -mno-relax
CFLAGS += -fno-common -nostdlib
CFLAGS += -fno-builtin-strncpy -fno-builtin-strncmp -fno-builtin-strlen -fno-
builtin-memset
CFLAGS += -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-
builtin-bzero
CFLAGS += -fno-builtin-strchr -fno-builtin-exit -fno-builtin-malloc -fno-
builtin-putc
CFLAGS += -fno-builtin-free
CFLAGS += -fno-builtin-memcpy -Wno-main
CFLAGS += -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf
CFLAGS += -I.
CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1
&& echo -fno-stack-protector)

# Disable PIE when possible (for Ubuntu 16.10 toolchain)
ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]no-pie'),)
CFLAGS += -fno-pie -no-pie
endif
ifneq ($(shell $(CC) -dumpspecs 2>/dev/null | grep -e '[^f]nopie'),)
CFLAGS += -fno-pie -nopie
endif

LDFLAGS = -z max-page-size=4096

$K/kernel: $(OBJS) $K/kernel.ld $U/initcode
```

```
	$(LD) $(LDFLAGS) -T $K/kernel.ld -o $K/kernel $(OBJS)
	$(OBJDUMP) -S $K/kernel > $K/kernel.asm
	$(OBJDUMP) -t $K/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
$K/kernel.sym

$U/initcode: $U/initcode.S
	$(CC) $(CFLAGS) -march=rv64g -nostdinc -I. -Ikernel -c $U/initcode.S -o
$U/initcode.o
	$(LD) $(LDFLAGS) -N -e start -Ttext 0 -o $U/initcode.out $U/initcode.o
	$(OBJCOPY) -S -O binary $U/initcode.out $U/initcode
	$(OBJDUMP) -S $U/initcode.o > $U/initcode.asm

tags: $(OBJS) _init
	etags *.S *.c

ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o

_%: %.o $(ULIB)
	$(LD) $(LDFLAGS) -T $U/user.ld -o $@ $^
	$(OBJDUMP) -S $@ > $*.asm
	$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym

$U/usys.S : $U/usys.pl
	perl $U/usys.pl > $U/usys.S

$U/usys.o : $U/usys.S
	$(CC) $(CFLAGS) -c -o $U/usys.o $U/usys.S

$U/_forktest: $U/forktest.o $(ULIB)
	# forktest has less library code linked in - needs to be small
	# in order to be able to max out the proc table.
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $U/_forktest $U/forktest.o
$U/ulib.o $U/usys.o
	$(OBJDUMP) -S $U/_forktest > $U/forktest.asm

mkfs/mkfs: mkfs/mkfs.c $K/fs.h $K/param.h
	gcc -Werror -Wall -I. -o mkfs/mkfs mkfs/mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
```

```
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_syscount\
	$U/_printhell\

fs.img: mkfs/mkfs README $(UPROGS)
	mkfs/mkfs fs.img README $(UPROGS)

-include kernel/*.d user/*.d

clean:
	rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
	*/*.o */*.d */*.asm */*.sym \
	$U/initcode $U/initcode.out $K/kernel fs.img \
	mkfs/mkfs .gdbinit \
		$U/usys.S \
	$(UPROGS)

# try to generate a unique GDB port
GDBPORT = $(shell expr `id -u` % 5000 + 25000)
# QEMU's gdb stub command line changed in 0.11
QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \
	then echo "-gdb tcp::$(GDBPORT)"; \
	else echo "-s -p $(GDBPORT)"; fi)
ifndef CPUS
CPUS := 3
endif

QEMUOPTS = -machine virt -bios none -kernel $K/kernel -m 128M -smp $(CPUS) -nographic
QEMUOPTS += -global virtio-mmio.force-legacy=false
QEMUOPTS += -drive file=fs.img,if=none,format=raw,id=x0
QEMUOPTS += -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

qemu: $K/kernel fs.img
	$(QEMU) $(QEMUOPTS)

.gdbinit: .gdbinit.tmpl-riscv
	sed "s/:1234/:$(GDBPORT)/" < $^ > $@

qemu-gdb: $K/kernel .gdbinit fs.img
	@echo "*** Now run 'gdb' in another window." 1>&2
	$(QEMU) $(QEMUOPTS) -S $(QEMUGDB)
```


Step 2: Modify `scheduler` Function in

Ensure that the `scheduler` function in

properly handles the `MLFQ` scheduler.

```
void scheduler(void) {
  #ifdef MLFQ
    printf("MLFQ SCHEDULER\n");
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;) {
      // Enable interrupts to avoid deadlock
      intr_on();

      struct proc *highest_priority_proc = 0;
      int highest_priority = -1;

      for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->state == RUNNABLE && p->priority > highest_priority) {
          highest_priority = p->priority;
          highest_priority_proc = p;
        }
        release(&p->lock);
      }

      if (highest_priority_proc) {
        p = highest_priority_proc;
        acquire(&p->lock);
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;

        if (p->state == RUNNABLE) {
          if (p->total_ticks >= p->time_slice) {
            if (p->priority < NUM_QUEUES - 1) {
              p->priority++;
            }
            p->time_slice = time_slices[p->priority];
            p->total_ticks = 0;
          }
        }
        release(&p->lock);
      } else {
        intr_on();
        asm volatile("wfi");
      }
    }
  #else
    printf("RR SCHEDULER\n");
```

```
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;) {
      // Enable interrupts to avoid deadlock
      intr_on();

      int found = 0;
      for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == RUNNABLE) {
          p->state = RUNNING;
          c->proc = p;
          swtch(&c->context, &p->context);
          c->proc = 0;
          found = 1;
        }
        release(&p->lock);
      }

      if(found == 0) {
        intr_on();
        asm volatile("wfi");
      }
    }
  #endif
}
```

Step 3: Compile and Run

To compile and run with the desired scheduler, use the following commands:

For Round Robin (RR) scheduler:


```
make clean
make qemu SCHEDULER=RR
```

For Multi-Level Feedback Queue (MLFQ) scheduler:

```
make clean
make qemu SCHEDULER=MLFQ
```

Summary

1. **Modify the Makefile**: Add a check to set `CPUS` to 1 if `SCHEDULER` is `MLFQ`.
2. **Modify `scheduler` Function in

**: Ensure that the `scheduler` function properly handles the `MLFQ` scheduler.
3. **Compile and Run**: Use the `SCHEDULER` macro to specify the desired scheduler during compilation.

This setup ensures that the correct scheduler is selected based on the `SCHEDULER` macro during the build process and that the number of CPUs is set to 1 when using the `MLFQ` scheduler.

**modiify usertests.c to be able to test mlfq and round robin **

To modify

to test MLFQ (Multi-Level Feedback Queue) and Round Robin scheduling, you can add specific test functions that create processes and observe their behavior under these scheduling policies. Here's how you can do it:

1. **Add Test Functions for MLFQ and Round Robin Scheduling:**
   - Create test functions that will create multiple processes and observe their scheduling behavior.
   - Use `fork()` to create child processes and `sleep()` to simulate work.
2. **Modify the `quicktests` and `slowtests` Arrays:**
   - Add the new test functions to these arrays so they can be executed.
3. **Implement the Test Functions:**
   - Implement the logic for the test functions to create processes and observe their scheduling.

Here is the modified

:


```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"
#include "kernel/fcntl.h"
#include "kernel/syscall.h"
#include "kernel/memlayout.h"
#include "kernel/riscv.h"

#define BUFSZ  ((MAXOPBLOCKS+2)*BSIZE)

char buf[BUFSZ];

// Test function for Round Robin scheduling
void rr_test(char *s) {
  int pid;
  for (int i = 0; i < 5; i++) {
    pid = fork();
```

```
    if (pid < 0) {
      printf("fork failed\n");
      exit(1);
    }
    if (pid == 0) {
      // Child process
      for (int j = 0; j < 5; j++) {
        printf("RR Process %d running\n", getpid());
        sleep(10);
      }
      exit(0);
    }
  }
  for (int i = 0; i < 5; i++) {
    wait(0);
  }
  printf("Round Robin test completed\n");
}

// Test function for MLFQ scheduling
void mlfq_test(char *s) {
  int pid;
  for (int i = 0; i < 5; i++) {
    pid = fork();
    if (pid < 0) {
      printf("fork failed\n");
      exit(1);
    }
    if (pid == 0) {
      // Child process
      for (int j = 0; j < 5; j++) {
        printf("MLFQ Process %d running\n", getpid());
        sleep(10);
      }
      exit(0);
    }
  }
  for (int i = 0; i < 5; i++) {
    wait(0);
  }
  printf("MLFQ test completed\n");
}

struct test quicktests[] = {
  {"rr_test", rr_test},
  {"mlfq_test", mlfq_test},
  // Add other quick tests here
};

struct test slowtests[] = {
  // Add slow tests here
};
```

```
int run(void f(char *), char *s) {
  int pid;
  int xstatus;
  if((pid = fork()) == 0){
    f(s);
    exit(0);
  }
  wait(&xstatus);
  if(xstatus == 0){
    printf("%s: OK\n", s);
    return 1;
  } else {
    printf("%s: FAILED\n", s);
    return 0;
  }
}

int runtests(struct test *tests, char *justone, int continuous) {
  int i;
  for(i = 0; tests[i].s != 0; i++){
    if(justone == 0 || strcmp(tests[i].s, justone) == 0){
      if(run(tests[i].f, tests[i].s) == 0 && continuous == 0)
        return 1;
    }
  }
  return 0;
}

int main(int argc, char *argv[]) {
  int quick = 1;
  int continuous = 0;
  char *justone = 0;

  if(argc > 1 && strcmp(argv[1], "-q") == 0){
    quick = 1;
    argc--;
    argv++;
  }
  if(argc > 1 && strcmp(argv[1], "-c") == 0){
    continuous = 1;
    argc--;
    argv++;
  }
  if(argc > 1)
    justone = argv[1];

  if(quick){
    if(runtests(quicktests, justone, continuous) != 0)
      exit(1);
  } else {
    if(runtests(slowtests, justone, continuous) != 0)
      exit(1);
  }
```

```
    exit(0);
  }
```

This code adds two new test functions, `rr_test` and `mlfq_test`, which create multiple processes and simulate work by printing messages and sleeping. These functions are added to the `quicktests` array so they can be executed when running the tests.

**performance comparison between the default and 2 implemented scheduling policies by showing the average waiting and running times for processes. Set the processes to run on only 1 CPU for this purpose. Create a timeline graph that shows which queue a process is in over time. Vary the length of time that each process consumes the CPU before willingly quitting. The graph should be a timeline/scatter plot between queue_id on the y-axis and time elapsed on the x-axis from start with color-coded processes. Ensure that the priority boost is visible in this graph.

Write nessacary test programs**

To compare the performance between the default and the two implemented scheduling policies (Round Robin and MLFQ), we need to:

1. **Modify the kernel to collect statistics**: Track the waiting and running times for each process.
2. **Create test programs**: Generate processes that run for varying lengths of time and collect the necessary data.
3. **Visualize the results**: Create a timeline graph showing the queue a process is in over time.

Step 1: Modify the Kernel to Collect Statistics

Modify the `proc` structure in

to include fields for tracking waiting and running times:


```
struct proc {
  // Existing fields...
  uint64 start_time;       // Time when the process starts
  uint64 end_time;         // Time when the process ends
  uint64 total_waiting_time; // Total waiting time
  uint64 total_running_time; // Total running time
  // Existing fields...
};
```

Update the scheduler in

to track these times:

```
void
scheduler(void)
{
```

```
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    // Enable interrupts on this processor.
    intr_on();
    // Loop over process table looking for process to run.
    for(p = proc; p < &proc[NPROC]; p++){
      acquire(&p->lock);
      if(p->state == RUNNABLE){
        // Track waiting time
        p->total_waiting_time += ticks - p->end_time;
        // Switch to chosen process. It is the process's job
        // to release p->lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        p->start_time = ticks;
        c->proc = p;
        swtch(&c->context, &p->context);
        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        p->end_time = ticks;
        p->total_running_time += p->end_time - p->start_time;
      }
      release(&p->lock);
    }
  }
}
```

Step 2: Create Test Programs

Create a test program to generate processes that run for varying lengths of time:


```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void run_for_time(int time) {
  int start = uptime();
  while (uptime() - start < time) {
    // Busy wait
  }
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <num_processes>\n", argv[0]);
    exit(1);
  }
```

```c
  int num_processes = atoi(argv[1]);
  int times[] = {10, 20, 30, 40, 50}; // Varying lengths of time

  for (int i = 0; i < num_processes; i++) {
    if (fork() == 0) {
      run_for_time(times[i % 5]);
      exit(0);
    }
  }

  for (int i = 0; i < num_processes; i++) {
    wait(0);
  }

  exit(0);
}
```

Step 3: Visualize the Results

Create a Python script to generate the timeline graph:

```python
import matplotlib.pyplot as plt
import pandas as pd

# Sample data
data = {
    'time': [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
    'queue_id': [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2],
    'process_id': [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3]
}

df = pd.DataFrame(data)

# Plot
fig, ax = plt.subplots()
for key, grp in df.groupby(['process_id']):
    ax.plot(grp['time'], grp['queue_id'], label=f'Process {key}')

plt.xlabel('Time Elapsed')
plt.ylabel('Queue ID')
plt.title('Process Queue Timeline')
plt.legend()
plt.show()
```


Running the Tests

1. **Compile and run the test program**:

```
gcc -o test_program test_program.c
./test_program 10
```

2. **Collect the data**: Modify the kernel to print the collected statistics at the end of each process's execution.

3. **Generate the graph**: Use the Python script to visualize the collected data.

By following these steps, you can compare the performance of the default, Round Robin, and MLFQ scheduling policies and visualize the process queue timeline.

**Write a test program to test mlfq **

To test the Multi-Level Feedback Queue (MLFQ) scheduling policy, we need to create a test program that generates multiple processes with varying CPU burst times. This will help us observe how processes are moved between different priority queues based on their CPU usage.

Here's a test program to test MLFQ scheduling:


```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void run_for_time(int time) {
  int start = uptime();
  while (uptime() - start < time) {
    // Busy wait to simulate CPU-bound process
  }
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <num_processes>\n", argv[0]);
    exit(1);
  }

  int num_processes = atoi(argv[1]);
  int times[] = {10, 20, 30, 40, 50}; // Varying lengths of time

  for (int i = 0; i < num_processes; i++) {
    int pid = fork();
    if (pid < 0) {
      printf("fork failed\n");
      exit(1);
    }
    if (pid == 0) {
      // Child process
      run_for_time(times[i % 5]);
      printf("Process %d finished\n", getpid());
```

```
      exit(0);
    }
  }

  for (int i = 0; i < num_processes; i++) {
    wait(0);
  }

  printf("MLFQ test completed\n");
  exit(0);
}
```

Explanation

1. `run_for_time(int time)`: This function simulates a CPU-bound process by busy-waiting for the specified amount of time.
2. `main(int argc, char *argv[])`: The main function takes the number of processes to create as a command-line argument. It then forks multiple child processes, each running for a different length of time (from the `times` array).
3. **Forking and Running**: Each child process runs the `run_for_time` function for a specified duration and then exits.
4. **Waiting for Children**: The parent process waits for all child processes to finish before printing a completion message.

Running the Test

1. **Compile the test program**:

   ```
   gcc -o mlfq_test mlfq_test.c
   ```

2. **Run the test program**:

   ```
   ./mlfq_test 10
   ```


This will create 10 processes with varying CPU burst times, allowing you to observe how the MLFQ scheduler handles them. You can modify the `times` array and the number of processes to further test the scheduler's behavior under different conditions.

**Write a test program to test mlfq **

To test the Multi-Level Feedback Queue (MLFQ) scheduling policy, we need to create a test program that generates multiple processes with varying CPU burst times. This will help us observe how processes are moved between different priority queues based on their CPU usage.

Here's a test program to test MLFQ scheduling:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void run_for_time(int time) {
  int start = uptime();
  while (uptime() - start < time) {
    // Busy wait to simulate CPU-bound process
  }
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <num_processes>\n", argv[0]);
    exit(1);
  }

  int num_processes = atoi(argv[1]);
  int times[] = {10, 20, 30, 40, 50}; // Varying lengths of time

  for (int i = 0; i < num_processes; i++) {
    int pid = fork();
    if (pid < 0) {
      printf("fork failed\n");
      exit(1);
    }
    if (pid == 0) {
      // Child process
      run_for_time(times[i % 5]);
      printf("Process %d finished\n", getpid());
      exit(0);
    }
  }

  for (int i = 0; i < num_processes; i++) {
    wait(0);
  }

  printf("MLFQ test completed\n");
  exit(0);
}
```


Explanation

1. `run_for_time(int time)`: This function simulates a CPU-bound process by busy-waiting for the specified amount of time.
2. `main(int argc, char *argv[])`: The main function takes the number of processes to create as a command-line argument. It then forks multiple child processes, each running for a different length

of time (from the `times` array).

3. **Forking and Running**: Each child process runs the `run_for_time` function for a specified duration and then exits.
4. **Waiting for Children**: The parent process waits for all child processes to finish before printing a completion message.

Running the Test

1. **Compile the test program**:

```
gcc -o mlfq_test mlfq_test.c
```

2. **Run the test program**:

```
./mlfq_test 10
```

This will create 10 processes with varying CPU burst times, allowing you to observe how the MLFQ scheduler handles them. You can modify the `times` array and the number of processes to further test the scheduler's behavior under different conditions.

---

modify procdump to also provide info regarding MLFQ if the schedfuler being used is an mlfq

To modify the `procdump` function to provide information regarding the Multi-Level Feedback Queue (MLFQ) if the scheduler being used is MLFQ, you can add additional details about the priority and time slice of each process.

Here's the modified `procdump` function:


```
void procdump(void) {
  static char *states[] = {
    [UNUSED]    "unused",
    [USED]      "used",
    [SLEEPING]  "sleep ",
    [RUNNABLE]  "runble",
    [RUNNING]   "run   ",
    [ZOMBIE]    "zombie"
  };
  struct proc *p;
  char *state;

  printf("\n");
  for(p = proc; p < &proc[NPROC]; p++) {
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
```

```
      state = states[p->state];
    else
      state = "???";

    #ifdef MLFQ
      printf("%d %s %s Priority: %d Time Slice: %d\n", p->pid, state, p->name,
p->priority, p->time_slice);
    #else
      printf("%d %s %s\n", p->pid, state, p->name);
    #endif
  }
}
```

Explanation

1. **State Array**: The `states` array maps process states to their string representations.
2. **Loop Through Processes**: The function loops through all processes in the `proc` array.
3. **Check Process State**: If the process state is `UNUSED`, it continues to the next process. Otherwise, it determines the state string.
4. **MLFQ Information**: If the `MLFQ` scheduler is defined (`#ifdef MLFQ`), it prints additional information about the process's priority and time slice. Otherwise, it prints the basic process information.

Compilation and Testing

1. **Compile the Kernel**: Ensure that the kernel is compiled with the MLFQ scheduler enabled.
2. **Run the Kernel**: Boot the kernel and run the `procdump` function to see the additional MLFQ information.

This modification will help you observe the behavior of processes under the MLFQ scheduler, providing insights into their priority levels and time slices.

PROFI

when does the priiority get reduced ?

In a Multi-Level Feedback Queue (MLFQ) scheduler, the priority of a process typically gets reduced (i.e., the process is moved to a lower-priority queue) when it has consumed its allocated time slice without yielding the CPU. This mechanism ensures that CPU-bound processes do not monopolize the CPU and allows I/O-bound and interactive processes to get more CPU time.

From the provided code excerpts, it appears that the priority reduction logic is implemented in the scheduler function. Specifically, the priority is reduced when the process has consumed its entire time slice. Here is the relevant part of the scheduler function from

:

```
void scheduler(void) {
  #ifdef MLFQ
    printf("MLFQ SCHEDULER\n");
    struct proc *p;
```

```
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;) {
      // Enable interrupts to avoid deadlock
      intr_on();
      struct proc *highest_priority_proc = 0;
      int highest_priority = -1;
      for (p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if (p->state == RUNNABLE && p->priority > highest_priority) {
          highest_priority = p->priority;
          highest_priority_proc = p;
        }
        release(&p->lock);
      }
      if (highest_priority_proc) {
        p = highest_priority_proc;
        acquire(&p->lock);
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
        if (p->state == RUNNABLE) {
          if (p->total_ticks >= p->time_slice) {
            if (p->priority < NUM_QUEUES - 1) {
              p->priority++;
            }
            p->time_slice = time_slices[p->priority];
            p->total_ticks = 0;
          }
        }
        release(&p->lock);
      } else {
        intr_on();
        asm volatile("wfi");
      }
    }
  #else
    // Other scheduler implementation
  #endif
}
```


Explanation

1. **Scheduler Loop**: The scheduler continuously loops, looking for the highest-priority runnable process.
2. **Process Selection**: It selects the highest-priority runnable process.
3. **Context Switch**: It switches context to the selected process.
4. **Priority Reduction**:

  - After the process runs, if it is still in the `RUNNABLE` state and has consumed its entire time slice (`p->total_ticks >= p->time_slice`), its priority is increased (`p->priority++`), effectively moving it to a lower-priority queue.
  - The process's time slice is updated to the time slice of the new priority level (`p->time_slice = time_slices[p->priority]`).
  - The total ticks counter is reset (`p->total_ticks = 0`).

This mechanism ensures that processes that use up their time slices are gradually moved to lower-priority queues, allowing other processes to get CPU time.
