OPERATING SYSTEM AND NETWORKS
TUT - 5

PROCESS SCHEDULING

Process scheduling is a fundamental aspect of an operating system (OS) that plays a crucial role in managing the execution of multiple processes on a computer system. The primary objectives of process scheduling in an OS are as follows:

1. Resource Utilization: The OS aims to make efficient use of system resources, such as the CPU (Central Processing Unit), memory, and I/O devices. Process scheduling ensures that these resources are utilized effectively by allowing multiple processes to share them fairly.
2. Fairness: The OS needs to provide fair access to system resources for all running processes. It prevents any single process from monopolizing the CPU and ensures that each process gets its fair share of CPU time.
3. Responsiveness: The OS should respond quickly to user inputs and I/O requests. Scheduling algorithms prioritize processes that are waiting for user input or I/O operations to keep the system responsive.
4. Throughput: The OS aims to maximize the number of processes completed within a given time period. A good scheduling algorithm should ensure that processes are executed efficiently to achieve high throughput.
5. Predictability: Certain applications, like real-time systems, require predictable and guaranteed response times. Scheduling algorithms need to provide predictable behavior to meet these requirements.
6. Minimization of Waiting Time: Scheduling algorithms attempt to minimize the time processes spend waiting in the ready queue. Reducing waiting times can improve overall system performance.
7. CPU Efficiency: Scheduling algorithms strive to keep the CPU busy as much as possible. They minimize idle time on the CPU to ensure efficient resource usage.

SCHEDULING ALGORITHMS

1) First-Come, First-Served (FCFS) Scheduling
- Description: Processes are executed in the order they arrive in the ready queue.
- Advantages: Simple and easy to implement.
- Disadvantages: May result in poor response time and inefficient CPU utilization.

2) Shortest Job Next (SJN) / Shortest Job First (SJF) Scheduling
- Description: Selects the process with the smallest execution time next.
- Advantages: Minimizes waiting time for short jobs, efficient CPU usage.
- Disadvantages: Needs knowledge of job times in advance, not suitable for interactive systems.

3) Round Robin (RR) Scheduling
- Description: Each process is given a fixed time slice (quantum) on the CPU.
- Advantages: Fairness, good for time-sharing systems.
- Disadvantages: Overhead of context switching, may not be efficient for long-running jobs.

4) Priority Scheduling
- Description: Processes are assigned priority levels, and higher-priority processes are executed first.
- Advantages: Allows for different service levels for processes.
- Disadvantages: Risk of starvation (low-priority processes never run).

5) Multilevel Queue Scheduling

- **Description: Processes are grouped into multiple queues, each with its own scheduling algorithm.**
- **Advantages: Supports different classes of processes with different scheduling policies.**
- **Disadvantages: Complex to manage with multiple queues.**

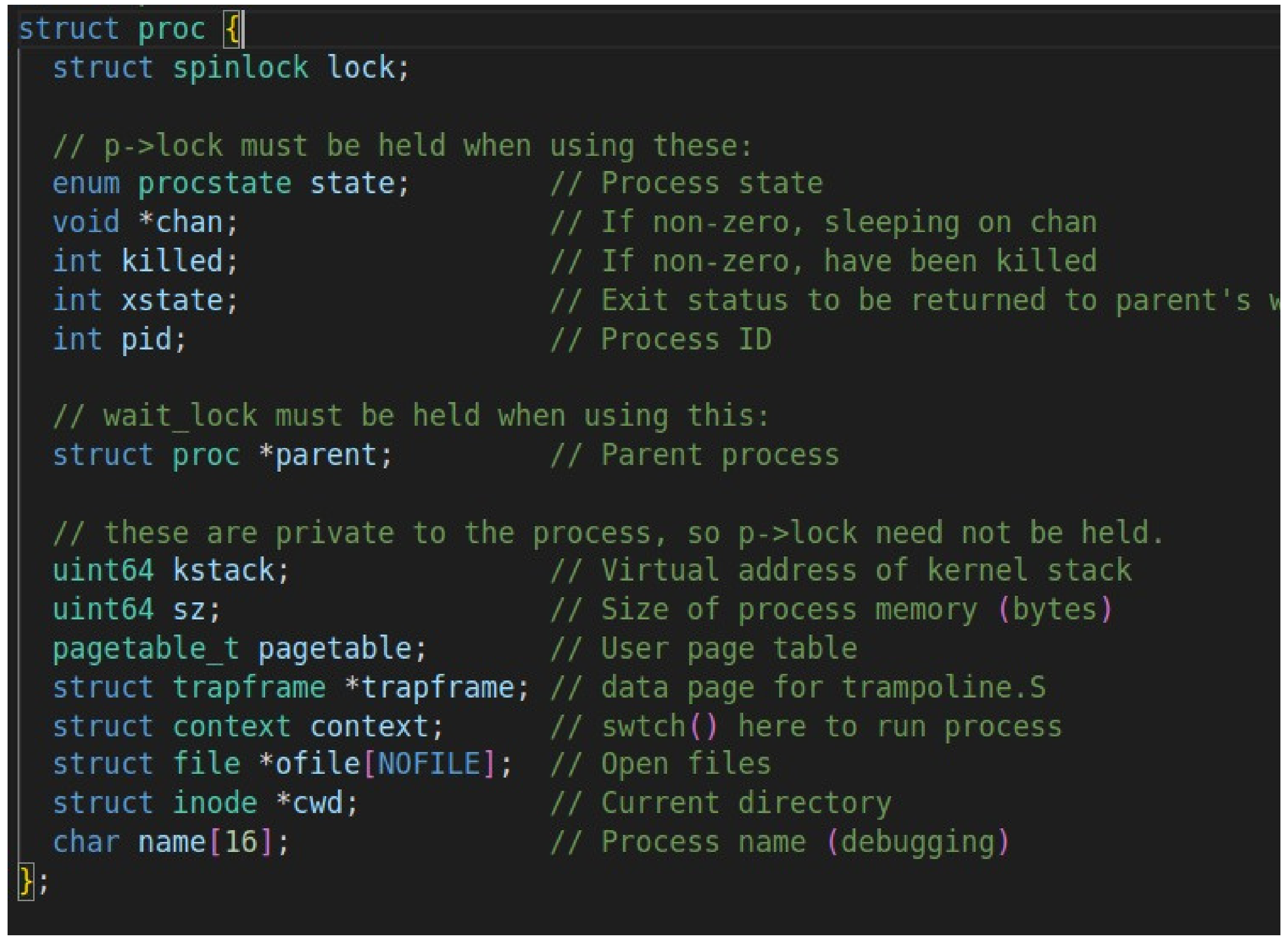
6) Multilevel Feedback Queue Scheduling

- **Description: A variation of multilevel queue scheduling where processes can move between queues based on their behavior.**
- **Advantages: Adaptive to varying workloads.**
- **Disadvantages: Complexity in implementation.**

CPU SCHEDULING IN XV6

- **In the uniprogrammming systems like MS DOS, when a process waits for any I/O operation to be done, the CPU remains idol. This is an overhead since it wastes the time and causes the problem of starvation. However, In Multiprogramming systems, the CPU doesn't remain idle during the waiting time of the Process and it starts executing other processes. Operating System has to define which process the CPU will be given.**

- **PROCESS CONTROL BLOCK (PCB) : Contains information about the process during its lifetime such as its starting time**



```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's w
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Found in kernel/proc.h .It is created for every newly created process to store information about that process which may be needed during execution and various other process.

Already contains info like process id , process name , size etc

ROUND ROBIN SCHEDULING IN XV6

- It is the default type of scheduling already implemented in xv6

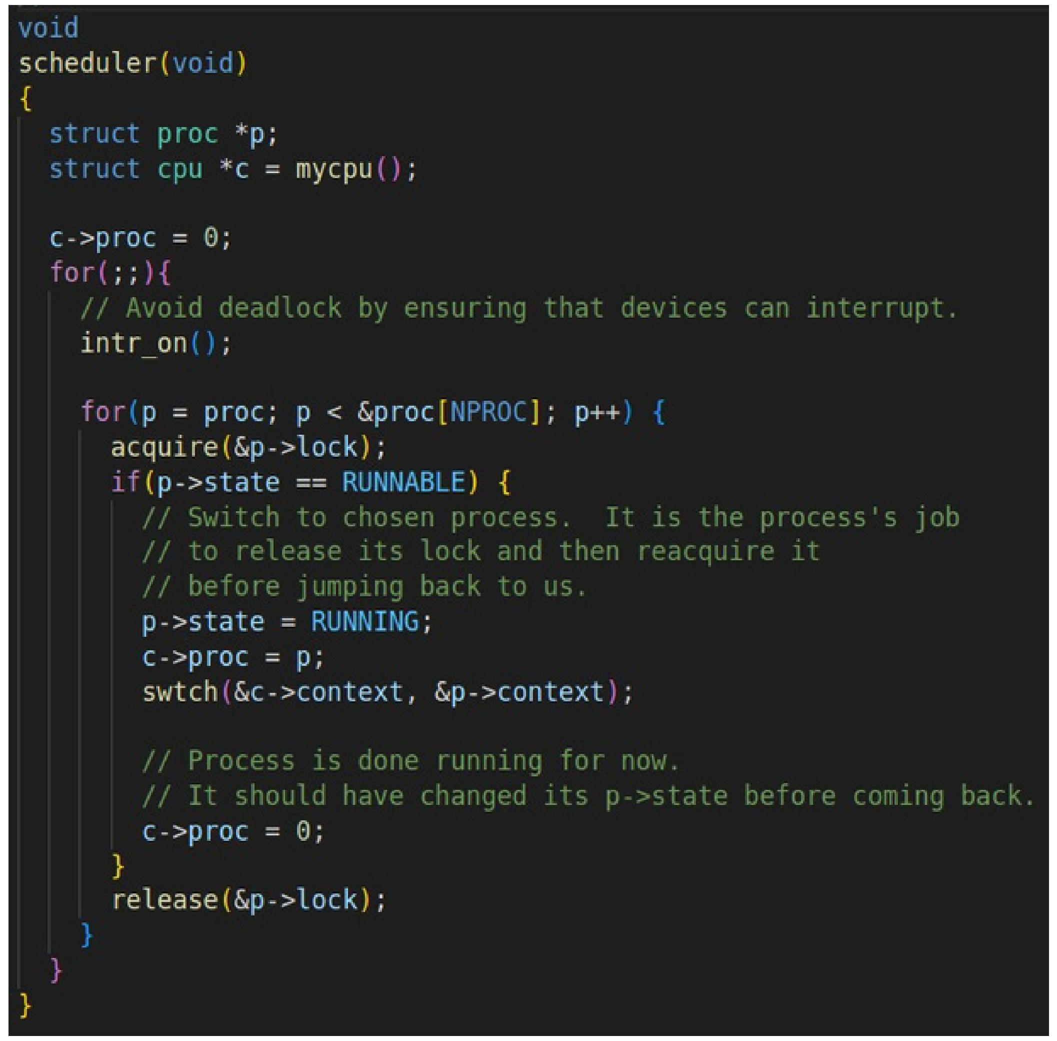
Process Creation

- Initially CPU allocates an array of size NPROC(=64) containing all unused processes
- When xv6 starts , it starts a default userprocess which has pid = 1 (Don’t mess with this)
- Whenever a new process has to be created , xv6 first calls a function “allocproc()” (kernel/proc.c)

- Scheduler function is called in main.c (by default uses a RR scheduler)

Process Scheduling

- allocproc initialises the process control block of the newly created process



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

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        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;
      }
      release(&p->lock);
    }
  }
}
```

You can see iteration over processes and context switching somewhere !!

There is an infinite for loop , so wont the processes keep doing context switching simultaneously , disallowing any process to run ??

Interrupt handing

Premptive and Nonpreemptive scheduling

- kernel/trap.c contains two kinds of functions to handle interrupts : usertrap() and kerneltrap()
- It contains a variable (find out which ?) that contains the type of interrupt which has occured
- If it is a timer interrupt , it preempts the process and deschedules it

What is the difference between Round Robin and FCFS scheduling ?? (Hint : Think in terms of interrupt handling)