MURAT MELİH EKİCİ
200111006

Linux Process Scheduling

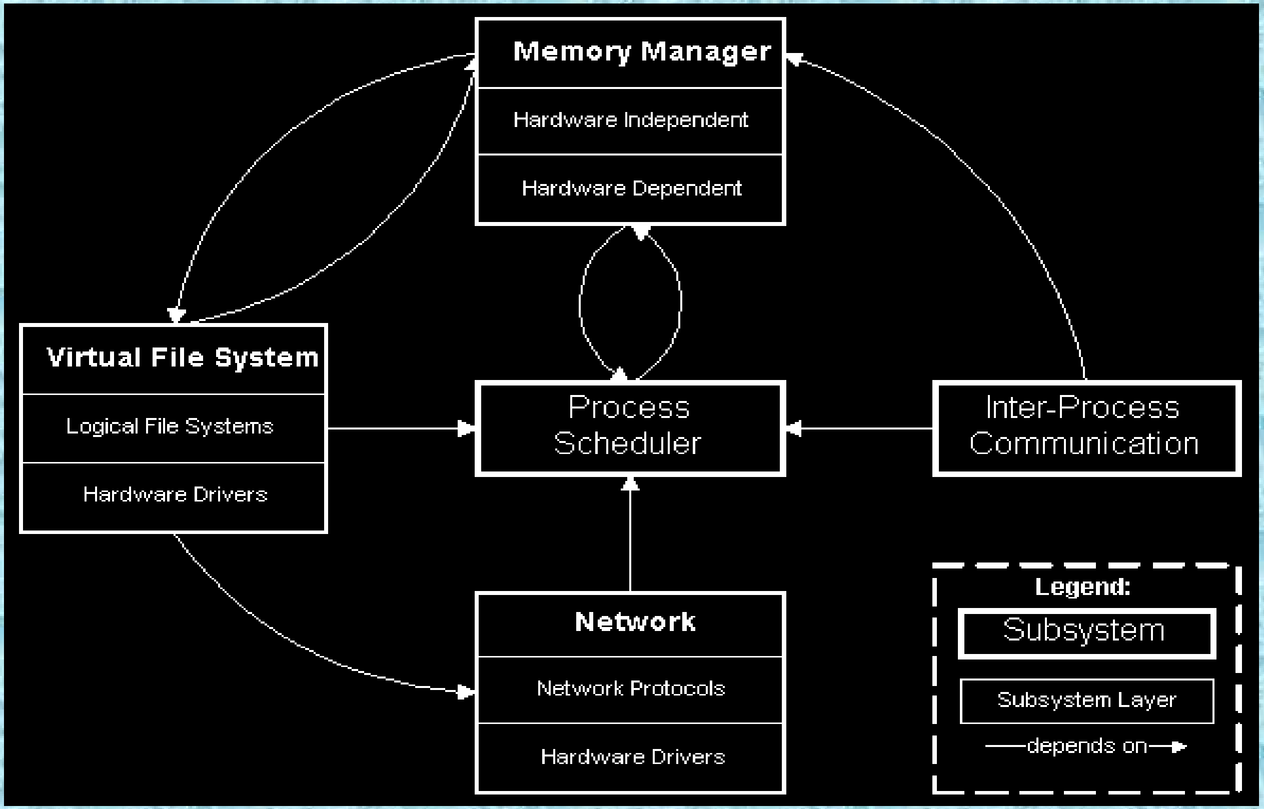
Purpose of the Kernel

- The Linux kernel presents a virtual machine interface to user **processes**. Processes are written without needing any knowledge of what physical hardware is installed on a computer . The kernel actually runs several processes concurrently, and is responsible for mediating access to hardware resources so that each process has fair access while inter-process security is maintained .

Overview of the Kernel Structure

- The Linux kernel is composed of five main subsystems:
- The **Process Scheduler** (SCHED) is responsible for controlling process access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time
- The **Memory Manager** (MM) permits multiple process to securely share the machine's main memory system. In addition, the memory manager supports virtual memory that allows Linux to support processes that use more memory than is available in the system. Unused memory is swapped out to persistent storage using the file system then swapped back in when it is needed.
- The **Virtual File System** (VFS) abstracts the details of the variety of hardware devices by presenting a common file interface to all devices. In addition, the VFS supports several file system formats that are compatible with other operating systems.
- The **Network Interface** (NET) provides access to several networking standards and a variety of network hardware.
- The **Inter-Process Communication** (IPC) subsystem supports several mechanisms for process-to-process communication on a single Linux system.

The figure *Kernel Subsystem Overview* shows a high-level decomposition of the Linux kernel, where lines are drawn from dependent subsystems to the subsystems they depend on.



This diagram emphasizes that the most central subsystem is the **process scheduler**: all other subsystems depend on the process scheduler since all subsystems need to suspend and resume processes. Usually a subsystem will suspend a process that is waiting for a hardware operation to complete, and resume the process when the operation is finished

Linux Process Scheduling

- *This is an attempt to describe the scheduling concepts used in Linux. Scheduling is a core part of every OS. Linux uses a **simple priority based scheduling algorithm** to choose between the current processes in the system. There are two types of processes in Linux, **normal** and **real time**. Real time processes will always run before normal processes and they may have either of two types of policy: **round robin** or **first in first out**. As Linux uses **Shortest-Job-First Scheduling, Priority Scheduling, Multi-level Queue Scheduling** .*

Scheduling

- It is the **scheduler** that must select the most deserving process to run out of all of the runnable processes in the system. A runnable process is one which is waiting only for a CPU to run on. Linux uses a reasonably **simple priority based scheduling algorithm** to choose between the current processes in the system. When it has chosen a new process to run it saves the state of the current process, the processor specific registers and other context being saved in the processes **task_struct** data structure .

task_struct

The most important structure for the scheduling (and may be the whole system) is the **task_struct**. This structure represents the states of all tasks running in the systems .

It is this information saved in the **task_struct** that is used by the scheduler to restore the state of the new process (this is processor specific) to run and then gives control of the system to that process. For the scheduler to fairly allocate CPU time between the runnable processes in the system it keeps information in the **task_struct** for each process:

Main variables:

- **policy**

  This is the scheduling policy that will be applied to this process. There are two types of Linux process, **normal** and **real time**. Real time processes have a higher priority than all of the other processes. If there is a real time process ready to run, it will always run first. Real time processes may have two types of policy, **round robin** and **first in first out**. In **round robin** scheduling, each runnable real time process is run in turn and in **first in, first out** scheduling each runnable process is run in the order that it is in on the run queue and that order is never changed.
- **priority**

  This is the priority that the scheduler will give to this process. It is the value used for recalculation when all runnable processes have a **counter** value of 0. You can alter the priority of a process (*use "nice" command*).
- **rt_priority**

  Linux supports real time processes and these are scheduled to have a higher priority than all of the other non-real time processes in system. This field allows the scheduler to give each real time process a relative priority. The priority of a real time processes can be altered using system calls.
- **counter**

  This is the amount of time that this process is allowed to run for. It is set to **priority** when the process is first run and is decremented each clock tick.

Start Of The Algorithm

SCHEDULING ALGORITHMS

1. **First-Come-First-Served or FIFO Scheduling**

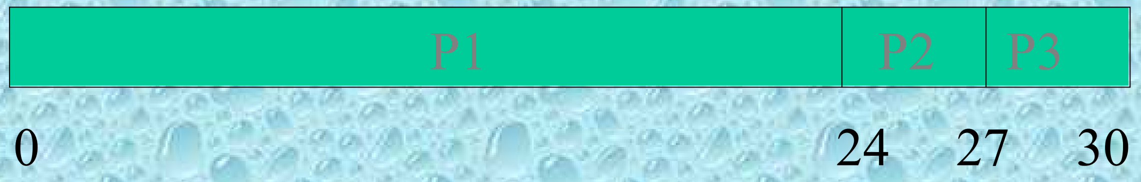
   The process that requests the CPU first is allocated the CPU first. The average waiting time for FCFS policy is often quite long.

Example:

Consider the following set of processes that arrive at time 0.

Process	**CPU Burst Time (ms)**
P1	24
P2	3
P3	3

Suppose that processes arrive in the order: P1, P2, P3, we get the result shown in the Gantt chart below:

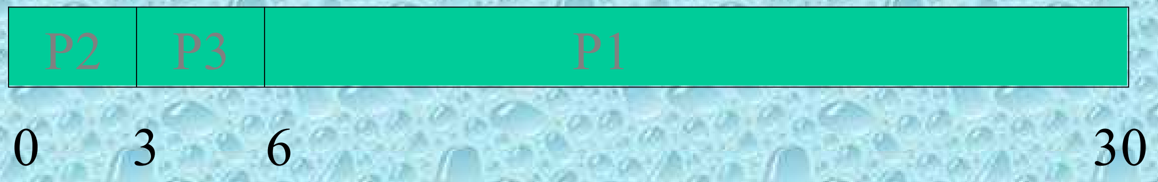


Waiting time for P1 = 0; P2 = 24; P3 = 27

Ave. waiting time: (0 + 24 + 27) /3 = 17 ms.

FCFS Scheduling, cont.

If the processes arrive in the order: P2, P3, P1, then the Gantt chart is as follows:



Waiting time for P1 = 6; P2 = 0; P3 = 3

Ave. waiting time : (6 + 0 + 3)/3 = 3

Much better than the previous case, where we had a

Convoy Effect: short process behind long process. Results in lower CPU utilization

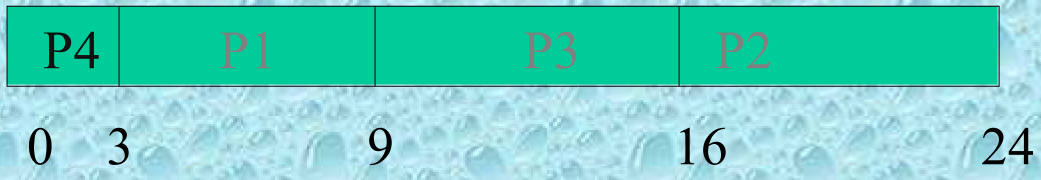
Shortest-Job-First Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

- Non-preemptive - once CPU is given to the process, it cannot be preempted until it completes its CPU burst.
- Preemptive - if a new process arrives with CPU burst length less than remaining time of of current executing process, preempt.

Example of non-preemptive SJF:

Process	CPU burst time
P1	6
P2	8
P3	7
P4	3

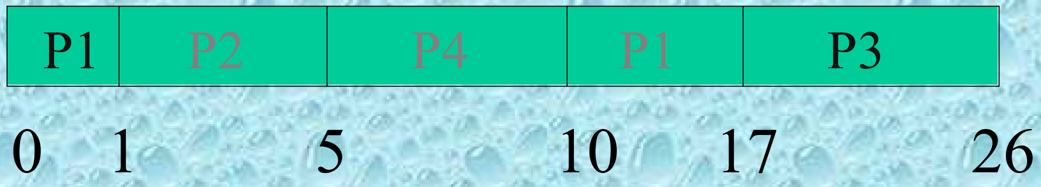


Average waiting time = (0 + 3 + 9 + 16)/4 = 7 ms

Example of Preemptive SJF

(Shortest-Remaining-Time-First)

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



SJF Scheduling, cont.

When process P2 arrives, the remaining time for P1 (7 ms) is larger than the time required for P2 (4 ms), so process P1 is preempted and P2 is scheduled.

Average waiting time is :

((10-1) + (1-1) + (17-2) + (5-3)) / 4 = 6.5 ms.

SJF Scheduling, cont.

- The SJF algorithm gives the minimum average waiting time for a given set of processes.
- The difficulty with SJF is knowing the length of the next CPU request.
- For long-term scheduling in a batch system, an estimate of the burst time can be acquired from the job description.
- For short-term scheduling, we have to predict the value of the next burst time.

Priority Scheduling

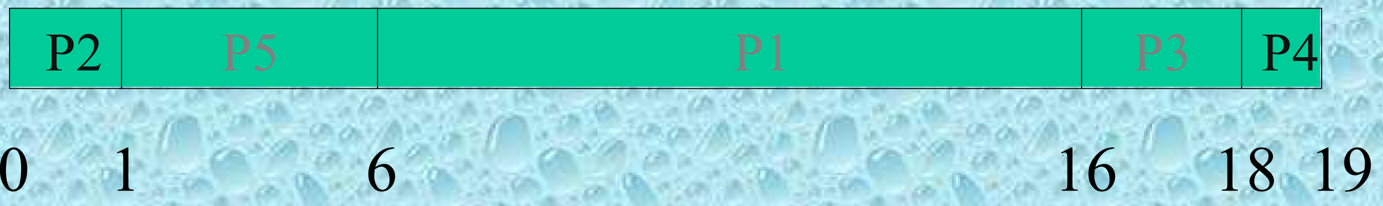
The SJF is a special case of the general *priority* scheduling algorithm.

- A priority (an integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority).
- Equal priority processes are scheduled in FCFS order.

Priority Scheduling, cont.

Example: The following processes arrive at time 0 in the order - P1, P2, P3, P4, P5.

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2



Priority Scheduling, cont.

The average waiting time is:

(0 + 1 + 6 + 16 + 18)/5 = 8.2 ms

- Priority scheduling can be either preemptive or non-preemptive.
- A major problem with priority scheduling algorithms is *indefinite blocking* or *starvation*. Low priority processes could wait indefinitely for the CPU.
- A solution to the problem of starvation is *aging*. Aging is a technique of gradually increasing the priority of processes that wait in the system a long time.

Round-Robin Scheduling

- Designed for time-sharing systems.
- Similar to FCFS, with preemption added.
- Each process gets a small unit of CPU time (a time slice), usually 10 - 100 milliseconds.
- After time slice has elapsed, the process is preempted and added to the end of the ready queue.

RR Scheduling, cont.

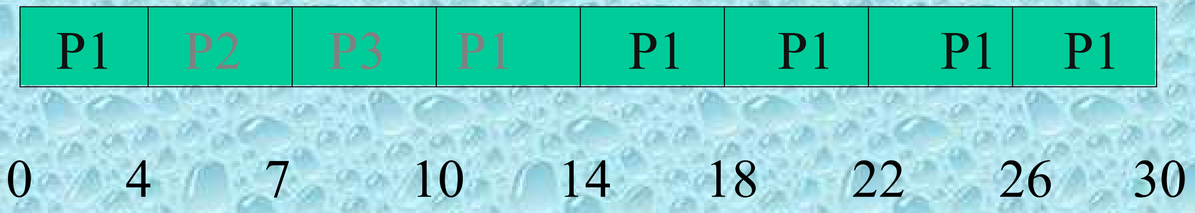
The ready queue can be implemented as a FIFO queue of processes. New processes are added to the tail of the queue. The scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum and then dispatches the process. One of two things will happen:

- The process may have a CPU burst of less than 1 time quantum, or
- CPU burst of the currently executing process is longer than one time quantum. In this case, the timer will go off, cause an interrupt, a context switch is then executed & the process put at the tail of the ready queue.

RR Scheduling, cont.

The average waiting time under the RR scheme is often quite long. Consider the following set of processes that arrive at time 0, the time quantum is set at 4 ms:

Process	CPU Burst Time
P1	24
P2	3
P3	3



RR Scheduling, cont.

The average waiting time is : 17/3 = 5.66 ms.

Performance of RR:

- If there are *n* processes in the ready queue at time quantum *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than **(*n*-1) x *q*** time units until its next time quantum.
- The performance of RR depends on the size of q.
- At one extreme, if q is very large, RR policy is the same as FCFS policy.
- If q is very small, the RR approach is called *processor sharing*. Overhead is too high.

Multi-level Queue Scheduling

- A multi-level queue-scheduling (MLQ) algorithm partitions the ready queue into several separate queues.
- Created for situations in which processes are easily classified into groups. For e.g.
  - *foreground* (interactive) processes and
  - *background* (batch) processes).
- These two types of processes have different response-time requirements, and thus, different scheduling needs.

Multi-level Queue Scheduling, cont.

- The processes are permanently assigned to one queue, based on some property of the process. (e.g. memory size, priority, or type).
- Each queue has its own scheduling algorithm. For e.g. the foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by a FCFS algorithm.

Multi-level Queue Scheduling, cont.

- There must be scheduling between the queues. Commonly implemented as fixed-priority preemptive scheduling. I.e. foreground processes have absolute priority over over the background processes, => starvation.
- Could also use a time slice algorithm where each queue gets a certain amount of CPU time which it can schedule among its processes. E.g.:
    - 80% to foreground in RR
    - 20% t o background in FCFS

End Of The Algorithm

Swap processes

- If the most deserving process to run is not the current process, then the current process must be suspended and the new one made to run. When a process is running it is using the registers and physical memory of the CPU and of the system. Each time it calls a routine it passes its arguments in registers and may stack saved values such as the address to return to in the calling routine. So, when the scheduler is running it is running in the context of the current process. It will be in a privileged mode, kernel mode, but it is still the current process that is running. When that process comes to be suspended, all of its machine state, including the program counter (PC) and all of the processor's registers, must be saved in the processes **task_struct** data structure. Then, all of the machine state for the new process must be loaded.
- This swapping of process context takes place at the end of the scheduler. The saved context for the previous process is, therefore, a snapshot of the hardware context of the system as it was for this process at the end of the scheduler. Equally, when the context of the new process is loaded, it too will be a snapshot of the way things were at the end of the scheduler, including this processes program counter and register contents.
- If the previous process or the new current process uses virtual memory then the system's page table entries may need to be updated.

Pseudo - Code

- do kernel work
  - run bottom half's
  - do soft IRQ's
- treat current process
  - if current process policy == ROUND_ROBIN: put process at the back of run queue.
  - if process id INTERRUPTIBLE and received a signal: current process state := RUNNING
  - if current process state == RUNNING: NOP
  - else remove process from run queue
- select process
  - calculate goodness
    - if process is a real time process: weight := counter + 1000
    - weight := weight + priority
  - select the process with the highest weight
  - put the current process at the end of run queue
- swap process
  - if (previous process /= next process)
    - save context of previous process
    - load context of next process

Time Accounting

- In order to be able to implement the scheduling policies described above, we must keep track of how long a process has run to be able to do a fair selectioning between the processes waiting to be processed. And if a process has used up its credit to run, we must signal this to system so another process can be chosen to run.
- In Linux this time accounting is done using bottom halfs

What are bottom halfs?

- There are often times in a kernel when you do not want to do work at this moment. A good example of this is during interrupt processing. When the interrupt was asserted, the processor stopped what it was doing and the operating system delivered the interrupt to the appropriate device driver. Device drivers should not spend too much time handling interrupts as, during this time, nothing else in the system can run. There is often some work that could just as well be done later on. Linux's bottom half handlers were invented so that device drivers and other parts of the Linux kernel could queue work.

What are bottom halfs?cont.

- Whenever a device driver, or some other part of the kernel, needs to schedule work to be done later, it adds work to the appropriate system queue, for example the timer queue, and then signals the kernel that some bottom half handling needs to be done. It does this by setting the appropriate bit in bh_active. Bit 8 is set if the driver has queued something on the immediate queue and wishes the immediate bottom half handler to run and process it. The bh_active bitmask is checked at the end of each system call, just before control is returned to the calling process. If it has any bits set, the bottom half handler routines that are active are called. Bit 0 is checked first, then 1 and so on until bit 31.
- The bit in bh_active is cleared as each bottom half handling routine is called. bh_active is transient; it only has meaning between calls to the scheduler and is a way of not calling bottom half handling routines when there is no work for them to do.

Time Accounting using bottom halfs

- Very early in the boot process when the system gets setup (paging, traps and IRQ get initialized) the scheduler too gets initialized. It's here where the infrastructure for time accounting is set up by setting a function pointer to the time accounting code which is run whenever the bottom halfs are processed, ergo every clock tick.

Summary

- SScheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- FCFS is the simplest scheduling algorithm but it can cause short processes to wait very long.
- SJF provides the shortest average waiting time. Implementing SJF is difficult, due to the difficulty in predicting the length of the next CPU burst.

REFERENCE

1.http://iamexwiwww.unibe.ch/studenten/schlpbch/linuxScheduling/LinuxScheduling.html
2.http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html