Lecture 6 (Scheduling)

The part of the OS that makes the choice which process should run next is called the **scheduler**, and the algorithm it uses is called the **scheduling algorithm**

Systems:

- **Batch systems** (just run the next job on the tape)
- Multiprogramming systems (multiple users are waiting for service and CPU time is a scarce resource, so the scheduling algorithm became more complex)
- **Personal computers** (users are usually working in one application at a time and CPUs became much faster. As a consequence, scheduling does not matter much on simple PCs)
- **Networked servers** (multiple processes often do compete for the CPU, so scheduling matters again)
- **Mobile devices** (since battery lifetime is one of the most important constraints on these devices, some schedulers try to optimize the power consumption)

User mode → kernel mode → state of current process save → new process select by scheduler → MMU reloaded to memory map of new process → new process start

Process behavior:

- **Compute-bound or CPU-bound processes** (the processes that spend most of their time computing) - long CPU burst & infrequent I/O waits
- **I/O-bound processes** (the processes that spend most of their time waiting for I/O) - short CPU burst & frequent I/O waits

Number of I/O-bound process increase as CPU develops faster than disks

When to schedule:

- **When a new process is created** (Both parent and child processes are in ready state and the scheduler can legitimately choose to run either the parent

or the child next)

- **When a process exits** (that process can no longer run, so some other process must be chosen from the set of ready processes)
- **When a process blocks on I/O or on a semaphore** (another process has to be selected to run)
- **When an I/O interrupt occurs** (a scheduling decision may be made. Scheduler will decide whether to run the process that was waiting for the interrupt, the process that was running at the time of the interrupt, or some third process)

Scheduling algorithms:

1. Nonpreemtive (scheduling algorithm picks a process to run and then just lets it run until it blocks or voluntarily releases the CPU. No scheduling decisions are made during clock interrupts)
2. Preemptive (scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run)

Environments:

1. Batch (bank, claim processing, periodic tasks; nonpreeamptive or preemptive with long period):
    - Throughput - maximize jobs per hour
    - Turnaround time - minimize time between submission and termination
    - CPU utilization - keep the CPU busy all the time
2. Interactive (servers; preemption is essential):
    - Response time - respond to requests quickly (time between issuing a command and result)
    - Proportionality - meet users' expectations (how long thing should take)
3. Real-time (programs that are intended to further the application at hand; preemption sometimes not needed):
    - Meeting deadlines - avoid losing data

- Predictability - avoid quality degradation in multimedia systems

All systems:

1. **Fairness -** giving each process a fair share of the CPU. Comparable processes should get comparable service
2. **Policy enforcement** - seeing that stated policy is carried out
3. **Balance** - keeping all parts of the system busy

Scheduling in Batch Systems:

1. **First Come First Served** (FCFS): processes are assigned the CPU in the order they request it - nonpreemptive
2. **Shortest Job First** (SJF): the scheduler picks the shortest job first, so the average turnaround time for each job would be minimal - nonpreemptive
3. **Shortest Remaining Time Next** (SRTN): the scheduler always chooses the process whose remaining run time is the shortest - preemptive version of SJF

Scheduling in Interactive Systems:

1. **Round-Robin Scheduling**: if the process is still running at the end of the **quantum** (time interval assigned to process), the CPU is preempted and given to another process. Quantum too short ⇒ too many process switches & low CPU efficiency. Quantum to big ⇒ poor response to short request.
2. **Priority Scheduling:** each process is assigned a priority, and the runnable process with the highest priority is allowed to run. Priorities can be assigned statically & dynamically.
3. **Multiple Queues:** processes in the highest class run for one quantum; processes in the next-highest class run for two quanta and etc
4. **Shortest Process Next:** analogue of the Shortest Job First algorithm in batch processing systems
5. **Guaranteed Scheduling:** make real promises to the users about performance and then live up to those promises. (n users input ⇒ 1/n CPU per person)

6. **Lottery Scheduling:** the basic idea is to give processes lottery tickets for various system resources, such as CPU time
7. **Fair-Share Scheduling:** scheduling algorithm which takes into consideration the owners of the processes and the number of processes each user owns

Context(process) switch is the process of storing the state of a process or thread.

Aging - technique to estimate next value in a series by taking the weighted average of current measured value and the previous estimate

Scheduling in Real-time systems:

Real-time system - time playing essential role. Real-time systems: hard real-time(all deadlines met) & soft-real time(some deadlines can be missed). Events in real-time systems: periodic(occur at regular intervals) & aperiodic(occur unpredictably)

Real-time system that meets this criterion is said to be schedulable (can be actually implemented):

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

$$C_i - CPU\ time,\ P_i - period$$

Real-time scheduling: static(scheduling decisions before system starts running) & dynamic (decisions at runtime)

Scheduling policy - rules defined by user for scheduler.

Policy-mechanism separation: It states that mechanisms (those parts of a system implementation that control the authorization of operations and the allocation of resources) should not dictate (or overly restrict) the policies according to which decisions are made about which operations to authorize, and which resources to allocate.

Thread scheduling

User-level threads:

Since the kernel is not aware of the existence of threads, it picks a process and gives it control for its quantum. The thread scheduler inside the process decides which thread to run. One thread might consume all of the process's time until it is finished, however, it will not affect the other processes

Kernel-level threads:

The kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to. The thread is given a quantum and is forcibly suspended if it exceeds the quantum.

Major differences:

1. **Performance**. Doing a thread switch with user-level threads takes a handful of machine instructions. With kernel-level threads it requires a full context switch
2. **Thread scheduling**. User-level threads can employ an application-specific thread scheduler that can tune an application better than the kernel can

Situation, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation.**

Nonsolution of **Dining philosophers problem**:

```
#define N 5               //number of philosophers
void philosopher(int i){
  while (TRUE){
    think();
    take_fork(i);               //take left fork
    take_fork((i+1)% N);        //take right fork
    eat();
    put_fork(i);                //put left fork
    put_fork((i+1)% N);         //put right work
  }
}
```

Solution of Dining philosophers problem:

```
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
```

```
#define HUNGRY 1
#define EATING 2

int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher{
  while (TRUE){
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}

void take_forks(int i){
  down(&mutex);          //enter critical region
  state[i] = HUNGRY;
  test(i);               //try to take forks
  up(&mutex);            //exit critical region
  down(&s[i]);           //block if forks not acquired
}

void put_forks(int i){
  down(&mutex);          //enter critical region
  state[i] = THINKING;
  test(LEFT);            //check can left neighbor eat
  test(RIGHT);
  up(&mutex);            //exit critical region
}

void test(int i){
  if (state[i] == HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING){
    state[i] = EATING;
    up(&s[i]);
  }
}
```