


**CS134: October 19, 2001**

- Process control and scheduling
  - Process Control Block
  - Resource Control Block
- Scheduling



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>      October 19, 2001

---

---

---

---

---


---

---

---

**Autonomous vs. shared scheduling**

- The scheduler may be a primitive (a function call), so it is shared by all processes
- The scheduler may be central; conceptually it is a separate process that either *polls* the system for work, or driven by *wakeup* signals
- One common model: a system call transfers control to a system routine, which performs the requested work, moves the current process to the ready-queue, and transfers control to the Scheduler
- Multiprocessing:
  - In asymmetric multiprocessing, one processor is devoted to system tasks, including scheduling
  - In symmetric multiprocessing, system work is divided among all the processors



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>      October 19, 2001

---

---

---

---

---


---

---

---

**Scheduling specification**

- $S_r$  is the set of running processes
- $S_{rd}$  is the set of ready and running processes
- Scheduler: choose  $S_a$  the set of processes that *should* be running
  - $|S_a| = \text{number of processors}$
  - For all  $p \in S_a$  and  $q \in S_{ra}$ , where  $p \neq q$ 
    - $p \rightarrow \text{priority} \geq q \rightarrow \text{priority}$
    - $p \rightarrow \text{priority} = q \rightarrow \text{priority}$  and  $p$  precedes  $q$  in the ready-queue



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>      October 19, 2001

---

---

---

---

---


---

---

---

### Dispatcher

- *Switch(p, q)*: switch from process *p* to process *q*
- *Interrupt(c)*: interrupt processor *c*
- For *Switch(p, q)*:
  - *Interrupt p->processor to get process context*
  - *Adjust the program counter if necessary*
  - *Load q->CPU\_state into the processor*



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

October 19, 2001

---

---

---

---

---

---

---


---

### Switch(p, q)

```

/* save the context for process p */
int c = p->processor;
if(p != current)
    Interrupt(c);
CopyState(c, p->CPU_state);
if(p == current)
    AdjustPC(p->CPU_state);

/* load the context for process q (may enter user mode) */
LoadState(c, q->CPU_state);
    
```



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

October 19, 2001

---

---

---

---

---


---

---

---

### Scheduler

- Construct a process/processor assignment recursively
- Starting from the highest priority
  - *Find the next highest priority process*
  - *Assign it to a CPU*
  - *If the CPU already had a lower priority process running, preempt that process*
- Be careful: don't replace the *current* process until the very last step (save it in the *deferred* variable)
- Processor affinity is not remembered



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

October 19, 2001

---

---

---

---

---

---

---

---

### Scheduler code

```


void Scheduler() {
    int cur_prio = n; // Highest priority
    Process deferred = NULL;
    while(true) {
        Process p = FindHighest(cur_prio);
        if(p == NULL)
            Finish(deferred); // Next schedulable process
        bool found = false; // Assign a CPU
        Process q_min = p;
        AllocateCPU(p, found, q_min, deferred);
        if(!found && q_min->priority < p->priority)
            Preempt(q_min, p, deferred); // Preemption
    }
}
    
```

Used for scheduling the process on the current CPU

Next schedulable process

Assign a CPU

Preemption



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>  
 October 19, 2001

---

---

---

---

---

---

---

---

---


---

### FindHighest(int cur\_prio)

- Look through the ready queues with priorities *cur\_prio* or less to find a process in the *Ready* state

```

Process FindHighest(int &cur_prio) {
    Process p = NULL;
    bool found = false;
    while(p == NULL && cur_prio > 0) {
        Process cur = ready[cur_prio].first;
        while(found && cur != NULL)
            if(cur->proc->status == Ready) {
                p = cur->proc;
                found = true;
            }
            else
                cur = cur->next;
        cur_prio--;
    }
}
    
```



Computing systems  
<http://www.cs.caltech.edu/cs134/cs134a>  
 October 19, 2001

---

---

---

---

---

---

---


---

---

---

### AllocateCPU(Process p, ...)

- Look for an idle CPU to assign to process *p*
- If an idle CPU is found
  - Assign process *p* to that processor
- Otherwise
  - Remember the running process with lowest priority, less than *p*.priority, in the variable *q\_min*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>  
 October 19, 2001

---

---

---

---

---

---

---

---

---


---

### Allocate CPU

```

void AllocateCPU(Process p, bool &found, Process &q_min, Process &deferred) {
  for(int cpu = 1; !found && cpu < NUM_PROC; cpu++) {
    if(process[cpu] == NULL) {
      found = true;
      process[cpu] = p;
      p->processor = cpu;
      p->status = Running;
      if(current->processor != cpu)
        deferred = p;
      else
        LoadState(cpu, p->CPU_state);
    }
    else if(process[cpu]->priority < q_min->priority)
      q_min = process[cpu];
  }
}
    
```

Idle processor  
Delay initialization for current CPU  
Start others right away  
Preemption



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>  
October 19, 2001

---

---

---

---

---

---

---

---

---

---


### Preempt(q, p, deferred)

- Preempt process *q* and replace it with process *p*
- Be careful: if *q* is the current process, defer startup

```

void Preempt(Process q_min, Process p, Process &deferred)
{
  q_min.status = Ready;
  p.status = Running;
  p.processor = q_min.processor;
  process[q_min.processor] = p;
  if(q_min == current)
    deferred = p;
  else
    Switch(q_min, p);
}
    
```

Assign process p to q.processor  
Preemption of *current* is deferred  
Otherwise, start process p



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>  
October 19, 2001

---

---

---

---

---

---

---

---

---


---

### Finish(deferred)

- For the final step, re-initialize the current processor

```

void Finish(Process deferred)
{
  if(current->status != Running)
    Switch(current, deferred);
  exit Scheduler;
}
    
```



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>  
October 19, 2001

---

---

---

---

---

---

---

---

---

---

### Universal scheduler (Ruschitzka & Fabry 1977)

- Priorities are a simplified management scheme
- May want multiple *policies*
  - *Batch, interactive, real-time, ...*
- A *universal scheduler* has the following parts:
  - A *decision mode*
  - A *priority function*
  - An *arbitration rule*
- At times defined by the *decision mode*, the scheduler evaluates the *priority function* on each process to determine a *current priority*. the processes with the highest priorities are assigned to the CPUs; the *arbitration rule* is applied to processes with the same priority



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---

---

---

---

### The decision mode

- The decision mode specifies when to invoke the scheduler
- *Non-preemptive* processes run as long as logically possible
- *Preemptive* processes can be stopped
  - *when a new process arrives*
  - *when an existing process is awakened*
  - *or periodically based on a time quantum*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---

---

---

---

### The priority function

- The priority function is arbitrary, but is usually based on the following process parameters:
  - *Memory requirements*
  - *Current CPU time*
  - *Current real time in system*
  - *External priorities (nice value)*
  - *Timeliness (e.g. interactive jobs need better response time; real-time jobs may have a deadline)*
  - *System load*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---

---

---

---

**The arbitration rule**

- Resolves conflicts for processes with the same priority
- Common policies
  - Random
  - Cyclic (*round-robin*)
  - Chronological (*FIFO*)



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---

---

---

---

**Time-based scheduling algorithms**

- Priority function  $P(a, r, t)$ 
  - $a$ : *attained service time*
  - $r$ : *real time*
  - $t$ : *total service time (predicted)*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---

---

---

---

**Time-based scheduling algorithms**

- FIFO: earliest arrival time first ( $P(a, r, t) = r$ )
- LIFO: latest arrival time first ( $P(a, r, t) = -r$ )
- SJN: shortest job next ( $P(a, r, t) = -t$ )
- SRT: shortest remaining time ( $P(a, r, t) = a - t$ )
- Round robin:
  - $P(a, r, t) = 0$  for all processes
  - Arbitration rule chooses oldest suspended process first



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---


---

---

---

### Time-based scheduling algorithms II

- Multilevel feedback
  - *n* different priorities
  - process is allowed to run for time  $T_i$  at level *i*, when it is bumped to level *i*-1
- Policy-based
  - Priority  $P(a, r, t) = r - f^1(a)$  for some policy function *f*
  - *f* tries to balance current process time with desired process time



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---


---

---

---

### Time-based scheduling algorithms III

Algorithm	Priority	Decision mode	Arbitration Rule
FIFO	<i>r</i>	Nonpreemptive	Random
LIFO	- <i>r</i>	Nonpreemptive	Random
SJN	- <i>t</i>	Nonpreemptive	Chronological or random
SRT	<i>a-t</i>	Preemptive (at arrival)	Chronological or random
RR	0	Preemptive (at quantum)	Cyclic
MLF	-log( <i>a</i> )	Preemptive (at quantum)	Cyclic or chronological
PD	$r \cdot f^1(a)$	Preemptive (at quantum)	Random



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---


---

---

---

### Comparison of scheduling algorithms

- FIFO, LIFO, SJN, SRT are usually used for batch processing systems
  - *SJN and SRT have better turnaround times*
- Time-shared systems need an acceptable response time, requiring a preemptive policy (either RR, MLF, or PD)
  - *RR is easiest to implement, and is used most often*
  - *Unix uses a MLF where priority increases while blocked*
  - *Windows assigns a longer quantum to the "current" application*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

October 19, 2001

---

---

---

---

---

---

---

---