LECTURE 1.3
CPU SCHEDULING ALGORITHMS I

COP4600

Dr. Matthew Gerber

2/1/2016

First-Come First-Served (6.3.1)

- The simplest of all reasonable schedulers
- Processes are allocated the CPU in the order in which they arrive
- Processes run until completion and termination
- Cannot be sensibly made preemptive
- Strengths
  - Really simple
  - Easy to write, easy to understand
- Weaknesses
  - Everything else
  - Completely unpredictable wait times
  - Non-preemptive nature makes it unsuitable for use in modern OSs

First-Come First-Served (6.3.1)

```
define pcb {
  int               process_id
  enumeration       state
  register          program_counter
  list<register>    registers
  scb               schedule
  mcb               memory
  acb               account
  iocb              io
}

define scb {
  time              arrival
}

queue<pcb> ready_queue;
```

```
function proc_ready(pcb &p) {
  p.state = READY
  p.schedule.arrival = now()
  ready_queue.enqueue(p)
}

function select_proc() {
  pcb p

  p = ready_queue.dequeue()
  dispatch(p)
}
```

First-Come First-Served (6.3.1)

```
define pcb {
  int               process_id
  enumeration       state
  register          program_counter
  list<register>    registers
  scb               schedule
  mcb               memory
  acb               account
  iocb              io
}

define scb {
  time              arrival
}

list<pcb> ready_queue;
```

```
function proc_ready(pcb &p) {
  p.state = READY
  p.schedule.arrival = now()
  ready_queue.add(p)
}

function select_proc() {
  pcb p

  p = find_minimum<pcb>(
    each rqp in ready_queue,
    rqp.schedule.arrival
  )
  ready_queue.delete(p)

  dispatch(p)
}
```

First-Come First-Served (6.3.1)

Process	Burst Time	Wait Time
P1	24	0
P2	3	24
P3	3	27

First-Come First-Served (6.3.1)

Process	Burst Time	Wait Time
P1	3	0
P2	3	3
P3	24	6

Shortest Job First (6.3.2)

- Processes are scheduled to the CPU giving priority to the process that will have the next-shortest burst
- Can be left cooperative or made preemptive
- Strengths
  - Guaranteed to produce an optimal schedule in terms of waiting time
- Weaknesses
  - Not actually possible
  - Longer processes may never get run (we'll look at this more in a bit…)

Shortest Job First (6.3.2)

```
define pcb {
  int               process_id
  enumeration       state
  register          program_counter
  list<register>    registers
  scb               schedule
  mcb               memory
  acb               account
  iocb              io
}

define scb {
  time              est_length
}

list<pcb> ready_queue;
```

```
function proc_ready(pcb &p) {
  p.state = READY
  time t = estimate_time(p)
  p.schedule.est_length = t
  ready_queue.add(p)
}

function select_proc() {
  pcb p

  p = find_minimum<pcb>(
    each rqp in ready_queue,
    rqp.schedule.est_length
  )
  ready_queue.delete(p)

  dispatch(p)
}
```

Shortest Job First (6.3.2) (Non-preemptive)

Process	Burst Time	Wait Time
P1	6	3
P2	12	18
P3	9	9
P4	3	0

P4	**P1**	**P3**	**P2**

0 3 9 18 30

Shortest Job First (6.3.2) (Preemptive)

Process	Burst Time	Arrival	Wait Time
P1	6	0	3
P2	12	1	17
P3	9	2	7
P4	3	3	0

P1	**P4**	**P1**	**P3**	**P2**

0 3 6 9 18 30

Priority (6.3.3)

- More general version of Shortest Job First
  - (Actually, it's the other way around…)
- Each process has a *priority* that determines how soon it is scheduled compared to other processes
- This priority can either be a number or determined by some sort of function
- Again, can be left cooperative or made preemptive
- Strengths
  - Flexible
- Weaknesses
  - Completely dependent on the priority being set reasonably
  - What happens if some process has a priority bad enough that some other process always has better?
    - This effect is called *starvation*

Priority (6.3.3)

```
define pcb {
  int             process_id
  enumeration     state
  register        program_counter
  list<register>  registers
  scb             schedule
  mcb             memory
  acb             account
  iocb            io
}

define scb {
  int             priority
}

list<pcb> ready_queue;
```

```
function proc_ready(pcb &p) {
  p.state = READY
  ready_queue.add(p)
}

function select_proc() {
  pcb p

  p = find_minimum<pcb>(
    each rqp in ready_queue,
    rqp.schedule.priority
  )
  ready_queue.delete(p)

  dispatch(p)
}
```

Priority (6.3.3) (Non-preemptive)

Process	Burst Time	Priority	Wait Time
P1	6	2	12
P2	12	1	0
P3	9	4	21
P4	3	3	18

P2	**P1**	**P4**	**P3**

0 12 18 21 30

Round-Robin (6.3.4)

- Algorithm explicitly designed for time-sharing systems
- Some similarities to First Come First Served
- Redesigned fundamentally to add preemption
- Define a *time quantum* or *time slice* (generally between 10ms and 100ms)
- Set the timer interrupt to stop the running process if its burst goes longer than the time quantum
- When a process becomes ready for whatever reason, put it on the back of the queue

Round-Robin (6.3.4)

```
define pcb {
  int               process_id
  enumeration       state
  register          program_counter
  list<register>    registers
  scb               schedule
  mcb               memory
  acb               account
  iocb              io
}

define scb {
  time              arrival
}

queue<pcb> ready_queue;
```

```
function proc_ready(pcb &p) {
  p.state = READY
  p.schedule.arrival = now()
  ready_queue.enqueue(p)
}

function timer_int(pcb &current) {
  proc_ready(current)
  select_proc()
}

function select_proc() {
  pcb p

  p = ready_queue.dequeue()
  dispatch(p)
}
```

Round Robin (6.3.4) (q = 2)

Process	Burst Time	Wait Time
P1	13	7
P2	4	7
P3	2	4
P4	1	6

P1	P2	P3	P4	P1	P2	P1	P1	P1	P1	P1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Round Robin (6.3.4)

- The basis of the scheduling algorithm for most interactive time-sharing systems
  - The *basis*, not what they actually use
- Not too great on overall wait time, but…
- Each process is guaranteed to have to wait at most $(n\text{-}1)q$ time units until it gets the processor back
- q needs to be large with respect to the context switch time
  - But that said, in the modern context, context switching *actually* takes a few *microseconds*
- q needs to be small enough for the system to be continually responsive
  - Round Robin degenerates to First Come First Served with a large enough q

NEXT TIME: SCHEDULING ALGORITHMS II