Process Scheduling & Synchronization intro

CS 241

February 29, 2012

Copyright © University of Illinois CS 241 Staff
Announcements

Mid-semestar feedback survey (linked off web page)

MP4 due Friday (not Tuesday)

Midterm
  • Next Tuesday, 7-9 p.m.
  • Study guide released this Wednesday
  • Next Monday’s lecture: review session
Today

Interactive scheduling
- Round robin
- Priority scheduling
- How long is a quantum?

Synchronization intro
Process scheduling

Deciding which process/thread should occupy each resource (CPU, disk, etc.) at each moment

Scheduling is everywhere...

- disk reads
- process/thread resource allocation
- servicing clients in a web server
- compute jobs in clusters / data centers
- jobs using physical machines in factories
Scheduling algorithms

Batch systems

• Usually non-preemptive: running process keeps CPU until it voluntarily gives it up
  ▪ Process exits
  ▪ Switches to blocked state
• First come first serve (FCFS)
• Shortest job first (SJF) (also preemptive version)

Interactive systems

• Running process is forced to give up CPU after time quantum expires
  ▪ Via interrupts or signals (we’ll see these later)
• Round robin
• Priority

These are some of the important ones to know, not a comprehensive list!
Thus far: Batch scheduling

FCFS, SJF, SRPT useful when fast response not necessary
  • weather simulation
  • processing click logs to match advertisements with users
  • ...

What if we need to respond to events quickly?
  • human interacting with computer
  • packets arriving every few milliseconds
  • ...

Interactive Scheduling

Usually preemptive
- Time is sliced into quanta, i.e., time intervals
- Scheduling decisions are made at the beginning of each quantum

Performance metrics
- Average response time
- Fairness (or proportional resource allocation)

Representative algorithms
- Round-robin
- Priority scheduling
Round-robin

One of the oldest, simplest, most commonly used scheduling algorithms

Select process/thread from ready queue in a round-robin fashion (i.e., take turns)

Problems
  • Might want some jobs to have greater share
  • Context switch overhead
Round-robin: Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose time quantum is 1 unit and P1, P2 & P3 never block

P1 waiting time: 1 unit
P2 waiting time: 1 unit
P3 waiting time: 1 unit

The average waiting time (AWT):
**Round-robin: Example**

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Order</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose time quantum is 1 unit and P1, P2 & P3 never block

P1 waiting time: 4  
P2 waiting time: 6  
P3 waiting time: 6

The average waiting time (AWT):  
\[(4+6+6)/3 = 5.33\]
Round-robin: Summary

Advantages
  • Jobs get fair share of CPU
  • Shortest jobs finish relatively quickly

Disadvantages
  • Larger than optimal average waiting time
    ▪ Example: 10 jobs each requiring 10 time slices
    ▪ RR: All complete after about 100 time slices
    ▪ FCFS performs about 2x better!
  • Performance depends on length of time quantum
Priority Scheduling

Rationale: higher priority jobs are more mission-critical
  • Example: DVD movie player vs. send email

Each job is assigned a priority

Select highest priority runnable job
  • FCFS or Round Robin to break ties

Problems
  • May not give the best AWT
  • Starvation of lower priority processes
**Priority Scheduling: Example**

(Lower priority number is preferable)

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Priority</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

P1 waiting time: 
P2 waiting time: 
P3 waiting time: 
P4 waiting time: 

The average waiting time (AWT):
Priority Scheduling: Example

(Lower priority number is preferable)

<table>
<thead>
<tr>
<th>Process</th>
<th>Duration</th>
<th>Priority</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

P1 waiting time: 18
P2 waiting time: 0
P3 waiting time: 11
P4 waiting time: 8

The average waiting time (AWT):
\[
\frac{0 + 8 + 11 + 18}{4} = 9.25
\]
(worse than SJF’s 7)
Setting priorities: nice

nice [OPTION] [COMMAND [ARG]...]
  • Run COMMAND with an adjusted niceness
  • With no COMMAND, print the current niceness.
  • Nicenesses range from -20 (most favorable scheduling) to 19 (least favorable).

Options
  • -n, --adjustment=N
    ▪ add integer N to the niceness (default 10)
  • --help
    ▪ display this help and exit
  • --version
    ▪ output version information and exit
Setting priorities in C

```c
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

Access scheduling priority of process, process group, or user

Returns:
- `setpriority()` returns 0 if there is no error, or -1 if there is
- `getpriority()` can return the value -1, so it is necessary to clear `errno` prior to the call, then check it afterwards to determine if a -1 is an error or a legitimate value

Parameters:
- `which`:
  - PRIO_PROCESS, PRIO_PGRP, or PRIO_USER
- `who`:
  - A process identifier for PRIO_PROCESS, a process group identifier for PRIO_PGRP, or a user ID for PRIO_USER
Choosing the time quantum

How should we choose the time quantum?

Time quantum too large
  - FIFO behavior
  - Poor response time

Time quantum too small
  - Too many context switches (overhead)
  - Inefficient CPU utilization
Choosing the time quantum

Objective 1: Fast response time
Best case: quantum = 0, response time = C

Objective 2: Efficiency
Best case: quantum = infinity, Job completion time = J

General strategy: set quantum somewhere in the middle
Choosing the time quantum

Choice depends on

- Priorities, architecture, etc.

Typical quantum: 10-100 ms

- Large enough that overhead is small percentage
- Small enough to give illusion of concurrency
- e.g., linux.ews.illinois.edu: 99.98 ms quantum using round-robin

Questions

- Does 100 ms matter? (how long is this in practical terms?)
- Does this mean all processes wait 100 ms to run?
typedef struct printer_arg_t {
  int thread_index;
} printer_arg_t;

#define BUF_SIZE 100

void * printer_thread( void *ptr )
{
  /* Create the message we will print out */
  printer_arg_t* arg = (printer_arg_t*) ptr;
  char message[BUF_SIZE];
  int i;
  for (i = 0; i < BUF_SIZE; i++)
    message[i] = ' ';
  sprintf(message + 10 * arg->thread_index, "thread %d\n", arg->thread_index);

  /* Print it forever */
  while (1)
    printf("%s", message);
}
## Experiment: results on linux.ews

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 1</th>
<th>thread 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 1</td>
<td>thread 0</td>
<td>thread 1</td>
</tr>
<tr>
<td>thread 1</td>
<td>thread 0</td>
<td>thread 0</td>
</tr>
<tr>
<td>thread 1</td>
<td>thread 0</td>
<td>thread 0</td>
</tr>
<tr>
<td>thread 1</td>
<td>thread 0</td>
<td>thread 0</td>
</tr>
<tr>
<td>thread 1</td>
<td>thread 0</td>
<td>thread 0</td>
</tr>
<tr>
<td>thread 0</td>
<td>thread 0</td>
<td>thread 0</td>
</tr>
<tr>
<td>thread 0</td>
<td>thread 0</td>
<td>thread 0</td>
</tr>
<tr>
<td>thread 0</td>
<td>thread 0</td>
<td>...</td>
</tr>
<tr>
<td>thread 0</td>
<td>thread 0</td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Experiment: results on Mac OS X

thread 0         thread 1
thread 0         thread 1
thread 0         thread 1
thread 0         thread 1
thread 0         thread 1
thread 0         thread 1
thread 0         thread 1
thread 0         thread 1
thread 0         thread 1
thread 0         thread 1

...
Experiment: results

![Graph showing the relationship between number of consecutive printf()s and measured probability for two platforms: Linux EWS and MacOS X.]
Experiment: results

![Graph showing measured probabilities against the number of consecutive printf()s for Linux and MacOS X.]
Take-away point: unpredictability

Scheduling varies across operating systems

Scheduling is non-deterministic even for one OS
- Default (non-real-time) scheduling does not guarantee any fixed length
- Potentially huge variability in work accomplished in one quantum
  - Factor of >10,000 difference in number of consecutive printfs in our experiment!

Quantum may be fairly long (visible to human)
Scheduling: Issues to remember

Why doesn’t scheduling have one easy solution?

What are the pros and cons of each scheduling policy?

How does this matter when you’re writing multiprocess/multithreaded code?
  • Can’t make assumptions about when your process will be running relative to others!
  • May need specialized scheduling for specialized applications
Synchronization

CS 241

February 29, 2012

Copyright © University of Illinois CS 241 Staff
Playing together is not easy

Easy to share data among threads

But, not always so easy to do it correctly...

Easy case: one obvious “owner”
  
  • e.g., main() creates arguments, hands off to child thread
  • child now owns it, no one else will never read or write it

What if threads need to work together? e.g., in web server:
  
  • multiple threads concurrently access cache of files in memory, occasionally adding or removing
  • multiple threads concurrently update count of total # clients
Do threads conflict in practice?

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>

int cnt = 0;

void * worker(void *ptr)
{
    int i;
    for (i = 0; i < 50000; i++)
        cnt++;
}
```
Do threads conflict in practice?

```c
#define NUM_THREADS 2

int main(void) {
    pthread_t threads[NUM_THREADS];
    int i, result;

    for (i = 0; i < NUM_THREADS; i++) {
        result = pthread_create(&threads[i], NULL, worker, NULL);
        assert(result == 0);
    }

    for (i = 0; i < NUM_THREADS; i++) {
        result = pthread_join(threads[i], NULL);
        assert(result == 0);
    }

    /* Print result */
    printf("Final value: %d\n", cnt);
}
```
Do threads conflict in practice?

If everything worked...

```bash
$ ./20-counter
Final value: 100000
```

Q: What are the **minimum** and **maximum** final value?

Q: What value do you expect in practice?

Next time

- How do we guarantee correct interaction between threads? **Synchronization!**
- Guess the final value (win a fabulous prize!)