amotaded Slides

Week 9: Scheduling and Concurrency

CS211: Programming and Operating Systems

Wednesday and Thursday, 11+12 March 2020

This week, in CS211, ...

1. Recall: Scheduling Processes
   - Algorithms
   - Scheduling metrics
   - Round Robin (RR)
   - Example

in class

2. Exercises
3. Concurrency
4. Race condition
5. Critical sections
   - Atomic Operations

Read yourself

6. Locks
7. Synchronization Hardware
8. Semaphores
9. Lab 6

in class

Recall: Scheduling Processes

***For more, see Chapter 7 of the Textbook*.**

Last week we started studying ***SCHEDULING***: algorithms by which the Operating System decides which of the available processes will be given access to the CPU, i.e., set **running**.

First: recall the **states of a process** and how they relate to each other.

.

Recall: Scheduling Processes

We studied four **Scheduling Algorithms**:

1. First-Come-First-Served (FSFS)
2. **Shortest-Job-First** (SJF)

non-premtive

3. Shortest Time-to-Completion First (STCF)
4. Round-Robin (RR)

preemptive.

For each of these, we consider a few examples of process mix; for each example we'll assumed that processes have a single CPU burst, measured in seconds (though this unit is not important).

We compared algorithms according to the following **METRICS**:

1. Turnaround time – the time that elapses between when process arrives in the system, and when it finally completes.
2. Wait time – the amount of time between when a process arrives, and when it completes, that it spends doing nothing.
3. Response time – the time that elapses between when process arrives in the system, and when it executes for the first time.

Recall: Scheduling Processes — Round Robin (RR)

Each process gets a small unit of CPU time called a ***time quantum*** or ***time slice*** –usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are n processes in the ready queue and the time quantum is 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 $(n-1)q$ time units. (***Why?***)

The size of the *quantum* is of central importance to the **RR** algorithm. If it is too large, then its is just the FCFS model. If it is too low, them too much time is spent on context switching.

Suppose the following arrive in the following order:

Proc	Arrive Time	Burst Time
P_1	0	20
P_2	2	15
P_3	4	10
P_4	6	5

Calculate the

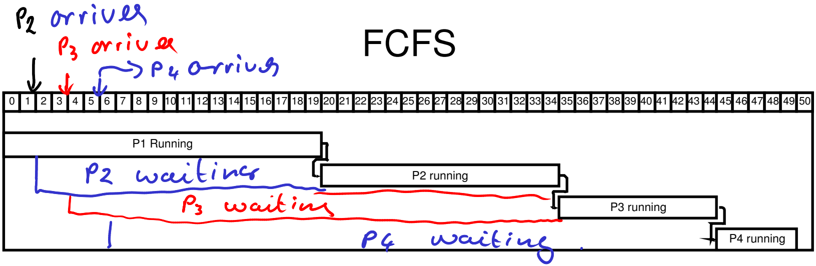
(a) ***Average Turnaround Time***,

(b) ***Average Wait Time***, and

(c) ***Average Response Time*** for

1. FCFS
2. SJF
3. STCF
4. RR with $q = 10$
5. ~~RR with~~ $\cancel{q = 2}$

Recall: Scheduling Processes — Example



Turnarond Times: $20 + 33 + 41 + 44$

Average Turnaround : $(138)/4 = 32.5$

Wait time : $0 + 18 + 31 + 39$

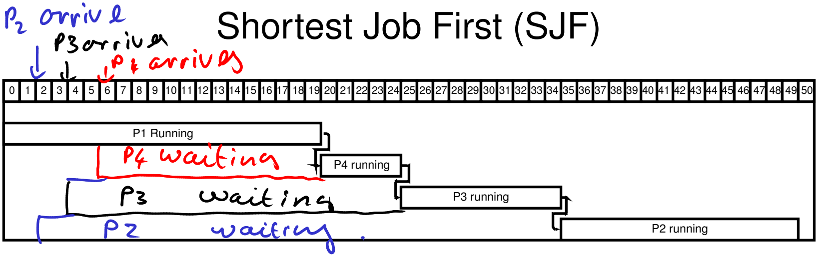
Average is $88/4 = 22$

Response time = wait time (for this example).

Recall: Scheduling Processes

Example

Shortest Job First (SJF)



Turnaround : (20 + 48 + 31 + 19) = 118.

Average : 29.5.

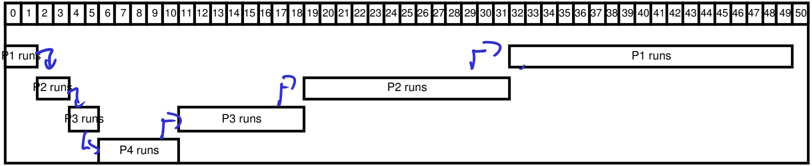
Wait : (0 + 33 + 21 + 16) = 70 Average = 17.5

Response = Wait.

Recall: Scheduling Processes

Example

Shortest Time to Completion First (STCF)



Turnaround time: $50 + 30 + 15 + 5 = 100$

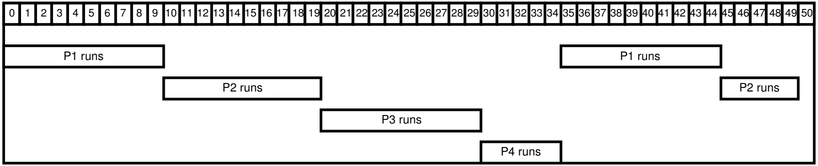
Average $= 25$.

Response time: $0 + 0 + 0 + 0 = 0$.

Ave Wait time? Try yourself!

Recall: Scheduling Processes — Example

Round Robin with $q = 10$



Check

Turnaround Average is 37.

Response Average is 15

Exercises

Exercise (8.1)

(This is taken from the CS211 Semester 2 from 2017/2018)

Given the data (all time in seconds)

Process	Arrival time	Process duration
P1	3	5
P2	1	3
P3	0	8
P4	4	6

for four processes,

determine the scheduling result for the policies of

1. *Round Robin (with time quantum 4)*
2. *First Come First Served*

(c) Calculate the average turnaround time and average waiting time for these examples.

Note: The "wait time" of a process is the length of time it spends doing nothing.

Concurrency

Please read Chapter 26 (Concurrency) of the textbook for much more detail on threads.

A *cooperating* process is one that can affect or be affected by another process that is executing on the system.
Threads are prime examples of this: we can think of them as a single process with multiple points of execution. They share program code and, crucially, data.

In this section, we consider the problems that occue then one or more threads try to access the same data, and we look at potential solutions.
A classic data inconsistency problem is the so-called

"***Race Condition***",

which we'll study in Lab 6 (a more complicated version is discussed in Sections 26.3-26.4).

Race condition

A **Race Condition** (also called a **data race**) is one where the result depends on the order in which instructions are executed.

For a single-thread process, this is predetermined.
But for multi-threaded processes, we do not have control over the order in which individual threads execute their instructions.

Race condition

Consider the following example: two cooperating process called P_1 and P_2 share the variable `count`. At various times during execution either may increment or decrement `count`. The machine usually implements an increment as follows:

1. load the value of `count` into a register: REG_1 `= count`
2. add 1 to the contents of the register: REG_1 `=` REG_1 `+ 1`
3. overwrite the contents of `count` with the contents of the register: `count =` REG_1

and decrement as

1. load the value of `count` into a register: REG_2 `= count`
2. subtract 1 from the contents of the reg: REG_2 `=` REG_2 `- 1`
3. save the contents of the register as `count`: `count =` REG_2

Race condition

Suppose the value of count is 5. If P_1 executes an increment and P_2 executes a decrement, then the value of count should still be 5. Unless the individual operations happen in the following order...

$$
\begin{array}{llcll}
P_1 \text{ executes} & REG_1 & = & count & REG_1 = 5 \\
P_1 \text{ executes} & REG_1 & = & REG_1 + 1 & REG_1 = 6 \\
P_2 \text{ executes} & REG_2 & = & count & REG_2 = 5 \\
P_2 \text{ executes} & REG_2 & = & REG_2 - 1 & REG_2 = 4 \\
P_1 \text{ executes} & count & = & REG_1 & count = 6 \\
P_2 \text{ executes} & count & = & REG_2 & count = 4
\end{array}
$$

We arrive at the wrong state because we allowed both threads to manipulate the variable `count` at the same time.
Since the outcome depends on the order in which each operation takes place, we have a ***race condition***.

Critical sections

Critical Section

(From Section 26.4 of the text-book). A **critical section** is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

Because, in the example above, multiple threads executing this code can result in a race condition, that is an example of a **critical section.**

To resolve this, we would like to enforce **mutual exclusion**: This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

One possible solution is to make the operation **"atomic"** (or ***indivisible***. This is, the critical section is executed as though it were a single operation, and so impossible to interrupt.

In a realistic setting, that is not possible for all race conditions. But, as we will see, the use of some atomic operations can help us solve the larger problem, by creating **locks**.

Locks

See Section 28.1 of the textbook

So now we know we would like to execute a series of instructions atomically. But, in general, on a multiprocessor system, we can't.

But what we can do is create a **lock** which we put around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction.

Locks

For this approach to work, the following 3 conditions must be satisfied:

- **Mutual Exclusion:** If process T_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Fairness/Progress:** If there are some procs that wish to enter the critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely.
- **Performance/Bounded Waiting:** after a process has made a request to enter its critical section and before that request is granted, then must be a **bound** (i.e., a limit) on the number of times other processes are allowed to enter their critical sections.

Synchronization Hardware

We consider two basic approaches to this:

1. Interrupt suspension
2. Automatic ***test-and-set()*** and ***swap()*** instructions.

1. Interrupt suspension (Section 28.5)

Suppose a process is in its critical section. If it cannot be preempted then data consistency should be maintained. On a single processor system, the problem could be solved by disabling interrupts while a shared variable is begin modified. However, such a method is not feasible on a multiproc system: large over-heads would be incurred informing all procs that interrupts are dis-allowed.

Synchronization Hardware

2 Automatic instructions

The processor has facilities to swap the contents of two words (in memory), or test and change the contents of a word *automatically* – i.e., as a single instruction.

There are other hardware and software solutions to synchronization problems. The most important, perhaps, is a tool known as a ***semaphore*** (Chapter 31).

Semaphores

Represents the number of available resources.

A Semaphore S is an integer variable that can only be accessed via one of two operations:

(Test/*sem_wait*) $P(S)$:

```
while (S ⩽ 0)
    { wait(); }
S--;
```

(**Increment/*sem_post***) $V(S)$:

```
S++;
```

(Historically, these functions were called *P*robern (Dutch for "test") and *V*erhogen (increment)).

These operations must be ***indivisible*** (or "atomic"). This is, when one process (or thread) modifies a semaphore value, no other process can modify it at the same time.

Semaphores

There are two types of semaphore:

1. **Binary semaphores (locks):** These are used to control access to a single resource, such as a memory location. If the resources is available then $S = 1$. Otherwise $S = 0$. When a process wants to access it,
   (i) it calls the function $P(S)$
   (ii) enters it critical section
   (iii) calls $V(S)$ when it exits the critical section.
2. **General (or counting) semaphores:** These are used to control access to a pool consisting of a finite number of identical resources. Say there are 5 units available. The S is initialised to 5. Whenever a process requests the resource, it calls $P(S)$ and decrements the value of S. If S reaches 0 then the next proc that requests that resource must wait until another frees it by running $V(S)$.

Lab 6

In our first example, in `adder.c` a (parent process) creates a child process. Then, the child tries to sum of 4 numbers by placing them in a pipe.

The parent then reads these four numbers, adds them, and sends the result to the child via another pipe.

The child then reads this solution and prints it.

Since there are no competing processes, nothing should go wrong (not does it).

Version 1: no problem

adder.c (main)

```c
int main(void )
{
  int ParentsPID, ans;
  pipe(inpipe);
  pipe(outpipe);
  ParentsPID = getpid(); // now I'll always know who I am
  fork(); //Now have 2 procs. Child will have diffrent pid
  if ( getpid() == ParentsPID )
    adder();   // The parent will be the adder
  else
  {
     ans=child(1,2,3,4);
     printf("Child (%d): 1+2+3+4= %d\n", getpid(), ans);
  }
  return(0);
}
```

Version 1: no problem

`adder.c`: `adder()`, run by parent

```c
void adder(void ) // run by parent
{
  int i, number, sum=0;

  for (i=0; i<4; i++)
  {
    read(inpipe[0], &number, sizeof(int));
    sum += number;
  }
  write(outpipe[1], &sum, sizeof(int));
}
```

adder.c: child(), run by child

```
int child(int a, int b, int c, int d)
{
  int ans;
  printf("Child (%d) writes four numbers to the pipe()\n",
  write(inpipe[1], &a, sizeof(int));
  write(inpipe[1], &b, sizeof(int));
  sleep(1); // Pause for a second to encourage race condition
  write(inpipe[1], &c, sizeof(int));
  write(inpipe[1], &d, sizeof(int));

  printf("Child (%d) reads the answer from a pipe()\n", get
  read(outpipe[0], &ans, sizeof(int));
  return(ans);
}
```

The output I get when I run this is:

```
Child (4285) writes four numbers to the pipe()
Child (4285) reads the answer from a pipe()
Child (4285): 1+2+3+4= 10
```

So - no problem!

But in the next version, the parent has two children both doing the same thing. See adder_race_condition.c
Now the output is

```
Child (4485) writes four numbers to the pipe()
Child (4486) writes four numbers to the pipe()
Child (4485) reads the answer from a pipe()
Child (4485): 1+2+3+4= 6
Child (4486) reads the answer from a pipe()
Child (4486): 1+2+3+4= 14
```

In tomorrow's lab we'll design a semaphore solution to this problem

Finished here