Week 8: Threads and Scheduling

CS211: Programming and Operating Systems

Niall Madden (Niall.Madden@NUIGalway.ie)

Wednesday 31 March and Thursday 1 April, 2021

This week, in CS211, . . .

This week, in CS211, . . .

CS211
Week 8: Threads and Scheduling

Start of ...

PART 1: Introduction to threads

Part 1: Introduction to threads

To date we have always assumed that every process has its own **Program Counter** (PC), which tells the OS at where it is in its instruction set. (You can think of this as saying which line of your program it is currently executing).

So even when we duplicate a process using `fork()` the new (sub)process has its own PC (even though, initially, have the same value as the parent's PC).

However, modern OSs allow for **threads**: a single process that can have multiple **PCs**. The reasons for this being useful include:

1. The process may need to preform lots of operations at the same time (on different cores) on the same data set; e.g., in adding vectors.
2. Process have to preform different operations that run at different speeds, such as adding vectors (fast), or writing to a file (slow).

But with, for example, a subprocess made with `fork()`, memory is not really shared, so the advantage of being able to do two things at once is lost by the need to communicate.

Solution: *threads*.

Part 1: Introduction to threads

From the text-book, Chapter 26 (Concurrency and Threads)
`http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf`

Threads

Instead of our classic view of a single point of execution within a program, a **multi-threaded** program has more than one point of execution.
Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.

Part 1: Introduction to threads

A *thread* (or *lightweight process*) is a basic unit of CPU utilisation. Rather than a process begin dedicated to one sequential list of tasks it may be broken up into threads. These consists a set of (distinct)

- **program counter ("PC")** – the next instruction to be executed
- **register set** – operands for CPU instructions
- **stack** – temporary variables, etc.

Threads belonging to the same process share

- code section,
- data section,
- operating-system resources,

collectively known as the ***task***.

A traditional process is sometimes known as a ***heavyweight process***.

Threads differ from the child processes that we looked at last week (created using `fork()`): though a child might inherit a copy of its parent's memory space, a thread shares it.

The reasons that an OS might use threads over heavy-weight processes are:

(i) **Parallelization:** Utilisation of multiprocessor architecture – each thread can execute on a different processor.

(ii) **Responsiveness**: a part of a process may continue working, even if another part is blocked, for example waiting for an I/O operation to occur.

(iii) **Resource sharing**: For example, you can have several hundred threads running at the same time. If they were separate procs, each would need their own memory space. However, as threads they can share that resource.

(iv) **Economy:** Thread creation is somewhat faster than processes creation. Also context switching is faster.

(v) **Efficiency:** Threads belonging to the same proc share common memory space so they do not need support from the OS to communicate.

User and Kernel Threads

Threads may be one of two types: ***User*** or ***Kernel***.

- User threads are implemented by a thread library: a collection is routines and functions for manipulating threads. Hence, user threads are dealt at the compiler/library level – above the level of the OS kernel.
  Advantage: User threads are quick to create and destroy because systems calls are not required.
  Disadvantage: If the kernel itself is not threaded, then if one thread makes a blocking system call, then the entire process with be blocked.

The `POSIX Pthreads` are user threads found on most Unix systems.

- Kernel Threads: thread creation, management and scheduling is controlled by the operating system. Hence they are slower to manipulate then user threads, but do not suffer from blocking problems: if one thread performs a blocking system call, the kernel can schedule another thread from that application for execution.
  Windows has support kernel threads.

CS211
Week 8: Threads and Scheduling

END OF PART 1

Part 2: Contrasting subprocs and threds

CS211
Week 8: Threads and Scheduling

Start of ...

PART 2: Comparing subprocs and threds

In this section, we will write a little code to contrast subprocesses (made by `fork()`), and threads.

We will focus on shared variables.

Part 2: Contrasting subprocs and threds

These programs will have a **global** `int` variable called *GlobalVar*. It is initialised as `0`. In the first example, a subprocess, made by `fork()`, increments *GlobalVar* but we will see that the parent's version is unchanged.

I suggest running these on https://www.onlinegdb.com/

01Fork_MemoryNotShared.c

```c
/* Which: 01Fork_MemoryNotShared.c
   Why:   fork a subproc which will then increment a shared variable.
   Who :  Niall Madden (Niall.Madden@NUIGalway.ie)
   When:  Week 9, 2021-CS211
   Why :  Try to show that a fork()'ed subproc and its parent do
          not share memory.   */

#include <stdio.h>
#include <assert.h>
#include <pthread.h> // not used here, but needed in Example 2
#include <unistd.h>

int GlobalVar = 0; // Global variable; will check if really shared
```

Part 2: Contrasting subprocs and threds

01Fork_MemoryNotShared.c

```
int main(void)
{
  printf("-------------------------------------\n");
  printf("   An example showing that fork()'ed    \n");
  printf(" subprocess has a copy of its parents \n");
  printf("     memory, but it is not shared.    \n");
  printf("START:\t This is Parent Process (PID=%d), and GlobalVar=%d\n",
         getpid(), GlobalVar);

  int pid = fork();
  if (pid == 0) // This is the subprocess
  {
    GlobalVar++;
    printf("This is Subprocess %d, and I think GlobalVariable = %d\n",
           getpid(), GlobalVar);
  }
  else // Parent
  {
    sleep(1); // just to make sure the subprocess does its thing first
    printf("END:\t This is Parent Process (PID=%d), and GlobalVar=%d\n",
           getpid(), GlobalVar);
  }
  return(0);
}
```

Part 2: Contrasting subprocs and threds

When I run this, the output I get is

```
--------------------------------------
  An example showing that fork()'ed
 subprocess has a copy of its parents
     memory, but it is not shared.
START:  This is Parent Process (PID=12720), and GlobalVar=0
This is Subprocess 12721, and I think GlobalVariable = 1
END:  This is Parent Process (PID=12720), and GlobalVar=0
```

Part 2: Contrasting subprocs and threds

On Linux, Macs, and related systems, in C we can create **threads** which are called `pthreads` (which means POSIX thread).

We will use two functions:

- `pthread_create()` which is used to create a thread. It takes four arguments:
  1. ID of the thread process
  2. The thread's "attributes"; we ignore this, and set to *NULL* ;
  3. A function that is called by the thread when it is created;
  4. Argument to pass to that function (ignored in this example).
- `pthread_join()` tells the parent to "wait" for the thread to finish.

These are defined in the *pthread.h* header file. On some compilers (but not onlinegdb), you need to compile with the `-pthread` option.

Part 2: Contrasting subprocs and threds

02Thread_MemoryIsShared.c

```
/* Which: 02Thread_MemoryIsShared.c
   Why:   make a thread that  will then increment a shared variable.
   Who :   Niall Madden (Niall.Madden@NUIGalway.ie)
   When:  Week 9, 2021-CS211
   Why :  Try to show that a thread shares its parent's memory.  */

#include <stdio.h>
#include <assert.h>
#include <pthread.h> // Where the pthread functions are defined
#include <unistd.h>

int GlobalVar = 0; // Global variable; will check if really shared

void *mythread(void *arg) // This is the function the thread will call
{
  GlobalVar++;
  printf("This is Thread with pid=%d, and I think GlobalVariable = %d\n",
         getpid(), GlobalVar);
  return(NULL);
}
```

Part 2: Contrasting subprocs and threds

02Thread_MemoryIsShared.c

```c
int main(void)
{
  printf("-----------------------------------------\n");
  printf("  An example showing that a pthread   \n");
  printf("        shares its parent's memory   \n");

  printf("START:\t This is Parent Process (PID=%d), and GlobalVar=%d\n",
         getpid(), GlobalVar);

  pthread_t p1; // declare the thread's ID
  pthread_create(&p1, NULL, mythread, "N");   // create the thread
  pthread_join(p1, NULL);                     // wait for it to finish

  printf("END:\t This is Parent Process (PID=%d), and GlobalVar=%d\n",
         getpid(), GlobalVar);
  return(0);
 }
```

Part 2: Contrasting subprocs and threds

When I run `02Thread_MemoryIsShared.c` the output I get is

```
---------------------------------------
  An example showing that a pthread
       shares its parent's memory
START:  This is Parent Process (PID=12761), and GlobalVar=0
This is Thread with pid=12761, and I think GlobalVariable = 1
END:  This is Parent Process (PID=12761), and GlobalVar=1
```

Part 2: Contrasting subprocs and threds

CS211
Week 8: Threads and Scheduling

END OF PART 2

CS211
Week 8: Threads and Scheduling

Start of ...

PART 3: Scheduling Processes

Part 3: Scheduling Processes

For more, see Chapter 7 of the Textbook

CPU scheduling is the basis of multiprogrammed operating systems, and so of multitasking operating systems. The underlying concepts of CPU scheduling are:

Goal:	Maximum CPU utilisation – i.e., there should be something running at any given time.
Based around:	The *CPU burst–I/O Burst Cycle* – a running process alternates between executing instructions (CPU burst) and waiting for interaction with peripheral devices (I/O burst)
Depends on:	the burst distribution – is a given proc predominantly concerned with computation or I/O. Also, a proc usually begins with a long CPU burst and almost always ends with a CPU burst.

Preempting

The CPU Scheduler Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. This happens when a process:

(i) Switches from running to waiting state

(ii) Terminates.

(iii) Switches from running to ready state

(iv) Switches from waiting to ready

Scheduling that depends only on (i) and (ii) is ***non-preemptive***. All other scheduling is ***preemptive***.

Preempting

If scheduler uses only *non-preemptive* methods, then once a process is allocated the CPU it will retain it until it terminates or changes its state. E.g., Windows 3.1 (early 1990's), versions MacOS prior to v8 (1998).

For preemptive methods, the scheduler must have an algorithm for deciding when a control of the CPU must be passed to another process.

The Dispatcher

The **Dispatcher** module gives control of the CPU to the process selected by the short-term scheduler; this involves:

1. switching context
2. switching to user mode
3. jumping to the proper location in the user program to restart that program

The dispatcher gives rise to the phenomenon of ***Dispatch latency***, i.e., the time it takes for the dispatcher to stop one process and start another running.

. . . .

We will make the very simplifying assumption that there is no dispatch latency, and that we can switch instantaneously between processes.

CS211
Week 8: Threads and Scheduling

END OF PART 3

CS211
Week 8: Threads and Scheduling

Start of ...

PART 4: Algorithms

Part 4: Algorithms

There are many **Scheduling Algorithms**, including

1. First-Come-First-Served (FSFS)
2. **Shortest-Job-First** (SJF)
3. Shortest Time-to-Completion First (STCF)
4. Round-Robin (RR)
5. Priority Scheduling $\longleftarrow$ (not considered)
6. **Multilevel queuing** $\longleftarrow$ (not considered)

In describing each of these, we'll consider a few examples. For each example we'll assume that each proc has a single CPU burst, measured in milliseconds.

Algorithms will be compared by calculating various metrics.

Single burst means:

Scheduling metrics

We would like to compare algorithms, and determine which is “best”. However, there are many choices of what qualifies as best. Here are a few **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.

First-Come, First-Served (FCFS)

Also called **First In, First Out** (as in OSTEP), the simplest algorithm is FCFS: the first proc that requests the CPU is given it.

Advantage: This is the simplest algorithm to implement, requiring only a **FIFO** queue. It is non-preemptive.

Disadvantage: It has a long average wait time, and suffers from the ***convoy effect***.

First-Come, First-Served (FCFS)

What are the average turnaround, wait and response times for **FCFS** in the following two scenarios?

Example 1

Proc	Arrival Time	Burst Time
P_1	0	10
P_2	0	10
P_3	0	10

Example 2

Proc	Arrival Time	Burst Time
P_1	0	19
P_2	0	10
P_3	0	1

Shortest-Job-First (SJF)

We can see that we could greatly improve the wait and response times for Example 2, if we let P_3 run first, then P_2, and then P_1.

That algorithm is called **Shortest Job First** (SJF): it runs the processes in increasing order of their (expected) burst time.

Advantage: **SJF** is optimal – it gives the minimum average waiting time for a given set of processes. Also: this version in ***non-preemptive***

Disadvantage: It cannot be employed in a realistic setting because we would be required to known the length of next CPU Burst before it happens.

One can only estimate the length. This can be done by using the length of previous CPU bursts, using some form of averaging.

Shortest-Job-First (SJF)

What are the average turnaround, wait and response times for **SJF** for the scenarios we used earlier?

Example 1

Proc	Arrival Time	Burst Time
P_1	0	10
P_2	0	10
P_3	0	10

Example 2

Proc	Arrival Time	Burst Time
P_1	0	19
P_2	0	10
P_3	0	1

Shortest Time-To-Completion First (STCF)

In all the previous examples, we assumed that the process all arrived at the same time. That is not realistic. Consider how **SJF** would work with the following variant on the previous examples:

Example 1

Proc	Arrival Time	Burst Time
P_1	0	10
P_2	1	10
P_3	2	10

Example 2

Proc	Arrival Time	Burst Time
P_1	0	19
P_2	1	10
P_3	2	1

While there is no change for the outcome to Example 1, we now find that SJF is no longer optimal:

Shortest Time-To-Completion First (STCF)

The solution is to introduce a **preemptive** version of SJF, giving **Shortest Time-To-Completion First (STCF)**: whenever a new process arrives to run, we check if it has a lower Burst time than the current running one. If so, we **preempt** that, and let the new one take over until it has finished, or a new one has arrive.

(It is also called ***Shortest-Remaining-Time-First*** or ***Preemptive Shortest Job First***).

Advantage	– “more optimal” than SJF when processes arrive at different times.
Disadvantage	– needs preempting.

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.

Example

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 $q = 2$

FCFS

Example

(Shortest Job First) SJF

(Shortest Time to Completion First) STCF

(Round Robin with $q = 10$

Exercise

Exercise (8.1)

(This is taken from the CS211 Semester 2 from 2017/2018)
Given the data below for four processes, determine the scheduling result for the policies of

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

(All times are in seconds).

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

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