Chapter 19 – Parallel Processing

CPSC 240-09
John Overton

Review

Interrupts

- Interrupts cause the computer to pause what it is currently doing and handle some very important, but short interruption before going back to what it was doing before
- Interrupts can be hardware related or can be software related
- Modern computers can handle 100’s, 1000’s, even millions of interrupts in a second, if needed
- Some interrupts are bad, such as program exceptions (these fall into the software interrupts category)
- Many processor architectures have a software interrupt, which programs use to ask the computer to provide a service for it (in x86_64, the INT instruction has been replaced by the syscall instruction)

Software Interrupts

- Programs cause interrupts all the time. A page fault is a type of software interrupt (of sorts)
- When a software interrupt is not expected, i.e., caused by a software problem, then we call it a Program Exception
- A SIGFAULT (which is also a type of page fault, but usually happens when you have a bad pointer) is a software interrupt and is an exception
- A DIVIDE BY ZERO is a software interrupt and is an exception

Hardware Interrupts

- I/O devices (keyboards, network adapters, disk drives, USB devices, etc)
- Interval timers - There's usually more than one. A system timer provides a constant "tick" interrupt periodically. Another timer is used to notify a program after a requested interval time has concluded
- Other CPUs (on multiprocessor systems)

More about Interrupts

- On many processor architectures, such as the x86_64, there are various privilege levels that instructions can run at. For example, an OS will run at level 0, a “user” program will run at level 3.
- Similarly, interrupt code can run at any of these four levels
- However, in Linux, only 2 levels are used, 0 and 3
- When an interrupt occurs, code that runs due to the interrupt is called an Interrupt Service Routine (ISR)
- Generally, the operating system must set up the address of an ISR. In the x86_64 architecture, the Interrupt Descriptor Table (IDT) is a table of ISRs
- When an interrupt occurs, the processor saves the current RIP register, then loads the appropriate ISR address into the RIP register to being executing the ISR.

More about Interrupts

- When the ISR starts running, it must save the current “context” of the processor so that the processor can restart where it left off after the ISR is finished
- In most processor architectures (software implementation of the Operating System), the ISR is divided into two parts: the first-level interrupt handler (FLIH) and the second-level interrupt handler (SLIH). These are also called the top-half and the bottom-half
- The FLIH usually saves the context, then takes care of any hardware requirements (such as acknowledging the interrupt, resetting hardware and saving any information that may only be available at the time of the interrupt. The FLIH is usually common to many different types of interrupts
- The SLIH is code that is more specific to an interrupt, such as a disk drive interrupt which may need to schedule the next I/O request to the disk drive

Polling Vs interrupts

- In polling, the CPU keeps on checking all the hardware of the availability of any request
- Polling is like standing at the window waiting for someone to come to your door
- The CPU spends wasted time waiting/checking for something to happen

- With Interrupts, the CPU takes care of the hardware only when the hardware requests for some service
- An interrupt is like a doorbell
- The CPU does not waste time waiting/checking, so it can do other things (like running programs)

Parallel Processing

Parallel Processing

- Moore’s Law:

Gordon Moore, then co-founder of Fairchild Semiconductor and future CEO of Intel, said in a 1965 paper that the number of components on an integrated circuit would double every year. In 1975, he revised his prediction to doubling every two years. This prediction has proved very accurate for about 37 years until it slowed in 2012.

- Effectively, with a slowing of Moore’s law, the increase in the capability of new processors (in the form of integrated circuits) has also slowed.
- So, we have reached close to the maximum clock rate on a single processor, therefore we must use multiple processors to distribute the work

Parallel Processing

- Parallel processing is also referred to as concurrency
- Concurrency can refer to either two (or more) programs (or sets of code) executing on separate processors or two (or more) programs executing interleaved, on one processor as if to appear as if two programs were executing on two different processors
- Parallel processing has a connotation that implies processes executing simultaneously, working together to solve a single complex problem

Parallel Processing

- The book talks about two approaches to parallel processing:
  - Multiprocessing is executing tasks on multiple processors, usually in the same computer, to solve a single problem
  - Distributed computing is executing tasks on different computers to solve a single problem (Book example: Folding@home)

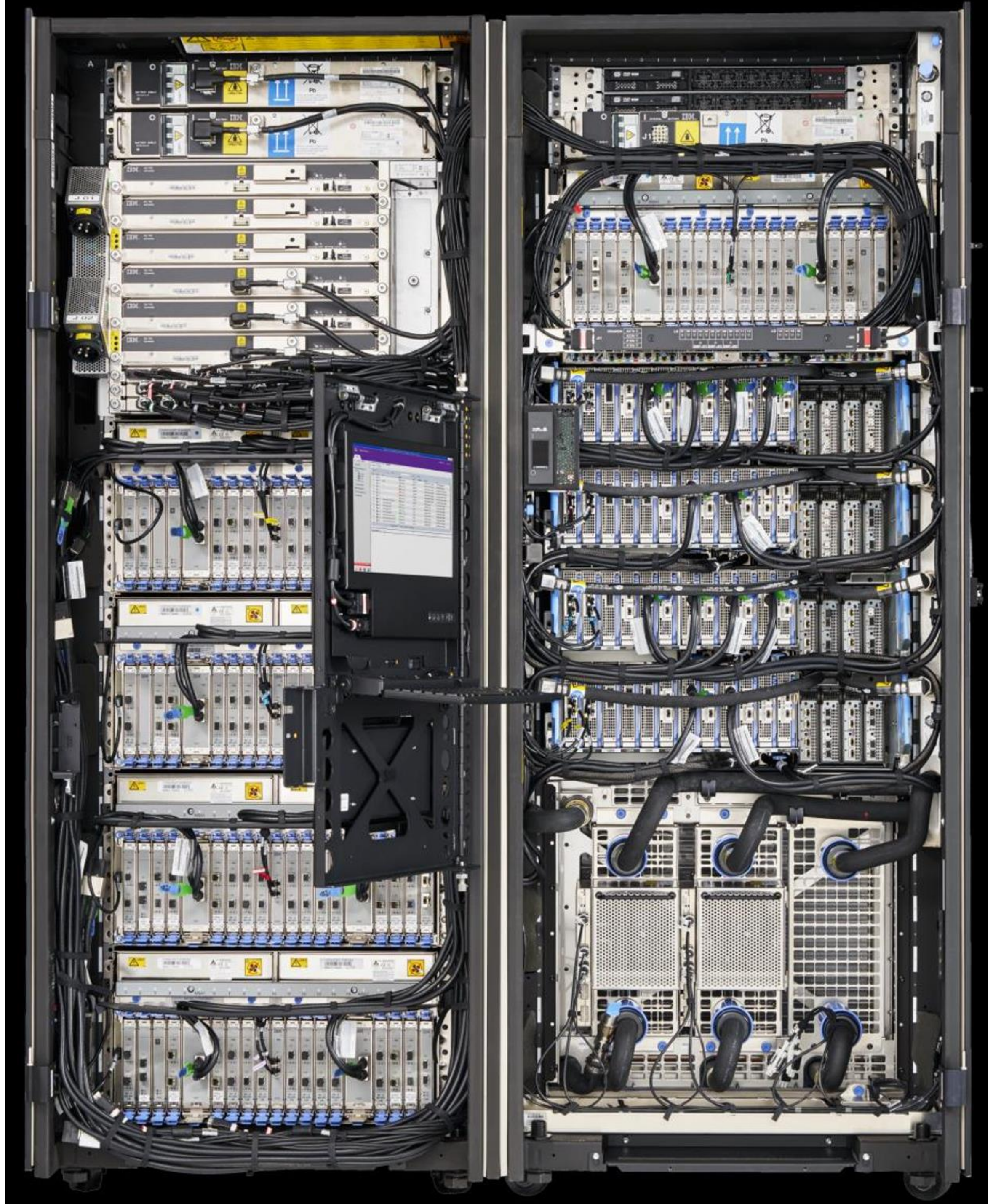
Parallel Processing

- Dividing a programming problem into multiple tasks is quite difficult
- However, if we can break a problem down into sub-problems and if these sub-problems could be executed at the same time, then we would have a total solution that could execute significantly faster using multiple processors than if a single “monolithic” serial-executing solution was running on a single processor
- Not all problems can be divided up and run in parallel

Parallel Processing

- In a multithreaded assembly program, each thread would have their own registers
- Each thread would have its own stack – that is, each thread would have stack space allocated for them from the same shared address space
- All threads share the same .data, .text and .bss section (the share the same address space). This can be a problem if multiple threads were updating a variable in memory at the same time
- However, we can use the stack to allocate "local" variables used by a single thread

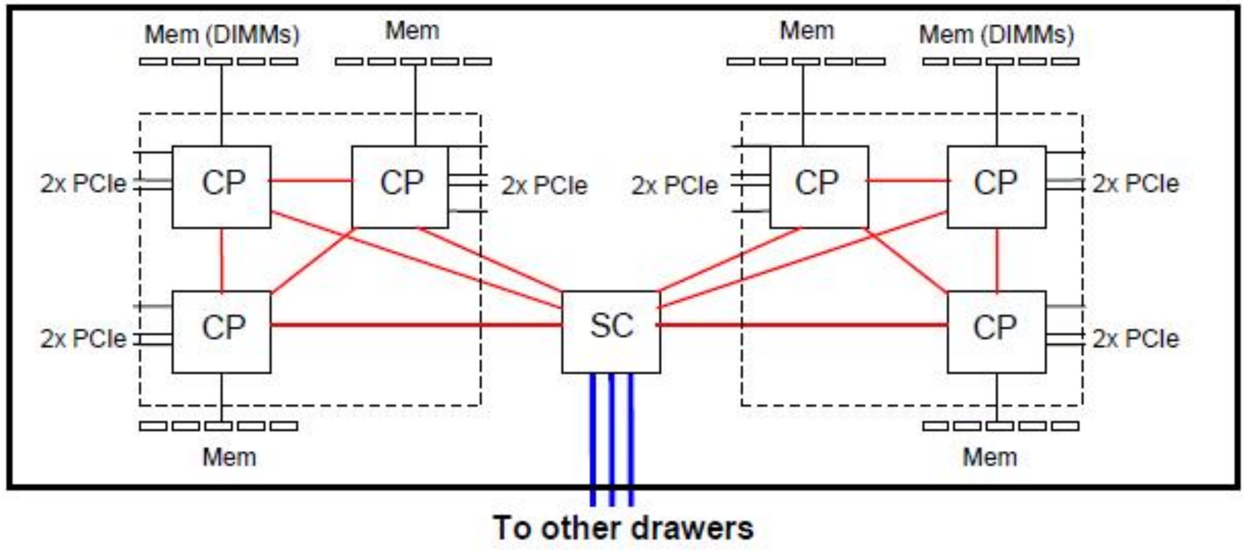
Parallel Processing Case Study
IBM z14 – The modern mainframe

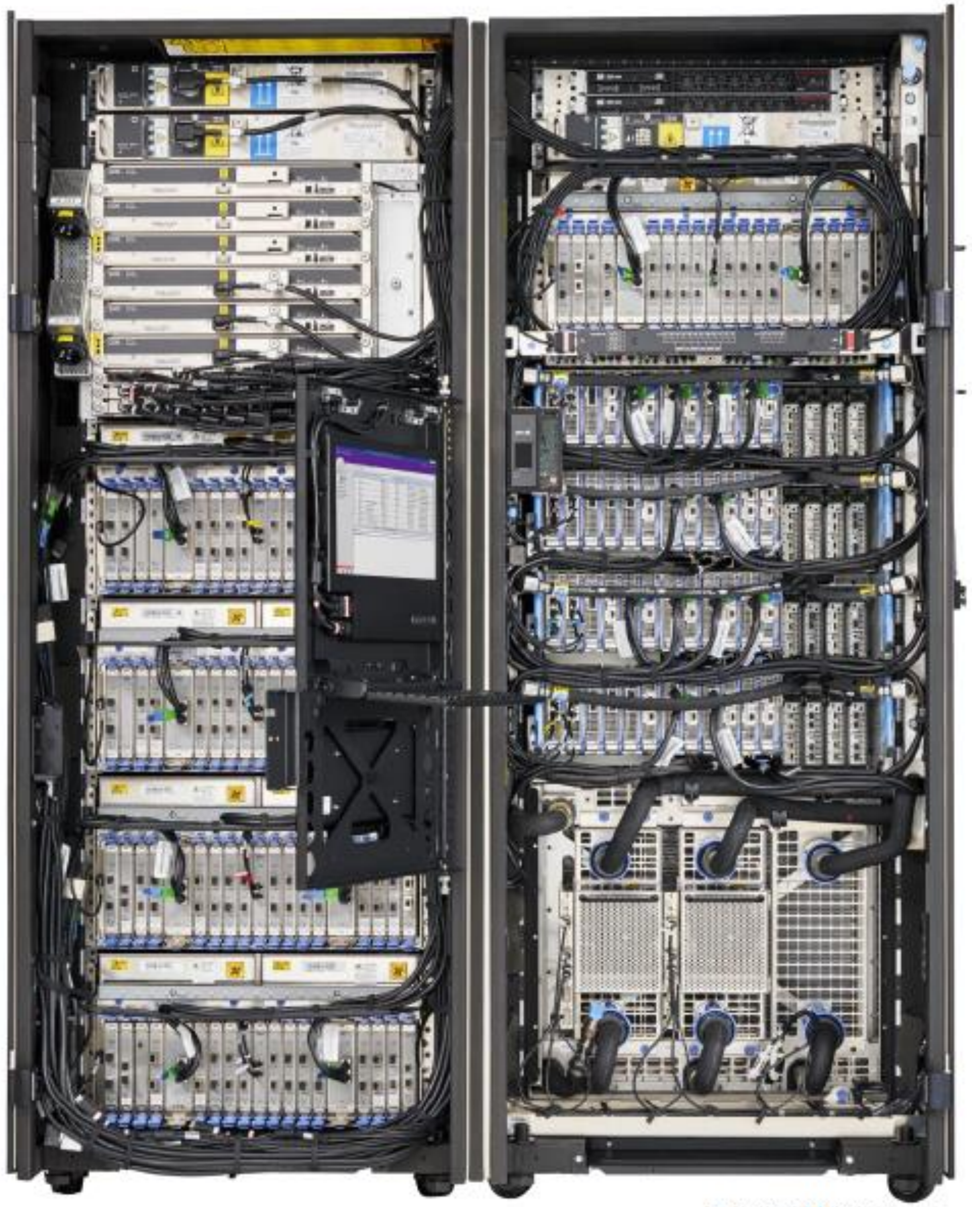


Parallel Processing - Case: IBM z14 – The modern mainframe

- A two rack system
- Contains 4 drawers for processors –
  - Each drawer has up to 6 CP chips and one SC chip
  - Each CP chip contains up to 10 processors (240 total processors)
  - Out of the 240, 170 are used as main processors running at 5.2gHz (with spares)
  - Each CP chip has 6.1B transistors, 14 miles of wire internally
  - An SC chip is responsible for communicating between drawers and I/O channels and also has a L4 (level 4) cache
  - Each SC chip has 9.7B transistors, 13.5 miles of wire internally
- Up to 32 TB of DRAM main memory
- Up to 85 LPARs (virtual machines)

z14 On-Drawer and System Topology





CP chip, 696 sqmm, 14nm, 17 layers of metal

- 10 cores, each 2+4MB I+D L2 cache
- Shared 128MB L3 cache

SC chip, 696 sqmm, 14nm, 17 layers of metal

- System interconnect & coherency logic
- Shared 672MB L4 cache

Max System:

- 24 CP sockets in SMP interconnect
- 32TB RAIM-protected memory
- 40 PCI gen3x16 fanouts to IO-drawers
- 320 IO cards

Parallel Processing – Processes and Threads

- How operating systems keep track of logical units of work: Processes and Threads
  - A process is a program executing in its own “address space” (early definition)
  - A thread is program executing in an existing “address space”
  - In some Oses, a process has a single thread until more threads are created
  - A thread is often referred to as a light-weight process since it will use the initial process’ allocations and address space, thus making it quicker to start (create)
- A threaded approach will not scale to the extent of the distributed approach because a single computer has a finite number of processors and memory, which place an upper bound on the number of simultaneous computations
- However, the distributed solution is bound by network communication (network speed/bandwidth)

Parallel Processing - pthreads

- pThreads is short for POSIX Threads
- pThreads is a thread library that works with the operating system to create and manage threads
- We will use pthread_create(), pthread_join() in our project from the C library
- pthread_create() is a function that will create a new thread and pass a parameter to the new thread. It will return the thread id of the newly created thread
- pthread_join() is a function to wait until a specific thread has ended. Can be used by the main thread to wait on a created thread

Assignment – Phase 1

For phase 1, you will write a program that creates only one thread and runs a subroutine that prints Hello World to the console. This will be the basis for phase 2 where we have multiple threads doing the same work

You will use the code snippets in the book for pthread_create and pthread_join

You can use the following commands to assemble and link (xxxxxxxx is the name of your program):

```
Assemble:  nasm -f elf64 -g -Fdwarf -l xxxxxxxx.lst xxxxxxxx.asm
Link:      gcc -no-pie -m64 -o xxxxxxxx xxxxxxxx.o -lpthread
```

For phase 2, you will need to refresh how to write a for loop…

```
For ( initializers; loop-tests; other-expressions ) {
        statement(s)
}



initializers here (like mov dword [i], 10)
loop:
        Do loop-test here
        …
        statement(s)
        …
        other-expressions
        jmp    loop
```

```
for ( int i = 0; i < MAX_THREADS; i++ ) {
        statement(s)
}
```

```
MAX_THREADS   equ   10

        mov   qword [i], 0
loop:
        cmp    qword [i], MAX_THREADS
        jge    loop_end
        …
        statement(s)
        …
        inc     qword [i]
        jmp     loop
loop_end:
```