Name: Yang Qi Uid:u1387749

Question 1

global fibonacci

section .text

fibonacci:

push rbp ; save previous frame pointer

mov rbp, rsp ; set current frame pointer

mov rax, [rbp+8] ; get argument num

cmp rax, 1 ; num = 1?

jnbe else ; if not, go do another statement num = 2?

mov rcx, 0 ; otherwise return 0

else:

cmp rax, 2 ; num = 2?

jnbe L1 ; if not, go do a recursive call

mov rcx, 1 ; otherwise return 1

jmp exit

L1:

dec rax ; num-1

mov rdx, rax ; num-1 to rdx

push rdx ; save num-1 on stack

push rax ; set argument = num-1

call fibonacci ; fibonacci(n-1), result goes in %rcx

pop rax ; pop num -1

dec rax ; num - 2

push rcx ; save fibonacci(num-1)

push rax ; set argument = num -2

call fibonacci ; fibonacci(num-2),

pop rax ; fibonacci(num-1)

add rcx, rax ; fibonacci(num-1) + fibonacci(num-2), stored in %rcx

exit:

mov rsp, rbp ; reset stack to value at function entry

pop rbp ; restore caller's frame pointer

ret ; and return

Question 2

1. The difference between threads and processes:

Fundamental difference: the process is the basic unit of operating system resource allocation, while the thread is the basic unit of processor task scheduling and execution.

Resource overhead: each process has independent code and data space, and switching between programs is a large overhead; threads can be regarded as lightweight processes, the same type of threads share code and data space, each thread has its own independent running stack and the program counter (PC), and the overhead of switching between threads is small.

Inclusion relationship: If there are multiple threads in a process, the execution process is not performed by one line, but by multiple lines (threads); threads are part of the process, so threads are also called light-weight processes.

Memory allocation: Threads of the same process share the address space and resources of the process, and the address space and resources between processes are independent of each other. After a process crashes, it will not affect other processes in protected mode, but One thread crashes and the whole process dies. So multiprocessing is more robust than multithreading.

Execution process: Each independent process has an entry for program operation, a sequential execution sequence, and a program exit. However, threads cannot be executed independently, they must depend on the application program, and the application program provides multiple thread execution control, both of which can be executed concurrently

1. Process scheduling

Scheduling is moving ready process from ready queue to start running on the CPU.

A way of a process is removed from the running: I/O

Workload Assumptions:

1. Each job runs for the same amount of time

2. All jobs arrive at the same time

3. All jobs only use the CPU (no I/O)

4. Run-time of each job is known in advance

Scheduling metrics: a measurement of process quality. Minimize turnaround time, response time, waiting time, throughput, resource utilization, overhead, fairness.

Scheduling policies(schedulers): First In, First Out; Shortest Job First; Shortest Time to Completion First; Round Robin

1. System calls:

fork(): The fork() system call is used to create a new process. the process that is created is an (almost) exact copy of the calling process.That means that to the OS, it now looks like there are two copies of the program running, and both are about to return from the fork() system call. The newly-created process (called the child, in contrast to the creating parent) doesn’t start running at main().

exec(): This system call is useful when you want to run a program

that is different from the calling program.

wait(): It is quite useful for a parent to wait for a child process to finish what it has been doing.

1. context switch

It is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single central processing unit, and is an essential feature of a multitasking operating system

Performs context-switch:

•Switch from user mode to kernel mode

•Save execution state (registers) of old process in PCB

•Insert PCB in ready queue

•Load state of next process from PCB to registers

•Switch from kernel to user mode

•Jump to instruction in new user process

Question 3

Locality is typically described as having two distinct forms: temporal locality

and spatial locality. In a program with good temporal locality, a memory location

that is referenced once is likely to be referenced again multiple times in the near

future. In a program with good spatial locality, if a memory location is referenced once, then the program is likely to reference a nearby memory location in the near

Future.

Programmers should understand the principle of locality because, in general,

programs with good locality run faster than programs with poor locality. All levels

of modern computer systems, from the hardware, to the operating system, to

application programs, are designed to exploit locality. At the hardware level, the

principle of locality allows computer designers to speed up main memory accesses

by introducing small fast memories known as cache memories that hold blocks of

the most recently referenced instructions and data items. At the operating system

level, the principle of locality allows the system to use the main memory as a cache

of the most recently referenced chunks of the virtual address space. Similarly, the

operating system uses main memory to cache the most recently used disk blocks in

the disk file system. The principle of locality also plays a crucial role in the design

of application programs. For example,Web browsers exploit temporal locality by

caching recently referenced documents on a local disk. High-volumeWeb servers

hold recently requested documents in front-end disk caches that satisfy requests

for these documents without requiring any intervention from the server.

Locality of References to Program Data(details):

For example:

int sumvec(int v[N])

{

int i, sum = 0;

for (i = 0; i < N; i++)

sum += v[i];

return sum;

}

the elements of vector v are read sequentially, one after the other, in the order they are stored in memory (we assume for convenience that the array starts at address 0). Thus, with respect to variable v, the function has good spatial locality but poor temporal locality since each vector element is accessed exactly once. Since the function has either good spatial or temporal locality with respect to each variable in the loop body, we can conclude that the sumvec function enjoys good locality.

Question 4

1. Physical memory: Physical memory refers to the RAM or the primary memory in the computer.

Virtual memory: It is a memory management technique performed by the operating system. Virtual memory allows the programmer to use more memory for the programs than the available physical memory.

1. In physical memory, there are some fragmentations such as internal fragmentation and external fragmentation which causes the problem of insufficient contiguous space. So paging is introduced to slove contiguousness restriction by eliminating external fragmentation and growing segments as needed. The register and MMU is involved in this process.
2. In computing, there are two types of processors existing, 32-bit and 64-bit processors. These types of processors tell us how much memory a processor can access from a CPU register. A 32-bit system can access 2^32 different memory addresses, A 64-bit system can access 2^64 different memory addresses.
3. Typical virtual address space lay out:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

↓ ↘ ↙ ↘↙ ↘↙ ↘↙ ↘↙

Kernel Stack Shared Libraries Heap Data Code

1,Stack region is at top,

and can grow down

2,Heap has free space to

grow up

3,Text is typically read-only

4,Kernel is in a reserved,

protected, shared region

1. The methods used in traditional 32-bit systems result in a five-level PT tree for 64-bit address spaces. The additional levels increase the cost of performing translations in the case of TLB misses.

Question 5

1. Threading: A new abstraction for a single running process that of a thread. 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.
2. Hardware level such as:

Fetch instructions from multiple threads at once, use large CPU

registers to hold data.

In addition to SMT (simultaneous multithreading), put a thread (or more) on each core.

Graphics processing units (GPUs) allow for massive multithreading.

Software level such as: User thread, Kernel thread

Thread models: Many to One, One to One, Many to Many

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. This protected section is the critical section.

Two possibilities to fix the interruption of cretical section:

1. “Wrap” the critical section in some structure, only one thread

inside it at a time (synchronization)

• The data structure is called a mutex (mutual exclusion).

2. Have a single assembly instruction corresponding to the critical

section.

• The single instruction is called an atomic instruction.

Mutex:

A mutex is a software abstraction. In computer science, mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions.

Conditional variable:

A conditional variable in operating system programming is a special kind of variable that is used to determine if a certain condition has been met or not.

Barrier:

In parallel computing, a barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

Implementing a barrier using mutex or conditional variable:

class Barrier {

public:

Barrier(unsigned int n);

void Wait(void);

private:

std::mutex counter\_Mutex;

std::mutex wait\_Mutex;

unsigned int threshold\_N;

unsigned int current\_threads\_N;

};

Barrier::Barrier(unsigned int n) {

threshold\_N = n;

current\_threads\_N = threshold\_N;

}

void Barrier::Wait(void) {

counter\_Mutex.lock();

if (current\_threads\_N == threshold\_N) {

wait\_Mutex.lock();

}

--current\_threads\_N; // Decrease thread counter

if (current\_threads\_N== 0) {

current\_threads\_N= threshold\_N;

wait\_Mutex.unlock();

current\_threads\_N = threshold\_N;

counter\_Mutex.unlock();

} else {

counter\_Mutex.unlock();

wait\_Mutex.lock();

wait\_Mutex.unlock();

}

}