Program: Instruction들의 집합

CPU Control Flow: CPU가 a sequence of instruction들을 한번에 하나씩 읽어서 execute(interpret)하는데, 이러한 Sequence를 CPU Control Flow라고 함

Control Flow를 Change 하는 경우

* program state의 변화(프로그래머가 의도한 event)
* Jumps and Branches
* Subroutine을 호출하고 Subroutine로부터 return 되는 경우
* system state의 변화(프로그래머가 의도하지 않은 system의 상태 변화)
* divides by zero
* user hit ctrl-c
* system timer expires
* data arrives from a disk or network adapter
* 따라서 System은 Exceptional control flow을 위한 mechanism 필요

Exceptional Control Flow

* computer system의 모든 level에서 존재
* Low level mechanisms
* Exceptions
* system event에 대한 반응으로 control flow change
* hardware와 OS software의 조합을 이용해 구현
* High level mechanisms
* Process context switch
* Hardware timer와 OS Software에 의해 구현됨
* Signals
* OS Software에 의해 구현됨
* Nonlocal jumps
* C runtime library에 의해 구현됨

User code: 사용자에게 할당된 주소 공간만 access 가능

Kernel code: 메모리의 모든 주소 공간 access 가능

Exception

* some event에 대한 반응으로 OS Kernel에 control을 넘겨줌(즉 processor state 변경)
* Kernel은 memory에 상주하는 OS의 일부
* Divide by 0, arithmetic overflow, page fault, I/O request completes, typing CTRL-C
* 각 type의 event는 unique exception number k를 가지고 있음
* k(interrupt vector)는 exception table의 index
* exception k가 일어나면 exception table의 index k 에 등록되어 있는 handler가 호출됨

Exception의 종류

* Interrupt
* Asynchronous(프로그램 수행의 결과물이 아님, 언제 발생할지 모름)
* current instruction을 다 끝내고 handler를 수행한 후 next instruction 수행
* 프로세서 외부의 이벤트에 의해 발생
* 프로세서의 interrupt pin을 set하여 나타냄
* ex) timer interrupt, I/O interrupt from External device(CTRL-C, Arrival data or packet)
* Trap
* Synchronous(프로그램 수행의 결과물)
* current instruction을 다 끝내고 handler를 수행한 후 next instruction 수행
* 의도적으로 발생시킨 event
* ex) system calls, breakpoint traps, special instructions
* Fault
* Synchronous(프로그램 수행의 결과물)
* handler를 수행하고 current instruction을 다시 실행 시키거나 abort 시킴.
* 의도적이지 않은 event이지만 회복 가능할 수 있음
* ex) page fault(회복 가능), protection faults(회복 불가능), floating point exceptions, Invalid memory Reference(Segmentation fault)
* Abort
* Synchronous(프로그램 수행의 결과물)
* handler에서 return 되지 않고 프로세스가 죽음
* 의도적이지 않은 event이고 회복 불가능
* ex) illegal instruction, parity error[hardware적으로 비트 flip이 일어난 경우], machine check

System Calls

* 각 System call은 unique ID number를 가지고 있음
* rax는 syscall number를 가지고 있음
* system call의 결과를 rax에 저장하여 return
* negative value는 음수 errno에 해당하는 error나타냄.

Process

* 실행중인 program의 인스턴스(active)
* Logical Control flow
* 각 프로그램은 CPU를 독점적으로 사용하는 것처럼 보임.
* Context Switching이라고 불리는 kernel mechanism에 의해 제공됨.
* Private Address space
* 각 프로그램은 Main Memory를 독점적으로 사용하는 것처럼 보임
* Virtual Memory라고 불리는 Kernel mechanism에 의해 제공됨

Multiprocessing

* CPU는 어떤 순간에 하나의 process만 돌릴 수 있음
* CPU는 굉장히 빠르게 context switching을 하여 많은 프로세스들을 동시에 돌림.
* 따라서 각 프로세스에서는 CPU를 혼자 독점해서 사용하는 듯한 환상
* Context: 각 Process가 가지고 있는 의미 있는 정보(PC, Value of CPU Registers, Process state등)
* Single core: multiple processes를 concurrently 실행
* Multicore: 각 core는 별도의 프로세스를 실행할 수 있음(Scheduling은 kernel에 의해 됨)

Concurrent Processes

* 각 프로세스는 logical control flow임
* flow가 시간상 겹치는 경우 두 프로세스는 concurrent하게 도는 것임
* flow가 시간상 겹치지 않는 경우 sequential하게 도는 것임
* Concurrent process들의 Control flow는 시간상 물리적으로 분리되어 있지만 서로 병렬적으로 도는 것으로 생각할 수 있음.

Context Switching

* process는 kernel이라는 memory 상주 OS code에 의해 관리됨
* kernel은 별도의 프로세스가 아니라 기존 프로세스의 일부로써 실행됨.
* Control flow는 Context Switch를 통해 한 Process에서 다른 Process로 전달됨.

System Call Error Handling

* Error가 발생하면 Linux System은 일반적으로 -1을 반환하고 원인을 나타내기 위해 global variable errno를 set함

Process State

* Running: Process가 실행되거나 실행되기를 기다리는 상태
* Stopped: Process의 실행이 Suspended되고 future notice전까지 더이상 Schedule되지 않음
* Terminated: Process가 영구적으로 멈춘 상태(종료된 상태)

Terminating Process

* Process는 Signal, main routine에서의 return, exit의 호출에 의해 종료됨
* exit은 한번만 호출되고 return하지 않음

Creating Process

* Parent process는 fork를 호출하여 child process를 생성
* 자식 process는 fork의 결과로 0을 반환하고, 부모 process는 fork의 결과로 자식 프로세스의 pid를 반환
* 자식 프로세스는 부모와 동일한 virtual address space와 file descriptor의 복사본을 가지고 있게 됨
* 부모 프로세스와 자식프로세스는 이후로 concurrent하게 돌게 됨.

Reaping Child Process

* process가 종료되더라도 process는 system resource를 소비하고 있게 됨(zombie)
* parent process가 wait을 통해서 종료된 child의 reaping을 해주게 됨
* parent는 child의 exit status를 받게 됨
* parent가 child를 reaping하지 않고 끝나게 되면 orphan child는 init process에 의해서 reaping 됨
* wait: children중 하나가 종료될 때까지 current process를 suspend함
* waitpid: 특정 process가 끝날 때까지 current process를 suspend함

EXECVE

* Current Process에 프로그램을 Load하고 실행
* Code, Data, Stack을 Overwrite함
* PID, open files, signal context는 유지함

Shell: 사용자를 대신하여 프로그램을 실행시켜주는 Application Program

Inter Process Communication

* 다양한 프로세스가 그들 사이에서 서로 통신할 수 있게 하는 메커니즘
* Ordinary Pipe
* 한 프로세스를 다른 프로세스와 연결하는 단방향 바이트 스트림
* 한쪽 끝에서 써진 데이터는 다른 쪽 끝에서 읽음
* 구조화되지 않은 communication(파이프에 포함된 데이터의 크기, 발신자/수신자를 알 수 없음)
* pipe에 대한 access는 file descriptor를 통해 이루어짐.
* 단점
* 프로세스가 종료되면 pipe가 사라짐
* 한 쌍의 프로세스만 통신하도록 허용
* Named Pipe(FIFO)
* 양방향 통신이며 부모-자식 관계가 아님.
* 파일 시스템에서 pipe와 관련된 entry가 관리됨(파일처럼 관리되지만 디스크 공간의 사용은 아님)
* 여러 writer를 가질 수 있음
* process가 종료되더라도 Named Pipe는 존재함

Signal

* 시스템에서 특정 유형의 이벤트가 발생했음을 프로세스에 알리는 small message
* Kernel은 Destination Process의 Context의 일부 상태를 업데이트하여 Signal을 전송
* Kernel이 System의 Event를 발견하거나, 다른 Process가 kill system call을 통해 Destination Process로 Signal을 보내달라고 Kernel에게 요청한 경우 Kernel은 Signal을 보냄

Receiving Signal

* Received : Destination Process가 Signal이 도착한 것을 확인하고 Action을 취한 것
* Destination Process는 Signal을 바로 확인하지 않고 (time quantum) Context Switch를 통해 CPU를 점유하고 나서 User Code를 수행하기 이전 순간에 확인
* Ignore: Signal을 무시함
* Terminate: 프로세스를 종료함
* Catch: Signal handler라는 user level function을 수행
* Pending: Signal이 deliver되었지만 Destination에 의해 아직 Receive되지 않은 상태
* 특정 type에 대해 최대 하나의 pending Signal이 있을 수 있음(Signal은 queueing되지 않음)
* Block: Process는 특정 Signal의 수신을 block할 수 있음
* Blocked Signal은 deliver되지만 signal을 unblock하기 전까지는 receive 되지 않음

Pending/Blocked Bits

* Kernel은 pending, block bit vector를 각 process의 context에 유지
* pending bit vector
* kernel은 type k의 signal이 deliver되면 pending bit vector의 bit k를 set하고, received되면 clear함
* blocked bit vector
* Process가 system call을 통해 kernel에 block을 요청하면 Kernel이 해당 Signal을 Set하거나 Clear

Process Group

* 각 Process는 정확히 하나의 Process Group에 속함
* /bin/kill: program이 signal을 process 또는 process group에 보냄
* pid 쪽에 -를 붙이면 process group에 signal을 보냄.
* Ctrl-c, Ctrl-z는 Foreground Process group에 있는 모든 job에 Kernel이 SIGINT(SIGTSTP)을 보내도록 함

Receiving Signals

* Kernel이 pnb = pending & ~blocked를 계산
* 처리해줄 Signal이 있으면(pnb!=0이면)
* pnb에서 0이 아닌 least bit k를 선택하고 프로세스 p가 signal k를 receive하도록 함
* pnb에서 0이 아닌 모든 k에 대해서 반복
* p에 대한 logical flow의 next instruction으로 control을 넘김

Signal Handler

* User Level의 Signal Handler를 작성하고 이를 등록하여 특정 Signal에 대한 Action을 지정해 줄 수 있음
* Signal Handler는 main program과 동시에 실행되는 별도의 Logical Flow임
* Handler는 다른 Handler에 의해 Interrupt될 수 있음(Nested Signal Handler)

Blocking and Unblocking Signal

* Implicit blocking Mechanism
* Kernel은 현재 handle하는 유형의 Pending Signal을 Block함
* ex) SIGINT Handler는 다른 SIGINT에 의해 Interrupt될 수 없음
* Explicit blocking and unblocking mechanism
* sigprocmask function

Safe Signal Handling

* Handler는 Main Program과 Concurrent하고 Global Data Structure를 공유하기 때문에 까다로움
* handler는 간단하게 작성하는 것이 좋음
* async-signal-safe function을 사용하는 것이 좋음
* 모든 Signal을 일시적으로 차단하여 Shared Data Structure에 대한 Access를 보호
* Global Variable을 volatile로 선언
* Global Flag를 volatile sig\_atomic\_t로 선언

Async-Signal-Safety

* 함수가 reentrant한 경우나 signal에 non-interruptible한 경우

Unix I/O Overview

* Linux File은 Sequence of chunk(m bytes)임
* 모든 I/O device는 file처럼 표현됨
* kernel 역시 file처럼 표현됨

File Types

* 각 파일에는 시스템에서의 역할을 나타내는 type이 있음
* Regular files
* 임의의 data를 포함하고 있음
* Application은 text file과 binary file을 구분(kernel은 차이를 모름)
* text file은 sequence of text line임
* Directories
* Directory는 link들의 array로 구성되어 있음
* 각 링크는 filename을 파일에 mapping함
* 모든 파일은 /라는 root directory에 고정된 계층 구조로 구성
* kernel은 각 process마다 current working directory를 maintain하고 있음

Read Files

* current file position부터 특정 bytes를 파일에서 메모리로 읽어오고, file position을 update
* read는 file fd로부터 buf로 읽어온 byte의 수를 반환
* Short count: 읽고자 하는 크기(n)만큼 파일에 데이터가 없는 경우 n보다 작은 값이 반환되는 것

Writing Files

* 특정 bytes를 메모리에서 파일로 쓰고(current file position부터), file position을 update
* write는 buf로부터 file fd에 쓴 byte의 수를 반환
* read와 마찬가지로 short count가 발생할 수 있음

Short Count

* Reading시 EOF를 만난 경우 발생할 수 있음
* terminal로부터 text line을 읽을 때 발생할 수 있음
* network socket에 쓰고 읽을때 발생할 수 있음
* Reading시 EOF를 만나지 않은 경우에는 shortcount가 발생할 수 없음
* Disk file에 쓰는 경우 shortcount가 발생할 수 없음

RIO Package

* Network Program과 같은 Application에서 I/O를 효율적으로 강인하게 할 수 있도록 하는 wrapper함수의 집합
* Network Programming을 할 때에 효율적
* 두가지 기능
* Unbuffered input and output
* network socket으로 데이터를 전송할 때 유용
* rio\_readn
* EOF를 만날 때에만 short count를 반환
* 읽을 byte의 수를 알 때에 사용
* rio\_writen
* shortcount를 반환하지 않음
* 동일한 descriptor에서 임의로 interleave할 수 있음
* Buffered Input
* file을 읽을 때, 여러 번 system call을 호출하는 것이 아닌 internal memory에 caching을 해와 이로부터 효율적으로 읽음
* rio\_readlineb
* network socket로부터 text line을 읽을 때 유용
* Stopping Condition: Maxlen byte만큼 읽은 경우, EOF만난 경우, \n 만난 경우
* rio\_readnb
* n byte만큼을 file fd로부터 읽음
* Stopping Condition: maxlen byte만큼 읽은 경우, EOF만난 경우
* thread-safe하고 동일한 descriptor에서 임의로 interleave할 수 있음
* 단, rio\_readn과 interleave하면 안됨

Buffered I/O

* File로부터 여러 bytes를 한번에 읽어와 internal memory buffer에 저장(caching)을 해놓고 user code에서 user가 읽고 싶은 만큼을 Buffer로부터 읽게 하여 System Call 횟수를 줄일 수 있도록 해줌

File Metadata

* 파일의 data에 관한 data
* 각 파일 별로 metadata가 kernel에 의해 maintain됨

File Sharing

* open을 두 번한 경우
* 2 distinct file descriptor가 2 distinct open file table entries를 각각 가리키고 두 entry는 모두 하나의 v-node table entry를 가리킴
* child가 parent를 inherit한 경우
* child의 file descriptor와 parent의 file descriptor가 하나의 open file table entry를 가리키고 해당 entry의 refcnt가 2가됨

Redirection

* dup(oldfd, newfd)
* old fd의 descriptor table entry를 newfd의 entry로 복사

Standard I/O

* 일반적으로 buffered I/O
* Standard I/O는 open files을 stream으로 봄
* ‘\n’, exit, return from main, fflush시 출력 fd로 버퍼가 flush됨

Unix I/O vs Standard I/O vs RIO

* Standard I/O와 RIO는 Unix I/O를 이용하여 구현됨
* RIO와 Standard I/O를 섞지 않는 것이 좋음
* UNIX I/O
* async-signal-safe하기 때문에 Signal handler안에서 적합
* 장점
* 가장 일반적이고 overhead가 낮은 I/O
* 파일에 대한 metadata 접근하는 함수를 제공
* async-signal-safe하고 signal handler에 의해 안전하게 사용 가능
* 단점
* short count를 다루기 까다로움
* 효율적인 read는 buffering을 필요로 하는데, 역시 까다로움
* Standard I/O
* Disk나 terminal file 작업에 적합
* 장점
* Buffering이 read와 write system call의 횟수를 줄여주기 때문에 효율적
* short count가 자동적으로 handle됨
* 단점
* 파일에 대한 metadata를 접근하는 함수를 제공하지 않음
* async-signal-safe하지 않고 따라서 signal handler에 적절하지 않음
* network socket의 input, output에 대해 적합하지 않음

////Network Programming

Client Server

* Server는 resource를 manage하고 resource를 가공하여 client들에게 service를 제공
* Server는 Client의 request에 의해 활성화

Network

* 지리적 근접성에 따라 구성된 계층적 시스템
* SAN(cluster or machine room), LAN(building, campus), WAN(country or world)
* Internet: interconnected set of networks
* Ethernet Segment: wire로 hub에 연결된 호스트들의 모음, room or floor
* Bridged Ethernet Segment: Bridge는 어떤 host을 어떤 port로 접근할 수 있는지 self learning, building or campus
* Router : 여러 incompatible Lan들을 물리적으로 연결
* Protocol: host와 Router가 네트워크에서 네트워크로 데이터를 전송할 때 협력하는 방식을 제어하는 규칙
* Internet Protocol: naming scheme(host address)과 delivery mechanism(packet[header, payload])제공

Global IP Internet

* IP
* basic naming scheme과 host와 host사이의 packet의 unreliable delivery 제공
* UDP
* process와 process 사이의 unreliable datagram delivery 제공
* TCP
* connection을 통해 process와 process사이의 reliable byte streams제공
* Unix file I/O와 socket interface의 기능을 혼합하여 접근

Programmer View of Internet

* Host는 32-bit IP addresses에 Mapping
* IP address들은 domain names에 Mapping
* Host안의 Process는 connection을 통해 다른 host안의 process와 통신 가능

IPv4 and IPv6

* IPv4: 32bit Address
* IPv6: 128bit Address

IP Addresses

* IP address struct에 저장되어 있음
* IP address는 memory안에 network byte order(bit endian byte order)로 저장되어 있음

Domain Name System(DNS)

* Internet은 IP address와 domain name의 mapping 정보를 DNS라는 distributed database에 유지함
* loopback address: 127.0.0.1(localhost address)

Internet Connections

* Client and Server는 connection을 통해 bytes stream을 전송하면서 통신
* Socket
* Kernel 관점: Endpoint of a connection(Socket Address=IPaddress:port)
* Application 관점: 네트워크로/로부터 읽고/쓸 수 있도록 하는 file descriptor
* 네트워크를 포함한 모든 Unix I/O 장치는 파일로 Modeling됨
* port: process을 식별하는 16 bits 정수
* Ephemeral port: client가 connection 요청 시 client kernel에 의해 자동으로 할당된 포트
* Well-know port: Server에서 제공하는 특정 서비스와 관련된 포트
* Connection은 Socket Pair(cliaddr:clipart, servaddr:servport)를 통해 식별됨

Socket

* Client와 Server는 Socket Descriptor로부터/로 읽고/씀으로써 서로 통신
* open\_listenfd(getaddrinfo+socket+bind+listen)
* open\_clientfd(getaddrinfo+socket+connect)
* connect는 server의 accept와 연결됨

getaddrinfo

* String 표현 방식의 hostname, host address, ports, service name을 socket address structure로 바꾸어 줌
* addrinfo 구조체의 linked list의 point를 반환
* Client는 socket 과 connect 함수가 성공할 때까지 list 탐색
* Server는 socket과 bind함수가 성공할 때까지 list 탐색
* 장점
* Reentrant(thread safe함)
* Protocol에 independent한 code를 작성할 수 있도록 해줌(IPv4, IPv6)
* 단점
* 복잡함
* getnameinfo는 getaddrinfo의 반대 작업을 해줌

Socket Interface

* Server는 Client로부터 오는 Connection Request을 Queueing함
* 다른 Client는 연결되어있는 Client의 연결이 끊길 때까지 기다려야함
* Client와 Server는 Socket Descriptor를 생성하기 위해 socket function을 이용
* Server는 bind function을 통해 커널에 Server의 Socket주소를 Socket Descriptor와 연결하도록 요청
* 기본적으로 Kernel은 Socket function으로부터 나온 sockfd가 Client쪽에 있는 Active Socket이라고 가정하기 때문에 Server는 listen function을 이용하여 kernel에게 client가 아닌 server가 sockfd를 이용할 것이라고 알려줌
* 즉 sockfd를 클라이언트의 연결 요청을 수락할 수 있는 listening socket으로 변환
* backlog: 몇개의 connection request를 kernel이 queueing할 수 있는 지
* Server는 accept function을 통해 Client로부터 connection 요청을 기다림
* accept function은 Client와 통신하는 데 사용할 수 있는 connected descriptor(connfd)를 반환
* Client는 connect function을 통해 Server와 Connection 형성
* 연결이 완료되면 Client는 Clientfd를 통해 읽고 쓰고 Server는 Connfd를 통해 읽고 씀

Connected vs Listening Descriptors

* Listening Descriptor
* Client의 Connection Request를 위한 End point
* 한번 생성되고 Server Process의 lifetime 동안 존재
* Connected Descriptor
* Client와 Server사이의 Connection의 Endpoint
* Server가 Client의 연결 요청을 수락할 때마다 새로운 Descriptor가 생성됨
* Client와 Connection이 유지되는 동안에만 존재
* 구분하는 이유
* 여러 Client들과 동시에 통신할 수 있는 Concurrent Server를 위해

//Concurrent Programming

Classical Problem(Concurrent programs)

* Race: Scheduling 순서에 따라 결과가 달라짐
* Deadlock: Resource 할당을 잘못하여 2개이상의 process가 progress를 할 수 없는 것
* Starvation/Fairness/ Livelock: External Event나 System Scheduling Decision이 sub-task가 progress하는 것을 막을 수 있음

Concurrent하지 않은 Server

* Server side TCP manager가 client의 connection request를 queue하기 때문에 connection이 accept되지 않았더라도 client의 connect함수는 return이 됨
* Server side의 TCP manager가 input data를 buffering하기 때문에 rio\_written도 return이 됨

Concurrent Server

* Process Based
* Kernel이 자동적으로 여러 logical flow를 interleaving
* 각 flow는 자신의 own private address space를 가지고 있음
* 각 client는 독립적인 child process에 의해 handle됨
* Listening Server는 반드시 zombie children을 reap해야 함
* Parent process는 반드시 connfd를 close해야함
* fd의 refcnt가 0이 되어야 나중에 connection이 close될 수 있으므로
* 장점
* multiple connection을 concurrently 다룰 수 있음
* Clean sharing model
* 간단하고 직관적
* 단점
* process control을 위한 overhead가 존재
* process간에 data의 공유를 위해 IPC mechanism이 필요
* Event Based
* Programmer가 manually 여러 logical flow를 interleaving
* 모든 flow가 같은 address space를 공유
* I/O multiplexing이라는 기술 사용
* Server는 active connection에 대한 set을 maintain하고 있음
* pending input이 있는 descriptor를 결정하여 이에 대해 처리를 해줌
* listenfd가 pending input이 있으면 connection을 accept해주고 새로운 connfd를 array(set)에 넣어줌
* 장점
* 하나의 logical control flow와 하나의 address space를 가지고 있음
* debugging이 간단함
* process, thread control overhead가 없음
* 단점
* process, thread based design보다 코드가 복잡
* fine grained concurrency가 어려움
* 현재 처리하는 fd에 대한 작업이 끝나기 전까지 다른 fd에 와있는 pending input은 처리할 수 없음
* multi-core의 advantage를 취할 수 없음
* Thread Based
* Kernel이 자동적으로 여러 logical flow를 interleaving
* 각 flow는 같은 address space를 공유
* 각 client는 개개의 peer thread에 의해 handle 됨
* Thread는 TID를 제외한 모든 process state를 공유
* 각 thread는 독립적인 stack을 가지고 있음
* memory leak을 방지하기 위해 detached mode로 수행해야함
* thread가 종료될 때 kernel에 의해 자동적으로 reaping됨
* detached thread는 다른 thread에 의해 reap되거나 kill 될 수 없음
* 의도하지 않은 sharing을 피하기 위해 조심해야 함
* thread에 의해 실행되는 function은 thread safe해야함
* 장점
* thread들 사이에 data structure의 공유 쉬움
* control overhead가 process보다 효율적
* 단점
* 의도하지 않은 data structure의 공유는 미묘하고 재현하기 어려운 오류 유발 가능

Process= Process context(user context, kernel context) + code, data, stack

Process= Thread(Stack, thread context) + code, data, kernel context

Thread

* 각 thread는 자기 자신의 own logical control flow를 가지고 있음
* thread들은 같은 code, data, kernel context를 공유함
* 각 thread는 자기 자신의 own stack을 가지고 있음
* 하지만 다른 thread에 의해 보호받는 것은 아님
* 각 thread는 자기 자신의 own thread id를 가지고 있음
* Concurrent Thread
* flow가 시간상 겹치는 경우 두 thread는 concurrent하게 도는 것임
* Single Core에서는 time slicing을 통해 병렬적으로 도는 것처럼 보임
* Multi Core에서는 실제로 병렬적으로 돌아감.

Thread vs Process

* 공통점
* 각각은 자기 자신의 own logical flow를 가지고 있음
* 각각은 다른 것들과 concurrently 돌아갈 수 있음
* Context Switch
* 차이점
* Thread는 stack을 제외한 code와 data영역을 공유
* Thread는 kernel context만을 공유하고 있기 때문에 process보다 덜 expensive

Shared Variable

* Definition
* 변수 x는 여러 thread가 x의 일부 instance를 참조하는 경우에만 공유 변수

Variable

* Global Variable
* 함수 밖에서 선언 된 변수
* Virtual memory는 어떤 global variable의 한 개 instance만 가질 수 있음
* Local Variable
* static attribute 없이 함수 안에서 선언된 변수
* 각 thread stack은 각 local variable의 한 개 instance를 포함하고 있음
* Local Static Variable
* static attribute 있이 함수 안에서 선언된 변수
* Virtual memory는 어떤 local static variable의 한 개 instance만 가질 수 있음

Process graph

* Process graph
* concurrent thread의 개별 execution state space를 나타냄
* trajectory
* concurrent하게 실행되는 thread들의 가능한 상태 transition의 sequence
* Critical Section
* shared memory를 접근하는 영역으로 명령어들의 Sequence가 Atomic하게 Execute 되어야 하는 영역
* Unsafe Region
* Process graph에서 지나가서는 안되는 영역으로 interleaving이 발생할 수 있는 영역

Mutual Exclusion

* 여러 thread중 한번에 하나의 thread만 critical section을 수행할 수 있도록 하는 것, 어떤 thread가 Critical Section에 들어가 있을 때, 다른 thread들은 Critical Section에 들어가지 못하도록 하는 것

Semaphore

* Synchronization을 위해 사용되는 음수가 아닌 global integer synchronization variable로 mutual exclusion을 강제할 수 있음
* P(s)
* s가 0이 아니면 1을 감소시키고 리턴(atomic한 operation)
* s가 0인 경우, s가 다른 thread의 V operation에 의해 0이 아닐 때 까지 thread를 suspend(누가 자신을 깨우기 전까지는 CPU를 사용하지 않음) 시킴
* V(s)
* s의 값을 1 증가시키고 return
* 만약 P operation에 의해 s가 0이 아니기를 기다리는 다른 thread들이 있다면, 그 중 하나의 thread들을 깨워줌
* Binary Semaphore
* 0과 1값만을 가지는 semaphore
* Mutex
* mutual exclusion을 위해 사용되는 binary semaphore
* Counting Semaphore
* Resource의 상태를 Tracking하고 다른 Thread들에게 알리기 위해 사용하는 semaphore

Reader Writers Problem

* First Reader-Writers Problem
* Reader에게 우선순위
* Waiting Writer 뒤에 도착한 Reader는 기다리는 Writer보다 더 높은 우선순위
* Second Reader-Writers Problem
* Writer에게 우선순위
* Writer 뒤에 오는 Reader는 반드시 기다려야 함, Writer가 waiting 중이더라도
* <https://gist.github.com/melouver/fee6365edc3fc74927b5305f48a055f6>
* 두 경우다 Starvation이 생길 수 있음

Prethreaded Concurrent Server

* 계속해서 Thread\_Create, Free하는 것은 OS에 무리이므로 Worker Thread Pool을 만들고 일이 끝난 Worker들이 buffer로부터 fd를 빼서 수행

Thread Safety

* thread로부터 호출되는 함수들은 반드시 thread safe해야함
* Thread Safe Function: 여러 concurrent thread들로부터 반복적으로 호출되더라도 항상 Correct한 Result를 내는 함수
* Class of thread Unsafe Function
* Class1: shared variable을 보호하지 않는 함수
* 해결 방법: Semaphore Operation을 사용하여 Shared Variable을 보호
* Synchronization은 코드의 속도를 느리게 할 수 있음
* Class2: 호출될 때마다 상태를 기록해야하는 함수
* 이전 상태를 기반으로 다음 상태를 만들어 내는데 다른 thread에서 상태를 바꾸어버리면 문제가 생김
* 해결 방법: State를 Global하게 기록하지 않고, State를 Local Stack에 기록(reentrant function으로 만들어 줌)
* Class3: static variable의 pointer를 반환하는 함수
* static variable의 값을 바꾸고 pointer를 반환했는데, 해당 pointer를 통해 바꾼 값을 읽기 전에 다른 thread에서 static variable의 값을 바꾸어 버리면 바뀐 값을 읽게 됨
* 해결방법
* Caller가 자신의 Local 변수의 주소를 전달하여 결과를 Local 변수에 값을 저장하도록 함수를 다시 작성
* Lock을 먼저하고 함수를 호출 한 뒤 pointer로 값에 접근하여 값을 Local Stack에 Copy해준 이후 Lock을 풀어 줌(Synchronization 해버림)
* Class4: thread-unsafe function을 호출하는 함수
* 해결방법: thread safe한 함수 만을 호출하도록 함수를 수정

Reentrant Function

* 정의
* 여러 thread에 의해 호출되는 함수가 shared variable을 접근하지 않는 것
* 특징
* Synchronization이 필요 없음
* Class2 function같은 경우 reentrant로 만드는 것밖에 해결방법이 없음
* Standard C Library 안의 모든 함수는 thread-safe
* 대부분의 Unix System Call은 Thread Safe

Race

* Thread들의 수행 순서를 보장하지 못하는데, Program의 Correctness가 Thread들의 수행 순서에 의존하는 경우

Deadlock

* 절대 참이 될 수 없는 조건을 기다리고 있는 경우(thread들이 서로가 각 조건들을 참으로 만들어주길 기다리는 상태)
* Thread A는 Thread B가 어떤 조건을 만족해줘야 진행할 수 있는데, 동시에 Thread B는 Thread A가 어떤 조건을 만족해줘야 진행할 수 있는 경우(두 Thread는 영원히 서로를 기다리게 됨)