컴퓨터학부, 20190511, 배준형, 비밀번호 (qwg724)

# 과제 개요

|  |
| --- |
| * xv6는 MIT에서 x86기반 RISK 시스템을 개발하기 위해 만든 교육용 OS 프로젝트이다. * xv6는 기본적으로 1tcks 마다 스케쥴링되는 RR 스케쥴러, 12개의 직접 매핑 + 1개의 2-level multilevel 페이징 블록을 사용하는 구조로 되어있다. * xv6를 ubuntu에서 실행하기위해 linux의 가상화 시스템 (kvm) 을 qemu 에뮬레이터를 실행하기 위해 여러 패키지를 install 해주어야 한다. * 이번 과제 목표는 xv6를 설치하여 간단한 “hello xv6 world” 출력을 하는 것과 마치 cat.c 와 같이 htac.c 를 이용하여 특정 파일을 읽어 라인수 만큼 반대로 출력하는 htac.c를 구현하는 것을 목표로한다. |

# 상세설계

|  |  |  |
| --- | --- | --- |
| 1. **xv6 다운로드 및 설치**  * Makefile 을 실행하기 위하여 **build-essential 패키지를 설치**하였다.   gcc apt-get install build-essential   * xv6는 기본적으로 **크로스 컴파일** 작업을 해주어야한다. 그렇기 때문에 cross compile을 지원해주는   **gcc-mulltilib 패키지**를 linux에 다운로드 받아주어야한다.  ☞ cross compile : 컴파일러가 실행하는 플랫폼이 아닌 다른 플랫폼에서 실행가능한 코드를 생성하는 컴파일 과정을 의미.  gcc apt-get install gcc-multilib   * xv6는 qemu에뮬레이터 (가상화 솔루션) 위해서 작동하기 때문에 qemu 에뮬레이터를 설치해주어야 한다. KVM 은 (Kernel-based Virtual Machine) 리눅스 커널 자체가 가상화 기능을 제공하고 있는데 이때 사용하는 모듈이 KVM이므로 **qemu-kvm** 패키지로 설치한다. * ‘**가상화** (Hypervisor)’ 라는 것은 해당 프로세스가 마치 자신의 가상공간 (커널 공간, 자원분배) 을 가지면서 (메모리 가상화) CPU 스케쥴링을 통해 Context switch(문맥교환)을 하여 프로세스마다 CPU를 점유할 수 있게 하여 독자적인 환경을 구성해주는 것이다. * ‘**에뮬레이터**’ 란, H/W를 S/W 적으로 구현하여 실행환경을 구축하는 것을 의미하고 여기서 QEMU 가 해당한다.   (<https://blog.naver.com/alice_k106/221179347223> 참고)  gcc apt-get install qemu-kvm   * xv6를 **github**에서 받아오기 위해 Linux 가상머신 (ubuntu) 에 **git을 설치한다**. (+ Token 등록)   sudo apt install git-all   * [xv6 실행] : * xv6 내부파일을 컴파일 하기 위하여 make 한다. * Makefile 을 이용하여 qemu 에뮬레이터로 (kvm이용) 커널 부팅을 시도하기 위해 make-qemu 를 한다. * 만약 그래픽 창이 필요가 없다면 qemu-nox로 실행해도 된다. * 디버깅 과정을 이용하려면 qemu-gdb를 이용할 수도 있다.   [xv6의 Makefile 내부에 존재하는 Makefile 옵션들..]   1. **helloxv6 설계**  * printf 를 표준출력하여 파일로 출력하도록 프로그램을 설계. * printf.c 에 선언된 프로토타입은 void printf(int fd, const char\* fmt, ...) 으로 파일디스크립터를 반드시 삽입하여야 하므로, STDOUT\_FILENO (=1) 을 첫 번째 파라미터로 넘겨주도록 설계. * 해당 파일에서는 main() 함수만 호출하여 printf.c의 printf 함수를 호출하는 방식.  1. **htac.c 설계**  * main 내부에서 htac 함수에 file descriptor 인자를 넘겨주어 원하는 파일 내용을 모두 읽어 간이적으로 만든 Stack (연결리스트기반) 을 이용하여 ‘\n’ 단위로 스택을 구성 (마지막 공백 문자포함) * 형성된 Stack을 바탕으로 main.c에 선언된 line에 저장된 line 수 만큼 파일을 역으로 출력함. * 만약, 파일의 라인 수가 32줄인데 line에 1,000 줄 등 라인 수보다 많은 수가 선언되어있으면 파일의   끝까지 내용을 역으로 출력하고 htac를 종료하도록 설계   * htac의 Stack은 동적할당을 통해 생성하므로 (Dummy Node 포함) htac 마지막 라인블록에 free(ptr)을 이용하여 할당해제 * htac는 연속으로 할당할 수 있도록 설계되어있다. (단, <line> “파일명” 구조를 연속적으로 지켜줘야함)   (실행결과에서 실행 예시를 볼 수 있음)   * **xv6의 다른 .txt파일을 넘겨주기위해 test\_file.txt를 작성하고 Makefile 에서 xv6 파일시스템 image파일에 등록한다.**  |  | | --- | | **<Makefile부분>**  **fs.img**: mkfs README **test\_file.txt** $(UPROGS)  ./mkfs fs.img README **test\_file.txt** $(UPROGS)  **PRINT** = runoff.list runoff.spec README toc.hdr toc.ftr **test\_file.txt** $(FILES) |  * htac 함수 프로토타입 void htac (int fd)  |  | | --- | | htac 함수 호출 : cat를 전역변수 line만큼 반대로 호출  @param fd : File Descriptor  @return 없음 (void형태) |     [htac.c Call Graph]   1. **Makefile.**  * 사용자 정의 파일을 xv6 내부로 전달하기 위해 EXTRA 변수에 htac.c 와 helloxv6.c 파일을 정의함. * 사용자가 정의한 파일 (test\_file.txt)을 xv6로 넘겨서 커널 명령어로사용할 수 있게하기 위해서 UPROGS 에 추가적으로 정의함. (\_htac, \_helloxv6) |

# 결과

## xv6 다운로드 및 설치

|  |
| --- |
|  |

## helloxv6.c

|  |
| --- |
| * helloxv6를 makefile에 등록하여 바로 출력하여 나온다. |

## htac.c

|  |
| --- |
| # htac 5 README   * README 의 마지막 라인은 \n으로 끝나지 않아도 공백라인 포함 출력을 지원하는 모습을 보여주고 있다. * htac는 README 의 마지막 라인부터 역으로 출력하는 모습을 볼 수 있다.     #htac 10줄을 시도한 모습   * 공백라인을 포함하여 10줄을 출력하는 모습을 보여주고 있다.     #htac 1000줄을 시도한 모습 (가장 마지막 라인에 도달하면 출력을 종료하고 htac를 종료하도록 설계)   * README 는 총 51줄로 1,000줄 보다 짧은 파일이다. 1,000줄을 시도한다면 반대로 51줄까지만 출력하고 htac가 종료하도록 설게되었다.     #없는 파일의 경우   * open 에 실패하여 에러 메시지를 출력한다.     # 다른 파일 test\_file.txt 를 출력한 모습 (다른 파일도 xv6 가상커널에 등록하면 사용할 수 있다.)   * makefile     # htac를 연속으로 시도하는 경우   * htac 10 README 5 test\_file.txt와 같이 <n1> 파일1 <n2> 파일2 ... 등으로 명령어를 실행한다면   연속으로 뒷 파일을 역순으로 출력하는 모습을 볼 수 있다. |

# 소스코드

## helloxv6.c

|  |
| --- |
| #include "types.h"  #include "stat.h"  #include "user.h"    int main(int argc, char\* argv[])  {  //표준 입출력  printf(1, "Hello xv6 World\n");  exit();  } |

## htac.c

|  |
| --- |
| #include "types.h"  #include "stat.h"  #include "user.h"  #include "fcntl.h"  #ifndef NULL  #define NULL 0  #endif  #define BUFSIZE 512  char buf[512];  int line;  //Stack용 구조체  typedef struct stack {  char strs[BUFSIZE];  struct stack \*next;  }Stack;  //#define DEBUG  /\*\*  \* htac 함수 호출 : cat를 전역변수 line만큼 반대로 호출  \* @param fd : File Descriptor  \* @return 없음  \*/  void  htac(int fd)  {  int n;  Stack \*head = (Stack\*)malloc(sizeof(Stack)); //Stack 선언부  head->next = NULL;  Stack \*ptr = head, \*tmp; // Stack 포인터  char \*remember = NULL, \*last = NULL; // 이전 문자열 기억하는 임시변수  char tmp\_line[BUFSIZE] = {0,}; // read간의 \n 단위로 짤린 문자열 처리  while((n = read(fd, buf, BUFSIZE-1)) > 0) {  #ifdef DEBUG  printf(1, "%s", buf);  #endif  remember = buf;  for (char\* s = buf ; \*s != '\0' ; s++) {  //개행 문자 단위로 Stack에 요소 추가  if (\*s == '\n') {  \*s = '\0'; //문자열 토큰화  Stack \*newS = (Stack\*)malloc(sizeof(Stack));  char\* s\_ptr = newS->strs;  //짤린 문자열 있으면 짤린 문자열 처리  if (strlen(tmp\_line) != 0) {  strcpy(s\_ptr, tmp\_line);  s\_ptr += strlen(s\_ptr);  memset(tmp\_line, 0, 512);  }  //Stack에 문자열 연결  strcpy(s\_ptr, remember);  newS->next = ptr; //스택 연결  ptr = newS;  #ifdef DEBUG  printf(1, "%s\n", remember);  printf(1, "debug : %s\n", newS->strs);  #endif  //remember 갱신  remember = s+1;  }  }  // 마지막에 개행문자가 없어 remeber가 잔여문자열이 남으면 해당 문자열 tmp\_line으로 전달  if (strlen(remember) > 0) {  last = remember + strlen(remember) - 1;  if (\*last != '\n') {  strcpy(tmp\_line, remember);  }  }  //쓰레기 코드 방지용 초기화  memset(buf, 0, 512);  }  //혹시 끝 문자가 개행으로 끝나지 않아 잔여 문자열이 남은 경우 스택 연결  if (strlen(tmp\_line) != 0) {  Stack \*last\_line = (Stack\*)malloc(sizeof(Stack));  strcpy(last\_line->strs, tmp\_line);  memset(tmp\_line, 0, 512);  last\_line->next = ptr; //스택 연결  ptr = last\_line;  }    //Stack 순회하며 line만큼 출력  for (n = 0, tmp = ptr ; n < line ; n++, tmp = tmp->next) {  if (tmp != head)  printf(1, "%s\n", tmp->strs);  else  break;  }  //Scack 할당해제 함수  for (tmp = ptr ; tmp != NULL ;) {  ptr = tmp; //임시저장  tmp = tmp->next; //  free(ptr);  }  }  int  main(int argc, char\* argv[])  {  int fd, i;  // argc가 부족한 경우 에러처리  if (argc <= 2) {  printf(2, "usage: %s <n> <FILE\_NAME>\n", argv[0]); //표준 에러 처리  exit();  }  // 연속 htac 지원을 위한 for문처리  for (i = 1 ; i < argc ; i+=2) {  line = atoi(argv[i]); //전역변수 line을 이용한 문자열처리  //에러처리  if ((fd = open(argv[i+1], O\_RDONLY)) < 0) {  printf(1, "htac: cannot open %s\n", argv[i+1]);  exit();  }  htac(fd); // htac함수호출  close(fd); // 파일 디스크립터 해제  }  exit();  } |