컴퓨터학부, 20190511, 배준형

# 과제 개요

|  |
| --- |
| 1. **시스템 콜 구현--** 2. OS는 기본적으로 제한적 직접 실행 (‘limited direct execute)을 추구한다. 그렇기에 OS와 같이 다른 프로세스와의 독립성을 지켜 줘야하는 민감한 작업등은 **커널모드 (kernel mode)** 에서 시행되며, 민감하지 않은 작업들 (일반 프로세스) 등은 **유저모드 (user mode)** 에서 실행된다. ~~User mode 에서 Kernel mode 로 전환하기 위해서는 프로세스는 H/W 에서 지원하는 trap 명령어를 호출하여야한다.~~ 3. ~~trap 명령어를 호출하기 위해서 프로세스는 ‘시스템 콜’ 을 호출함으로써 OS를 호출하여 OS가 kernel mode로 진입하여 민감한 작업등을 수행할 수 있다. OS는 이~~ **~~시스템콜~~** ~~을 일종의 trap table로 전달받게 된다. xv6에서는 syscall.h 에~~ **~~시스템콜 핸들러~~**~~를 실행시켜 진행되고있음을 볼 수 있다.~~ (수업 듣고 수정..) 4. Interrupt는 Timer Interrupt 나 Disk Interrupt와 같이 H/W에서 PIC로 전달되는 인터럽트를 전달하여 프로세스도 모르는 사이에 **비동기 호출**방식인 **H/W 인터럽트**와, 실행 프로세스 내부에서 초래하여 **동기방식**으로 OS(커널) 으로 처리되는 **S/W 인터럽트**가 있다. 5. S/W 인터럽트 중 **비명시적으로 호출**되는 trap 명령어는 Divide\_Zero 나 swap공간에 데이터블록이 있어 메모리에서 페이지를 바로 사용할 수 없는 상태인 Page-Fault 오류가 있다. 6. **그리고 명시적으로 호출** 되는 **시스템콜 (Supervisor Call)** 은 int 0x80 #(xv6에선 0x40) 와 movl 시스템콜넘버 %eax으로 user Mode에서 직접 인터럽트를 걸어 부팅 시 물리메모리 하위번지에 올려진 IDT(IVT=Interrupt Vector Table) 의 0x80번지 값으로 system call 인터럽트를 호출하여 전달받은 범용레지스터 eax로부터 시스템 콜번호를 받아 sys\_call\_table에서 시스템 콜 핸들러를 실행시키는 방식으로 시스템 콜이 전달된다. 7. xv6에선 UNIX에서 Interrupt를 관장하는 entry.S 의 역할을 하는 파일인 trapasm.S 와 trap.c 로부터 Interrupt를 검사하여 처리하게 되며, 시스템 콜 테이블에서 시스템 콜을 찾아 핸들러를 실행시키는 syscall.c 의 syscall 함수가 처리한다. 이러한 과정 등을 대략적으로 요약하여 이번 과제에서 수정하여야하는 파일들은 아래와 같다. 8. user.S : 위의 movl 시스템 콜 번호 %eax 와 int 0x40를 처리하는데 시스템 콜 번호를 등록하기 위해 사용되는 파일이다. 아래에 시스템 콜을 추가하여야 user mode에서 프로세스가 시스템 콜을 전달할 수 있게된다. 9. user.h : Process에서 시스템 콜을 사용하기 위해 정의하는 header 부분이다. 10. syscall.h : syscall.c에서 시스템 콜 테입블에서 해당되는 인덱스를 정의한 시스템 콜 헤더이다. 11. syscall.c : 시스템 콜 핸들러를 실행시켜줄 시스템콜 핸들러 호출 함수 syscalls에 시스템 콜을 추가하고, 실제로 만든 핸들러를 다른 파일에 있으므로 extern 으로 연결해준다. 12. sysfile.c : 제작한 시스템 콜 내부를 구현한 파일이다. (꼭 해당파일일 필요는 없으나, 규칙성을 따름)   <시스템 콜 이외의 proc을 건들인 부분>  > alarm을 구현하기 위해 PCB 마다 alarmticks와 alam\_timer를 추가정의함.  > trap.c에서 시스템 콜이 호출될 때마다 인터럽트 핸들러로 분기할 때 해당 파일에서 alarm이 되도록 구현. |

# 상세설계

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. 공통 구현  * user.S   시스템 호출 리스트를 저장한다. 해당 파일에서 assemble 어로 적용하여 시스템콜 함수를 만들어준다.  해당 파일에서 user모드를 kernel 모드로 격상하도록 interrupt 를 호출하는 것을 볼 수 있다.  (int $T\_SYSCALL)  //추가  SYSCALL(date)  SYSCALL(alarm)   * sysfile.c  1. **int sys\_date (void)**  * 시스템콜 핸들러는 인자를 받지 않는 void 형태로 테이블로 전달하기 때문에 syscall.c의 argint, argptr 등을 이용하여 넘겨받은 인자를 사용할 수 있다. argint 함수는 내부적으로 fetchint를 호출하여 현재프로세스의 시스템콜을 호출하기 전에 user 프로세스의 정보를 **esp** 레지스터에 **INT\_SIZE + INT\_SIZE\*파라미터 번호** 를 더해 스택주솟값을 바탕으로 데이터를 받아온다. * **argptr**은 내부적으로 argint (32비트 주소값이기 때문) 로 주솟값을 받아와 변환시켜준다.   **argptr의 3번째인자**는 argptr 값이 size를 벗어나는지 미리 체크를 해주는역할로 사용된다.   * argptr로 인자를 넘겨받은 후 lapic.c에 정의된 cmostime 을 호출하여 rtcdate 구조체를 현재시간으로 정리한다 UNIX의 localtime으로 struct tm을 초기화하는 것과 비슷한 맥락이다. (time() + localtime() 모두 해줌) * **만약 argptr 함수가 호출이 실패하여 시스템콜로 전달한 인자를 사용할 수 없는 경우 -1을 리턴하고 시스템콜이 종료된다.** * 성공 시 0을 리턴한다.  1. **int sys\_alarm(void)**  * argint로 user process로부터 timer 변수를 넘겨받는다. * **사용자는 int 형태로 값을 넘긴다. 하지만 내부적으로 uint**값으로 전환하며, 사용자가 넘겨준 timer가 음수인 경우 -1을 리턴한다. (**proc.h에서 alarm\_timer는 0xFFFFFFFF로 설정되어있으며 사용자는 초기에 음수값을 전달할 수 없기 때문에 0xF??????? 시간은 모두 받을 수 없다! --> 즉, 사용자가 넘겨줄 수 없는 변수로 초기화되었기 때문에 안정적이다**. ) * 내부에서 현 프로세스의 alarm\_timer 를 초기화해준다. * **xv6 의 1 ticks 는 대략 10ms 이므로 tick\_convert로 100을 곱해줘서 timer(sec) 초 단위로 변환해준다.** * syscall.c * OS가 시스템 콜을 호출할 수 있도록 시스템콜을 extern 화 시켜준 후 시스템 콜 trap table을 초기화한다.   **<전역변수 중..>**   * extern int sys\_date(void); * extern int sys\_alarm(void);   **<시스템콜 trap table중 ...>**   * static int (\*syscalls[])(void) = { ... * [SYS\_date] sys\_date, * [SYS\_alarm] sys\_alarm, * } * syscall.h * 시스템 콜 entry 인덱스 등록을 해준다.   <시스템 콜 define 부분 중 ...>   * #define SYS\_date 22 * #define SYS\_alarm 23 * user.h * 사용자 process 내부에서 시스템 콜 호출을 할 수 있도록 함수를 정의한다.  1. int date (struct rtcdate\* r) 구현   date 함수 **프로토타입** void htac (int fd)   |  | | --- | | date 시스템 콜 호출 : 인자로 전달받은 r을 현재시간의 정보를 담은 date.h time 추상화 구조체 rtcdate 로 전달.  @param struct rtcdate \*r : date.h 에 정의된 date 추상화 구조체  @return 성공시0, 에러시-1 (인자로 전달받은 rt가 실패한경우) |  1. int alarm (int timer) 구현   alarm 함수 **프로토타입** int alarm (int timer)   |  | | --- | | alarm 시스템 콜 호출 : 프로세스를 timer 후에 종료시키는 시스템콜  @param struct rtcdate \*r : date.h 에 정의된 date 추상화 구조체  @return 성공시0, 에러시-1 (인자로 전달받은 rt가 실패한경우) |  * proc.h 수정 * PCB 블록 (proc 구조체) 내부에 멤버변수 추가 * uint alarmticks;   : 초기에 0으로 설정해뒀다가, alarm\_timer 가 0xFFFFFFFF 가 아닐 시 tick 이 증가하도록 설계   * uint alarm\_timer;   : 초기에 0xFFFFFFFF 로 설정해뒀다가 시스템콜 alarm이 호출 시 해당 timer로 초기화   * **초기화를 0xFFFFFFFF로 한 이유 :** 시스템콜 내부에서 **timer를 받을 때는 int (즉 정수형)** 으로 받는다. 반면 멤버변수 **alarm\_timer는 uint**로 MSB에 상관이 없다. 더군다나 시스템 콜에 alarm(-3) 처럼 timer가 음수인 경우 (MSB가 1즉, 0xF???????, ?는 0~9 wildcard) 는 시스템콜을 에러처리하도록 설계하였기 때문에 사용자 입력으로 alarm\_timer가 0xFFFFFFFF에 도달할 수 있는 경우는 존재하지 않으므로 초기화할 수 있는 값으로 적당하다. (사용자 입력은 0 ~ 0xEFFFFFFF 까지 가능) * proc.c 수정 * allocproc() 함수 내부에서 alarm\_timer와 alarmticks 를 초기화한다. (0xFFFFFFFF와 0) * trap.c 수정 * void trap(struct trapframe \*tf) 함수내부 수정 * 현 프로세스 alarm\_timer 가 설정되었을 시   (0xFFFFFFFF가 아닐 시) alarm tick 증가 시작 추가   |  | | --- | | if (myproc() && myproc()->alarm\_timer != 0xFFFFFFFF)  myproc()->alarmticks++; |  * 프로세스 alarm\_timer 가 0xFFFFFFFF 가 아니고 process tick alarmticks 가 alarm\_timer를 넘어선 경우   프로세스 종료하도록 수정 (SSU\_Alarm 출력 + 시간 출력 추가)   |  | | --- | | //프로세스 alarm\_timer 설정이 되었다면 timer() check  if (myproc() && myproc()->alarm\_timer != 0xFFFFFFFF) {  if (myproc()->alarmticks >= myproc()->alarm\_timer) { //timer 초과여부 확인  cprintf("SSU\_Alarm!\n");  #ifdef P3\_TIMER  //과제 명세에 종료 전 출력부분이 있어 수정  struct rtcdate r;  cmostime(&r);  cprintf("Current time : %d-%d-%d %d:%d:%d\n", r.year, r.month, r.day, r.hour, r.minute, r.second);  #endif  exit(); //프로세스 종료  }  } |  * **Call Graph 전체모습. (Kernel모드와 UserMode 호출그림)**     [이번과제 전체적인 Call Graph.]   * (너무 커서 커널 부분만 세부적으로 다시 첨부)     [Kernel Mode 부분]   1. **Makefile.**  * 사용자 정의 파일을 xv6 내부로 전달하기 위해 EXTRA 변수에 datetest.c 와 alarm\_test.c 파일을 정의함. * 사용자가 정의한 파일 (test\_file.txt)을 xv6로 넘겨서 커널 명령어로사용할 수 있게하기 위해서 UPROGS 에 추가적으로 정의함. (\_datetest, \_alarm\_test) * (클래스룸 피드백 반영) xv6는 기본적으로 utc를 사용하기 때문에 Makefile 내부의 qemu 뒤에 -rtc base=localtime 를 넣어 컴퓨터 시간을 기준으로 변경함. |

# 결과

## **date 시스템콜**

|  |
| --- |
| * datetest 프로그램을 실행하였을 때 현재 시간이 출력되는 점을 볼 수 있다.   printf(1, “format”) 의 format 을 별도로 지정하지 않아 칸수가 정확하게 맞춰지진 않는 점을 볼 수 있다. |

## **alarm 시스템 콜**

|  |
| --- |
| * alarm 을 3초로 진행      * 시스템콜이 종료될 때 종료된 시간이 호출되는 모습으로 몇 초 후 종료되었는지 알 수 있다. * alarm 을 10초로 진행      * alarm을 10초로 넣었으나 11초가 지나 종료됨을 볼 수 있다. 이는 다음의 요인들이 영향을 미친 것 같다. * A. alarm 을 시작하기 전 time 을 찍고 System Call을 호출하여 시스템 콜이 처리되기까지 시간 + * 시스템 콜이 종료될 때 시스템콜이 **Timer**가 지났음을 인지하고 trap.c에서 cprint 를 찍기까지의 시간등이 용인으로 작용한 듯 하다. * alarm을 0초로 설정      * 원래 UNIX alarm과 다르게 alarm을 0초로 설정해두면 프로세스가 즉시 종료되도록 커스터마이징 해보았다. 실제로 적용해보니, 프로세스에서 printf가 앞에 있음에도 출력하기까지의 시간 전에 프로세스가 종료하여 프로세스 출력이 되지 않은 것을 볼 수 있다. * alarm 에 시간을 설정하지 않은 경우 * alarmtest 코드 중 argc 가 1보다 작으면 종료되도록 설정되어있다. 이를 통해 즉시 종료됨을 볼 수 있다. |

# 소스코드

### datetest.c

|  |
| --- |
| 1. #include "types.h" 2. #include "user.h" 3. #include "date.h" 5. int main(void) 6. { 7. struct rtcdate r; 8. date(&r); //시스템 콜 호출 9. printf(1, "Current time : %d-%d-%d %d:%d:%d\n", 10. r.year, r.month, r.day, r.hour, r.minute, r.second); //현재시간 출력 11. exit(); 12. } |

### alarm\_test.c

|  |
| --- |
| 1. #include "types.h" 2. #include "user.h" 3. #include "date.h" 5. int main(int argc, char\* argv[]) 6. { 7. int seconds; 8. struct rtcdate r; 10. //오류 처리 11. if (argc <= 1) 12. exit(); 14. seconds = atoi(argv[1]); // 시간 인자받기 15. alarm(seconds); //alarm 설정 17. date(&r); 18. printf(1, "SSU\_Alarm Start\n"); //시작 출력 19. printf(1, "Current time : %d-%d-%d %d:%d:%d\n", 20. r.year, r.month, r.day, r.hour, r.minute, r.second); //현재 시간 출력 21. while(1) 22. ; 23. exit(); 24. } |

### proc.c

|  |
| --- |
| 1. #include "types.h" 2. #include "defs.h" 3. #include "param.h" 4. #include "memlayout.h" 5. #include "mmu.h" 6. #include "x86.h" 7. #include "proc.h" 8. #include "spinlock.h" 9. struct { 10. struct spinlock lock; 11. struct proc proc[NPROC]; 12. } ptable; 13. static struct proc \*initproc; 14. int nextpid = 1; 15. extern void forkret(void); 16. extern void trapret(void); 17. static void wakeup1(void \*chan); 18. void 19. pinit(void) 20. { 21. initlock(&ptable.lock, "ptable"); 22. } 23. // Must be called with interrupts disabled 24. /\*\* 25. \* cpuid == 현재 실행되는 CPU 번호 (0번이면 0, 1번이면1...) 26. \* @return cpu 테이블에서 현재 cpu index 27. \* @warning proc.c 28. \*/ 29. int 30. cpuid() { 31. return mycpu()-cpus; 32. } 33. // Must be called with interrup disabled to avoid the caller being 34. // rescheduled between reading lapicid and running through the loop. 35. /\*\* 36. \* 현재 실행중인 CPU가 유효한지 판단 후 CPU 구조체 (추상화) 리턴 37. \* @return cpu\* 추상화 구조체 38. \* @warning proc.c 39. \*/ 40. struct cpu\* 41. mycpu(void) 42. { 43. int apicid, i; 45. if(readeflags()&FL\_IF) 46. panic("mycpu called with interrupts enabled\n"); 48. apicid = lapicid(); 49. // APIC IDs are not guaranteed to be contiguous. Maybe we should have 50. // a reverse map, or reserve a register to store &cpus[i]. 51. for (i = 0; i < ncpu; ++i) { 52. if (cpus[i].apicid == apicid) 53. return &cpus[i]; 54. } 55. panic("unknown apicid\n"); 56. } 57. // Disable interrupts so that we are not rescheduled 58. // while reading proc from the cpu structure 59. /\*\* 현재 실행중인 프로세스 리턴 60. \* @return 현재 실행중인 프로세스 구조체 proc\* 리턴 61. \* @warning proc.c 62. \*/ 63. struct proc\* 64. myproc(void) { 65. struct cpu \*c; 66. struct proc \*p; 67. pushcli(); 68. c = mycpu(); 69. p = c->proc; 70. popcli(); 71. return p; 72. } 73. //PAGEBREAK: 32 74. // Look in the process table for an UNUSED proc. 75. // If found, change state to EMBRYO and initialize 76. // state required to run in the kernel. 77. // Otherwise return 0. 78. /\*\* 79. \* process 가 새로 할당될 때마다 호출 80. \* 프로세스 할당 후 멤버변수들을 초기화하는 부분 존재 81. \* 프로세스의 커널 스택을 할당하고 초기화. 82. \* @return 프로세스를 하나 할당하고 리턴한 proc\* 구조체 (실패시 NULL = 0) 83. \* @warning proc.c 84. \*/ 85. static struct proc\* 86. allocproc(void) 87. { 88. struct proc \*p; 89. char \*sp; 90. acquire(&ptable.lock); 91. //프로세스 중 UNUSED 프로세스 찾으면 found로 이동 92. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) 93. if(p->state == UNUSED) 94. goto found; 95. release(&ptable.lock); 96. return 0; 97. found: 98. //프로세스 커널 Runnable 전 초기 사애는 EMBRYO로 설정 99. p->state = EMBRYO; 100. p->pid = nextpid++; 101. release(&ptable.lock); 102. // 프로세스 커널 스택할당 103. if((p->kstack = kalloc()) == 0){ 104. p->state = UNUSED; 105. return 0; 106. } 107. //커널 스택 프레임만큼 sp 증가.. 108. sp = p->kstack + KSTACKSIZE; 109. //커널 스택에 trapframe 을 둠. 110. sp -= sizeof \*p->tf; 111. p->tf = (struct trapframe\*)sp; 112. // Set up new context to start executing at forkret, 113. // which returns to trapret. 114. sp -= 4; 115. \*(uint\*)sp = (uint)trapret; 116. sp -= sizeof \*p->context; 117. p->context = (struct context\*)sp; 118. memset(p->context, 0, sizeof \*p->context); 119. p->context->eip = (uint)forkret; 120. //#P2 설계과제 ticks 설정 --> 둘 다 초기값이 0이므로 trap.c에서 잘 조정할 것 121. p->alarm\_timer = 0xFFFFFFFF; 122. p->alarmticks = 0; 123. return p; 124. } 125. //PAGEBREAK: 32 126. // Set up first user process. 127. /\*\* 128. \* 초기 프로세스할당 시 사용 (프로세스이름 : initcode) 129. \* @return 없음 130. \* @warning xv6 초기실행시 실행되는 구문 --> sh을 돌려주는(init.c) 프로세스로 추측됨 131. \*/ 132. void 133. userinit(void) 134. { 135. struct proc \*p; 136. extern char \_binary\_initcode\_start[], \_binary\_initcode\_size[]; 137. p = allocproc(); 139. initproc = p; 140. if((p->pgdir = setupkvm()) == 0) 141. panic("userinit: out of memory?"); 142. inituvm(p->pgdir, \_binary\_initcode\_start, (int)\_binary\_initcode\_size); 143. p->sz = PGSIZE; 144. memset(p->tf, 0, sizeof(\*p->tf)); 145. p->tf->cs = (SEG\_UCODE << 3) | DPL\_USER; 146. p->tf->ds = (SEG\_UDATA << 3) | DPL\_USER; 147. p->tf->es = p->tf->ds; 148. p->tf->ss = p->tf->ds; 149. p->tf->eflags = FL\_IF; 150. p->tf->esp = PGSIZE; 151. p->tf->eip = 0; // beginning of initcode.S 152. safestrcpy(p->name, "initcode", sizeof(p->name)); 153. p->cwd = namei("/"); 154. // this assignment to p->state lets other cores 155. // run this process. the acquire forces the above 156. // writes to be visible, and the lock is also needed 157. // because the assignment might not be atomic 158. acquire(&ptable.lock); 159. p->state = RUNNABLE; 160. release(&ptable.lock); 161. } 162. // Grow current process's memory by n bytes. 163. // Return 0 on success, -1 on failure. 164. /\*\* 165. \* 현재 프로세스의 메모리 공간을 늘리거나 키움 166. \* @param n : 양수(키움), 음수(줄임) 167. \* @return 성공시 0 에러시 -1 168. \*/ 169. int 170. growproc(int n) 171. { 172. uint sz; 173. struct proc \*curproc = myproc(); 174. sz = curproc->sz; 175. if(n > 0){ 176. if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0) 177. return -1; 178. } else if(n < 0){ 179. if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0) 180. return -1; 181. } 182. curproc->sz = sz; 183. switchuvm(curproc); 184. return 0; 185. } 186. // Create a new process copying p as the parent. 187. // Sets up stack to return as if from system call. 188. // Caller must set state of returned proc to RUNNABLE. 189. /\*\* 190. \* 프로세스 fork 함수 191. \*/ 192. int 193. fork(void) 194. { 195. int i, pid; 196. struct proc \*np; 197. struct proc \*curproc = myproc(); 198. // Allocate process. 199. if((np = allocproc()) == 0){ 200. return -1; 201. } 202. // Copy process state from proc. 203. // 부모 mm\* 복사 204. if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){ 205. kfree(np->kstack); 206. np->kstack = 0; 207. np->state = UNUSED; 208. return -1; 209. } 210. np->sz = curproc->sz; 211. np->parent = curproc; 212. \*np->tf = \*curproc->tf; 213. // Clear %eax so that fork returns 0 in the child. 214. np->tf->eax = 0; 215. for(i = 0; i < NOFILE; i++) 216. if(curproc->ofile[i]) 217. np->ofile[i] = filedup(curproc->ofile[i]); 218. np->cwd = idup(curproc->cwd); 219. safestrcpy(np->name, curproc->name, sizeof(curproc->name)); 220. pid = np->pid; 221. acquire(&ptable.lock); 222. np->state = RUNNABLE; 223. release(&ptable.lock); 224. return pid; 225. } 226. // Exit the current process. Does not return. 227. // An exited process remains in the zombie state 228. // until its parent calls wait() to find out it exited. 229. /\*\* 230. \* @warning proc.c 231. \*/ 232. void 233. exit(void) 234. { 235. struct proc \*curproc = myproc(); 236. struct proc \*p; 237. int fd; 238. if(curproc == initproc) 239. panic("init exiting"); 240. // 모든 파일 디스크립터 + openfile table을 종료시킴 241. for(fd = 0; fd < NOFILE; fd++){ 242. if(curproc->ofile[fd]){ 243. fileclose(curproc->ofile[fd]); 244. curproc->ofile[fd] = 0; 245. } 246. } 247. begin\_op(); //로그 작성시작 248. iput(curproc->cwd); //현재프로세스 부모위치의 i-node 이 ref-cnt 를 하나 줄임 249. end\_op(); //log 작성 250. curproc->cwd = 0; 251. acquire(&ptable.lock); 252. // Parent might be sleeping in wait(). 253. wakeup1(curproc->parent); 254. // Pass abandoned children to init. 255. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 256. if(p->parent == curproc){ 257. p->parent = initproc; 258. if(p->state == ZOMBIE) 259. wakeup1(initproc); 260. } 261. } 262. // Jump into the scheduler, never to return. 263. curproc->state = ZOMBIE; 264. sched(); 265. panic("zombie exit"); 266. } 267. // Wait for a child process to exit and return its pid. 268. // Return -1 if this process has no children. 269. /\*\* 270. \* 자식 프로세스가 종료될 때까지 기다림 271. \* @return 성공시 자식 pid, 에러시 -1 272. \* @warning proc.c 273. \*/ 274. int 275. wait(void) 276. { 277. struct proc \*p; 278. int havekids, pid; 279. struct proc \*curproc = myproc(); 281. acquire(&ptable.lock); //lock 획득 (병행성문제) 282. for(;;){ 283. // Scan through table looking for exited children. 284. havekids = 0; 285. //Zombie 프로세스를 찾는 과정 (자식이 fork되서 죽은 케이스) 286. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 287. if(p->parent != curproc) 288. continue; 289. havekids = 1; 290. //좀비 상태인 프로세스가 있으면? 291. // ZOMBIE : \*\*자식프로세스는 종료되었지만 부모프로세스가 wait하지 않은 상태 292. if(p->state == ZOMBIE){ 293. // Found one. 294. pid = p->pid; 295. kfree(p->kstack); //커널 스택 할당해제 296. p->kstack = 0; 297. freevm(p->pgdir); //물리적 pagetable 할당해제 (프로세스 페이지테이블 --> 메인메모리 할당.) 298. //프로세스 초기화 및 UNUSED 설정 299. p->pid = 0; 300. p->parent = 0; 301. p->name[0] = 0; 302. p->killed = 0; 303. p->state = UNUSED; 304. release(&ptable.lock); //리턴 전 lock 해제` 305. return pid; 306. } 307. } 308. // No point waiting if we don't have any children. 309. // 아무런 자식 프로세스를 가지지 않은 경우 Or 지금 프로세스가 죽은 경우 310. if(!havekids || curproc->killed){ 311. release(&ptable.lock); 312. return -1; 313. } 314. //모든 자식 프로세스가 실행 중이면 sleeplock 315. sleep(curproc, &ptable.lock); //DOC: wait-sleep 316. } 317. } 318. //PAGEBREAK: 42 319. /\*\* 320. \* 321. \* xv6 스케쥴러를 정의한 함수. 단순히 proc[NPROC] 를 앞에서부터 순회하며 아직 실행하지 않은 프로세스 선택 (RUNNABLE) 322. \* @warning cpu는 여러개 존재할 수 있으므로 Lock 획득 acquire ~ release를 잘해줄 것 323. \* 324. \* @return 스케쥴러는 리턴하지않음! (계속 for문을 돌며 switch함 : sched() 함수 내부에 switch 존재) 325. \* @warning Scheduler never returns. It loops, doing: (스케쥴러는 리턴하지않고 계속 루프를 돔) 326. \* @warning - choose a process to run (다음에 실행시킬 프로세스를 선택함) 327. \* @warning - swtch to start running that process (다음 프로세스를 전환시켜줌) 328. \* @warning - eventually that process transfers control (프로세스로 컨트롤을 넘겨줌) 329. \* @warning - switch() 호출로 스케쥴러로 다시 돌아옴 (via swtch back to the scheduler.) 330. \* @warning - switch() 는 어셈블리어로 다시 리턴시켜주는 함수 331. \* @warning #3 과제 프로세스 스케쥴러 손보기 과제 332. \* 333. \*/ 334. void 335. scheduler(void) 336. { 337. struct proc \*p; 338. struct cpu \*c = mycpu(); 339. c->proc = 0; 341. for(;;){ 342. // Enable interrupts on this processor. 343. sti(); 344. // Loop over process table looking for process to run. 345. acquire(&ptable.lock); //스케쥴러를 진입하여 Lock 설정 346. //Process Queue를 순회하며 process 가 Runnable 상태가 있으면 해당 process 실행 347. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 348. if(p->state != RUNNABLE) 349. continue; 350. // Switch to chosen process. It is the process's job 351. // to release ptable.lock and then reacquire it 352. // before jumping back to us. 353. c->proc = p; 354. //Context Switch (새롭게 할당받은 process 로 context switch) 355. switchuvm(p); 356. p->state = RUNNING; 357. //switch (A,B) : A-->B로 명령어 위치 분기함 (스케쥴러에서 p프로세스로 전환 (어셈블리어)) 358. swtch(&(c->scheduler), p->context); //Context Switch (Assembly 어로 switch) 359. switchkvm(); //Process 가 실행이 종료됨. 360. // Process is done running for now. 361. // It should have changed its p->state before coming back. 362. c->proc = 0; //프로세스가 종료되어돌아오면 c->proc=0 으로 설정 (현재 cpu의 프로세스가 없는 상태로 설정) 363. } 364. release(&ptable.lock); 365. } 366. } 367. // Enter scheduler. Must hold only ptable.lock 368. // and have changed proc->state. Saves and restores 369. // intena because intena is a property of this 370. // kernel thread, not this CPU. It should 371. // be proc->intena and proc->ncli, but that would 372. // break in the few places where a lock is held but 373. // there's no process. 374. /\*\* 375. \* 현 프로세스를 종료하고 스케쥴러로 돌아감 376. \* @warning sched() 내부에서 switch(A,B) 를 호출하여 A->B 로 분기함 (B는 scheduler 이겠지?) 377. \*/ 378. void 379. sched(void) 380. { 381. int intena; 382. struct proc \*p = myproc(); 383. if(!holding(&ptable.lock)) 384. panic("sched ptable.lock"); 385. if(mycpu()->ncli != 1) 386. panic("sched locks"); 387. if(p->state == RUNNING) 388. panic("sched running"); 389. if(readeflags()&FL\_IF) 390. panic("sched interruptible"); 391. intena = mycpu()->intena; 392. //switch (A,B) : A-->B로 명령어 위치 분기함 (스케쥴러로 돌아감) 393. swtch(&p->context, mycpu()->scheduler); 394. mycpu()->intena = intena; 395. } 396. // Give up the CPU for one scheduling round. 397. /\*\* 398. \* 프로세스 양보 함수 399. \* @warning 내부에서 sched() 호출함 400. \*/ 401. void 402. yield(void) 403. { 404. acquire(&ptable.lock); //DOC: yieldlock 405. myproc()->state = RUNNABLE; 406. sched(); 407. release(&ptable.lock); 408. } 409. // A fork child's very first scheduling by scheduler() 410. // will swtch here. "Return" to user space. 411. /\*\* 412. \* 프로세스가 가장 처음 fork 될 때 forkret 로 실행된다 413. \* @warning forkret 로 실행되면 iinit ~ initlog 를 설정하여 프로세스 초기설정을 담당한다. 414. \*/ 415. void 416. forkret(void) 417. { 418. static int first = 1; 419. // Still holding ptable.lock from scheduler. 420. release(&ptable.lock); //스케쥴러가 lock을 잡고있는 상태이므로 풀고 시작 421. if (first) { 422. // Some initialization functions must be run in the context 423. // of a regular process (e.g., they call sleep), and thus cannot 424. // be run from main(). 425. first = 0; //초기설정이 종료되었으므로 종료 426. iinit(ROOTDEV); 427. initlog(ROOTDEV); 428. } 429. // Return to "caller", actually trapret (see allocproc). 430. } 431. // Atomically release lock and sleep on chan. 432. // Reacquires lock when awakened. 433. /\*\* 434. \* 현재 프로세스를 sleep하고 커널모드로 스케쥴러로 돌아감 (sched() 호출) 435. \* @warning : 프로세스 전체를 아우르는 ptable 락을 푸는 동안 atomic 연산을 지원하기 위해 개별적인 lk을 두르고있음 436. \*/ 437. void 438. sleep(void \*chan, struct spinlock \*lk) 439. { 440. struct proc \*p = myproc(); 442. if(p == 0) 443. panic("sleep"); 444. if(lk == 0) 445. panic("sleep without lk"); 446. // Must acquire ptable.lock in order to 447. // change p->state and then call sched. 448. // Once we hold ptable.lock, we can be 449. // guaranteed that we won't miss any wakeup 450. // (wakeup runs with ptable.lock locked), 451. // so it's okay to release lk. 452. if(lk != &ptable.lock){ //DOC: sleeplock0 453. acquire(&ptable.lock); //DOC: sleeplock1 454. release(lk); 455. } 456. // Go to sleep. 457. p->chan = chan; 458. p->state = SLEEPING; 459. sched(); 460. // Tidy up. 461. p->chan = 0; 462. // Reacquire original lock. 463. if(lk != &ptable.lock){ //DOC: sleeplock2 464. release(&ptable.lock); 465. acquire(lk); 466. } 467. } 468. //PAGEBREAK! 469. // Wake up all processes sleeping on chan. 470. // The ptable lock must be held. 471. /\*\* 472. \* 자고있는 프로세스 중 파라미터 chan과 동일한 프로세스를 깨움 473. \* @param chan : 깨울 프로세스 주소 (void\*) 474. \*/ 475. static void 476. wakeup1(void \*chan) 477. { 478. struct proc \*p; 479. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) 480. if(p->state == SLEEPING && p->chan == chan) 481. p->state = RUNNABLE; 482. } 483. // Wake up all processes sleeping on chan. 484. /\*\* 485. \* wakeup 과정을 atomic 연산을 지원하기 위해 lock을 걸어줌 486. \*/ 487. void 488. wakeup(void \*chan) 489. { 490. acquire(&ptable.lock); 491. wakeup1(chan); 492. release(&ptable.lock); 493. } 494. // Kill the process with the given pid. 495. // Process won't exit until it returns 496. // to user space (see trap in trap.c). 497. /\*\* 498. \* 프로세스를 종료시킴 (프로세스에 kill 플래그(멤버변수) 를 설정함) 499. \* @param pid : 죽일 프로세스 ID 500. \*/ 501. int 502. kill(int pid) 503. { 504. struct proc \*p; 505. acquire(&ptable.lock); 506. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 507. if(p->pid == pid){ 508. //프로세스 죽이기 flag를 심어줌 509. p->killed = 1; 510. // Wake process from sleep if necessary. 511. //프로세스 상태가 Sleep 라면 Sched에 올려줌 512. if(p->state == SLEEPING) 513. p->state = RUNNABLE; 514. release(&ptable.lock); 515. return 0; 516. } 517. } 518. release(&ptable.lock); 519. return -1; 520. } 521. //PAGEBREAK: 36 522. // Print a process listing to console. For debugging. 523. // Runs when user types ^P on console. 524. // No lock to avoid wedging a stuck machine further. 525. /\*\* 디버깅용?\*/ 526. void 527. procdump(void) 528. { 529. static char \*states[] = { 530. [UNUSED] "unused", 531. [EMBRYO] "embryo", 532. [SLEEPING] "sleep ", 533. [RUNNABLE] "runble", 534. [RUNNING] "run ", 535. [ZOMBIE] "zombie" 536. }; 537. int i; 538. struct proc \*p; 539. char \*state; 540. uint pc[10]; 541. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ 542. if(p->state == UNUSED) 543. continue; 544. if(p->state >= 0 && p->state < NELEM(states) && states[p->state]) 545. state = states[p->state]; 546. else 547. state = "???"; 548. // 현재 실행되고있는 프로세스 출력 (embryo, sleep, runable ...) 549. cprintf("%d %s %s", p->pid, state, p->name); 550. if(p->state == SLEEPING){ 551. getcallerpcs((uint\*)p->context->ebp+2, pc); 552. for(i=0; i<10 && pc[i] != 0; i++) 553. cprintf(" %p", pc[i]); 554. } 555. cprintf("\n"); 556. } 557. } |

### proc.h

|  |
| --- |
| 1. // Per-CPU state 2. struct cpu { 3. uchar apicid; // Local APIC ID 4. struct context \*scheduler; // swtch() here to enter scheduler 5. struct taskstate ts; // Used by x86 to find stack for interrupt 6. struct segdesc gdt[NSEGS]; // x86 global descriptor table 7. volatile uint started; // Has the CPU started? 8. int ncli; // Depth of pushcli nesting. 9. int intena; // Were interrupts enabled before pushcli? 10. struct proc \*proc; // The process running on this cpu or null 11. }; 12. extern struct cpu cpus[NCPU]; 13. extern int ncpu; 14. //PAGEBREAK: 17 15. // Saved registers for kernel context switches. 16. // Don't need to save all the segment registers (%cs, etc), 17. // because they are constant across kernel contexts. 18. // Don't need to save %eax, %ecx, %edx, because the 19. // x86 convention is that the caller has saved them. 20. // Contexts are stored at the bottom of the stack they 21. // describe; the stack pointer is the address of the context. 22. // The layout of the context matches the layout of the stack in swtch.S 23. // at the "Switch stacks" comment. Switch doesn't save eip explicitly, 24. // but it is on the stack and allocproc() manipulates it. 25. struct context { 26. uint edi; 27. uint esi; 28. uint ebx; 29. uint ebp; 30. uint eip; 31. }; 32. enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE }; 33. // Per-process state 34. struct proc { 35. uint sz; // Size of process memory (bytes) 36. pde\_t\* pgdir; // Page table 37. char \*kstack; // Bottom of kernel stack for this process 38. enum procstate state; // Process state 39. int pid; // Process ID 40. struct proc \*parent; // Parent process 41. struct trapframe \*tf; // Trap frame for current syscall 42. struct context \*context; // swtch() here to run process 43. void \*chan; // If non-zero, sleeping on chan 44. int killed; // If non-zero, have been killed 45. struct file \*ofile[NOFILE]; // Open files 46. struct inode \*cwd; // Current directory 47. char name[16]; // Process name (debugging) 48. uint alarmticks; // #P2 Process ticks (default:0 alarm 설정 시 timer 증가) 49. uint alarm\_timer; // #P2 Process 타이머 (default:0xFFFFFFFF) 50. }; 51. // Process memory is laid out contiguously, low addresses first: 52. // text 53. // original data and bss 54. // fixed-size stack 55. // expandable heap |

### syscall.h

|  |
| --- |
| 1. // System call numbers 2. #define SYS\_fork 1 3. #define SYS\_exit 2 4. #define SYS\_wait 3 5. #define SYS\_pipe 4 6. #define SYS\_read 5 7. #define SYS\_kill 6 8. #define SYS\_exec 7 9. #define SYS\_fstat 8 10. #define SYS\_chdir 9 11. #define SYS\_dup 10 12. #define SYS\_getpid 11 13. #define SYS\_sbrk 12 14. #define SYS\_sleep 13 15. #define SYS\_uptime 14 16. #define SYS\_open 15 17. #define SYS\_write 16 18. #define SYS\_mknod 17 19. #define SYS\_unlink 18 20. #define SYS\_link 19 21. #define SYS\_mkdir 20 22. #define SYS\_close 21 23. #define SYS\_date 22 24. #define SYS\_alarm 23 |

### syscall.c

|  |
| --- |
| 1. #include "types.h" 2. #include "defs.h" 3. #include "param.h" 4. #include "memlayout.h" 5. #include "mmu.h" 6. #include "proc.h" 7. #include "x86.h" 8. #include "syscall.h" 9. // User code makes a system call with INT T\_SYSCALL. 10. // System call number in %eax. 11. // Arguments on the stack, from the user call to the C 12. // library system call function. The saved user %esp points 13. // to a saved program counter, and then the first argument. 14. // Fetch the int at addr from the current process. 15. /\*\* 16. \* systemcall 호출 등에 쓰이는 함수 인자를 받아오는 함수 (정수 전용) 17. \* @param addr : 받아올 인자 주소 --> 직접계산해서 넣어줘야함 18. \* @param ip : 인자를 받을 call-by-reference 변수 19. \* @return 성공시 0 에러시 -1 20. \* @warning argint 내부에서 호출됨 21. \* @warning syscall.c 22. \*/ 23. int 24. fetchint(uint addr, int \*ip) 25. { 26. struct proc \*curproc = myproc(); 27. if(addr >= curproc->sz || addr+4 > curproc->sz) 28. return -1; 29. \*ip = \*(int\*)(addr); 30. return 0; 31. } 32. // Fetch the nul-terminated string at addr from the current process. 33. // Doesn't actually copy the string - just sets \*pp to point at it. 34. // Returns length of string, not including nul. 35. /\*\* 36. \* systemcall 호출 등에 쓰이는 함수 인자를 받아오는 함수 (문자열 전용) 37. \* @param addr : 받아올 인자 주소 --> 직접계산해서 넣어줘야함 38. \* @param pp : 인자를 받을 call-by-reference 변수 39. \* @return 성공시 문자열 길이 에러시 -1 40. \* @warning argstr 내부에서 호출됨 41. \* @warning syscall.c 42. \*/ 43. int 44. fetchstr(uint addr, char \*\*pp) 45. { 46. char \*s, \*ep; 47. struct proc \*curproc = myproc(); 48. if(addr >= curproc->sz) 49. return -1; 50. \*pp = (char\*)addr; 51. ep = (char\*)curproc->sz; 52. for(s = \*pp; s < ep; s++){ 53. if(\*s == 0) 54. return s - \*pp; 55. } 56. return -1; 57. } 58. // Fetch the nth 32-bit system call argument. 59. /\*\* 60. \* n번째 정수형 파라미터를 \*ip로 call-by-ref 방식으로 가져옴 61. \* @param n : n번째 인자 (n=0부터 1번째 인자로 인식) 62. \* @param ip : n번째 파라미터 값을 가져올 변수 포인터 63. \* @return 성공시 0 에러시 -1 64. \* @warning 4 + 4\*n 이므로 0-based으로 가져옴 (즉 n=0 이면 1번째 parameter를 가져옴) 65. \* @warning syscall.c 66. \*/ 67. int 68. argint(int n, int \*ip) 69. { 70. //4 + 4\*n 이므로 0-based으로 가져옴 (즉 n=0 이면 1번째 parameter를 가져옴) 71. return fetchint((myproc()->tf->esp) + 4 + 4\*n, ip); 72. } 73. // Fetch the nth word-sized system call argument as a pointer 74. // to a block of memory of size bytes. Check that the pointer 75. // lies within the process address space. 76. /\*\* 77. \* n번째 주소정보를 \*\*pp 로 call-by-ref 방식으로 가져옴 78. \* @param n : n번째 인자 (n=0부터 1번째 인자로 인식) 79. \* @param pp : n번째 파라미터에 해당하는 주솟값을 가져올 call-by-ref 변수 80. \* @param size : 받아올 주소공간의 크기? --> 해당 파라미터 위치에서 + size 가 프로세스 메모리보다 큰지 체크해줌 81. \* @return 성공시 0 에러시 -1 82. \* @warning 4 + 4\*n 이므로 0-based으로 가져옴 (즉 n=0 이면 1번째 parameter를 가져옴) 83. \* @warning syscall.c 84. \*/ 85. int 86. argptr(int n, char \*\*pp, int size) 87. { 88. int i; 89. struct proc \*curproc = myproc(); 90. //우선 주소정보를 정수형으로 받아옴 (32비트 주소체게라서 주소정보가져올 수 있음) 91. if(argint(n, &i) < 0) 92. return -1; 93. // size가 초과하는지 체크 94. if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz) 95. return -1; 96. // 주소를 call-by-ref 형태로 넣어줌 97. \*pp = (char\*)i; 98. return 0; 99. } 100. // Fetch the nth word-sized system call argument as a string pointer. 101. // Check that the pointer is valid and the string is nul-terminated. 102. // (There is no shared writable memory, so the string can't change 103. // between this check and being used by the kernel.) 104. /\*\* 105. \* n번째 파라미터를 문자열 (string) 형태로 pp에 call-by-ref 형태로 넣어줌 106. \* @param n : n번째 인자 (n=0부터 1번째 인자로 인식) 107. \* @param pp : n번째 파라미터 값을 가져올 문자열 포인터(이중포인터 char\* 의 call-by-ref) 108. \* @return 성공시 '문자열 크기' 에러시 -1 109. \* @warning 4 + 4\*n 이므로 0-based으로 가져옴 (즉 n=0 이면 1번째 parameter를 가져옴) 110. \* @warning syscall.c 111. \*/ 112. int 113. argstr(int n, char \*\*pp) 114. { 115. int addr; 116. //우선 문자열 포인터가 유효한지 검사 --> 문자열 포인터를 가져옴 (32비트체계) 117. if(argint(n, &addr) < 0) 118. return -1; 119. //해당 포인터 위치의 값을 가져옴 120. return fetchstr(addr, pp); 121. } 122. extern int sys\_chdir(void); 123. extern int sys\_close(void); 124. extern int sys\_dup(void); 125. extern int sys\_exec(void); 126. extern int sys\_exit(void); 127. extern int sys\_fork(void); 128. extern int sys\_fstat(void); 129. extern int sys\_getpid(void); 130. extern int sys\_kill(void); 131. extern int sys\_link(void); 132. extern int sys\_mkdir(void); 133. extern int sys\_mknod(void); 134. extern int sys\_open(void); 135. extern int sys\_pipe(void); 136. extern int sys\_read(void); 137. extern int sys\_sbrk(void); 138. extern int sys\_sleep(void); 139. extern int sys\_unlink(void); 140. extern int sys\_wait(void); 141. extern int sys\_write(void); 142. extern int sys\_uptime(void); 143. extern int sys\_date(void); 144. extern int sys\_alarm(void); //전역변수로 sysfile.c에서 연결. (함수주소연결) 145. //시스템 콜 trap-table (전역변수) 146. static int (\*syscalls[])(void) = { 147. [SYS\_fork] sys\_fork, 148. [SYS\_exit] sys\_exit, 149. [SYS\_wait] sys\_wait, 150. [SYS\_pipe] sys\_pipe, 151. [SYS\_read] sys\_read, 152. [SYS\_kill] sys\_kill, 153. [SYS\_exec] sys\_exec, 154. [SYS\_fstat] sys\_fstat, 155. [SYS\_chdir] sys\_chdir, 156. [SYS\_dup] sys\_dup, 157. [SYS\_getpid] sys\_getpid, 158. [SYS\_sbrk] sys\_sbrk, 159. [SYS\_sleep] sys\_sleep, 160. [SYS\_uptime] sys\_uptime, 161. [SYS\_open] sys\_open, 162. [SYS\_write] sys\_write, 163. [SYS\_mknod] sys\_mknod, 164. [SYS\_unlink] sys\_unlink, 165. [SYS\_link] sys\_link, 166. [SYS\_mkdir] sys\_mkdir, 167. [SYS\_close] sys\_close, 168. [SYS\_date] sys\_date, 169. [SYS\_alarm] sys\_alarm, 170. }; 171. /\*\* 172. \* syscall 호출 할 때 trapframe 테이블에서 시스템콜을 찾아서 호출 173. \* @warning 프로세스 tf->eax = syscalls[num](); 으로 호출됨 174. \* @warning syscall.c 175. \*/ 176. void 177. syscall(void) 178. { 179. int num; 180. struct proc \*curproc = myproc(); 181. num = curproc->tf->eax; //현 프로세스에 걸려있는 시스템 콜 찾아내기 182. //시스템 콜이 유효한 범위 내에 있으면 실행됨 183. if(num > 0 && num < NELEM(syscalls) && syscalls[num]) { 184. // 여기 라인을 조정하면 모든 시스템 콜을 출력하게 해줄 수 있음 185. curproc->tf->eax = syscalls[num](); 186. } else { 187. //시스템 콜 호출 실패 시 출력오류 출력 188. cprintf("%d %s: unknown sys call %d\n", 189. curproc->pid, curproc->name, num); 190. curproc->tf->eax = -1; 191. } 192. } |

### sysfile.c

|  |
| --- |
| 1. // 2. // File-system system calls. 3. // Mostly argument checking, since we don't trust 4. // user code, and calls into file.c and fs.c. 5. // 6. #include "types.h" 7. #include "defs.h" 8. #include "param.h" 9. #include "stat.h" 10. #include "mmu.h" 11. #include "proc.h" 12. #include "fs.h" 13. #include "spinlock.h" 14. #include "sleeplock.h" 15. #include "file.h" 16. #include "fcntl.h" 17. #include "date.h" 18. // Fetch the nth word-sized system call argument as a file descriptor 19. // and return both the descriptor and the corresponding struct file. 20. static int 21. argfd(int n, int \*pfd, struct file \*\*pf) 22. { 23. int fd; 24. struct file \*f; 25. if(argint(n, &fd) < 0) 26. return -1; 27. if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0) 28. return -1; 29. if(pfd) 30. \*pfd = fd; 31. if(pf) 32. \*pf = f; 33. return 0; 34. } 35. // Allocate a file descriptor for the given file. 36. // Takes over file reference from caller on success. 37. static int 38. fdalloc(struct file \*f) 39. { 40. int fd; 41. struct proc \*curproc = myproc(); 42. for(fd = 0; fd < NOFILE; fd++){ 43. if(curproc->ofile[fd] == 0){ 44. curproc->ofile[fd] = f; 45. return fd; 46. } 47. } 48. return -1; 49. } 50. int 51. sys\_dup(void) 52. { 53. struct file \*f; 54. int fd; 55. if(argfd(0, 0, &f) < 0) 56. return -1; 57. if((fd=fdalloc(f)) < 0) 58. return -1; 59. filedup(f); 60. return fd; 61. } 62. int 63. sys\_read(void) 64. { 65. struct file \*f; 66. int n; 67. char \*p; 68. if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0) 69. return -1; 70. return fileread(f, p, n); 71. } 72. int 73. sys\_write(void) 74. { 75. struct file \*f; 76. int n; 77. char \*p; 78. if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0) 79. return -1; 80. return filewrite(f, p, n); 81. } 82. int 83. sys\_close(void) 84. { 85. int fd; 86. struct file \*f; 87. if(argfd(0, &fd, &f) < 0) 88. return -1; 89. myproc()->ofile[fd] = 0; 90. fileclose(f); 91. return 0; 92. } 93. int 94. sys\_fstat(void) 95. { 96. struct file \*f; 97. struct stat \*st; 98. if(argfd(0, 0, &f) < 0 || argptr(1, (void\*)&st, sizeof(\*st)) < 0) 99. return -1; 100. return filestat(f, st); 101. } 102. // Create the path new as a link to the same inode as old. 103. int 104. sys\_link(void) 105. { 106. char name[DIRSIZ], \*new, \*old; 107. struct inode \*dp, \*ip; 108. if(argstr(0, &old) < 0 || argstr(1, &new) < 0) 109. return -1; 110. begin\_op(); 111. if((ip = namei(old)) == 0){ 112. end\_op(); 113. return -1; 114. } 115. ilock(ip); 116. if(ip->type == T\_DIR){ 117. iunlockput(ip); 118. end\_op(); 119. return -1; 120. } 121. ip->nlink++; 122. iupdate(ip); 123. iunlock(ip); 124. if((dp = nameiparent(new, name)) == 0) 125. goto bad; 126. ilock(dp); 127. if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){ 128. iunlockput(dp); 129. goto bad; 130. } 131. iunlockput(dp); 132. iput(ip); 133. end\_op(); 134. return 0; 135. bad: 136. ilock(ip); 137. ip->nlink--; 138. iupdate(ip); 139. iunlockput(ip); 140. end\_op(); 141. return -1; 142. } 143. // Is the directory dp empty except for "." and ".." ? 144. static int 145. isdirempty(struct inode \*dp) 146. { 147. int off; 148. struct dirent de; 149. for(off=2\*sizeof(de); off<dp->size; off+=sizeof(de)){ 150. if(readi(dp, (char\*)&de, off, sizeof(de)) != sizeof(de)) 151. panic("isdirempty: readi"); 152. if(de.inum != 0) 153. return 0; 154. } 155. return 1; 156. } 157. //PAGEBREAK! 158. int 159. sys\_unlink(void) 160. { 161. struct inode \*ip, \*dp; 162. struct dirent de; 163. char name[DIRSIZ], \*path; 164. uint off; 165. if(argstr(0, &path) < 0) 166. return -1; 167. begin\_op(); 168. if((dp = nameiparent(path, name)) == 0){ 169. end\_op(); 170. return -1; 171. } 172. ilock(dp); 173. // Cannot unlink "." or "..". 174. if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0) 175. goto bad; 176. if((ip = dirlookup(dp, name, &off)) == 0) 177. goto bad; 178. ilock(ip); 179. if(ip->nlink < 1) 180. panic("unlink: nlink < 1"); 181. if(ip->type == T\_DIR && !isdirempty(ip)){ 182. iunlockput(ip); 183. goto bad; 184. } 185. memset(&de, 0, sizeof(de)); 186. if(writei(dp, (char\*)&de, off, sizeof(de)) != sizeof(de)) 187. panic("unlink: writei"); 188. if(ip->type == T\_DIR){ 189. dp->nlink--; 190. iupdate(dp); 191. } 192. iunlockput(dp); 193. ip->nlink--; 194. iupdate(ip); 195. iunlockput(ip); 196. end\_op(); 197. return 0; 198. bad: 199. iunlockput(dp); 200. end\_op(); 201. return -1; 202. } 203. static struct inode\* 204. create(char \*path, short type, short major, short minor) 205. { 206. struct inode \*ip, \*dp; 207. char name[DIRSIZ]; 208. if((dp = nameiparent(path, name)) == 0) 209. return 0; 210. ilock(dp); 211. if((ip = dirlookup(dp, name, 0)) != 0){ 212. iunlockput(dp); 213. ilock(ip); 214. if(type == T\_FILE && ip->type == T\_FILE) 215. return ip; 216. iunlockput(ip); 217. return 0; 218. } 219. if((ip = ialloc(dp->dev, type)) == 0) 220. panic("create: ialloc"); 221. ilock(ip); 222. ip->major = major; 223. ip->minor = minor; 224. ip->nlink = 1; 225. iupdate(ip); 226. if(type == T\_DIR){ // Create . and .. entries. 227. dp->nlink++; // for ".." 228. iupdate(dp); 229. // No ip->nlink++ for ".": avoid cyclic ref count. 230. if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0) 231. panic("create dots"); 232. } 233. if(dirlink(dp, name, ip->inum) < 0) 234. panic("create: dirlink"); 235. iunlockput(dp); 236. return ip; 237. } 238. int 239. sys\_open(void) 240. { 241. char \*path; 242. int fd, omode; 243. struct file \*f; 244. struct inode \*ip; 245. if(argstr(0, &path) < 0 || argint(1, &omode) < 0) 246. return -1; 247. begin\_op(); 248. if(omode & O\_CREATE){ 249. ip = create(path, T\_FILE, 0, 0); 250. if(ip == 0){ 251. end\_op(); 252. return -1; 253. } 254. } else { 255. if((ip = namei(path)) == 0){ 256. end\_op(); 257. return -1; 258. } 259. ilock(ip); 260. if(ip->type == T\_DIR && omode != O\_RDONLY){ 261. iunlockput(ip); 262. end\_op(); 263. return -1; 264. } 265. } 266. if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){ 267. if(f) 268. fileclose(f); 269. iunlockput(ip); 270. end\_op(); 271. return -1; 272. } 273. iunlock(ip); 274. end\_op(); 275. f->type = FD\_INODE; 276. f->ip = ip; 277. f->off = 0; 278. f->readable = !(omode & O\_WRONLY); 279. f->writable = (omode & O\_WRONLY) || (omode & O\_RDWR); 280. return fd; 281. } 282. int 283. sys\_mkdir(void) 284. { 285. char \*path; 286. struct inode \*ip; 287. begin\_op(); 288. if(argstr(0, &path) < 0 || (ip = create(path, T\_DIR, 0, 0)) == 0){ 289. end\_op(); 290. return -1; 291. } 292. iunlockput(ip); 293. end\_op(); 294. return 0; 295. } 296. int 297. sys\_mknod(void) 298. { 299. struct inode \*ip; 300. char \*path; 301. int major, minor; 302. begin\_op(); 303. if((argstr(0, &path)) < 0 || 304. argint(1, &major) < 0 || 305. argint(2, &minor) < 0 || 306. (ip = create(path, T\_DEV, major, minor)) == 0){ 307. end\_op(); 308. return -1; 309. } 310. iunlockput(ip); 311. end\_op(); 312. return 0; 313. } 314. int 315. sys\_chdir(void) 316. { 317. char \*path; 318. struct inode \*ip; 319. struct proc \*curproc = myproc(); 321. begin\_op(); 322. if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){ 323. end\_op(); 324. return -1; 325. } 326. ilock(ip); 327. if(ip->type != T\_DIR){ 328. iunlockput(ip); 329. end\_op(); 330. return -1; 331. } 332. iunlock(ip); 333. iput(curproc->cwd); 334. end\_op(); 335. curproc->cwd = ip; 336. return 0; 337. } 338. int 339. sys\_exec(void) 340. { 341. char \*path, \*argv[MAXARG]; 342. int i; 343. uint uargv, uarg; 344. if(argstr(0, &path) < 0 || argint(1, (int\*)&uargv) < 0){ 345. return -1; 346. } 347. memset(argv, 0, sizeof(argv)); 348. for(i=0;; i++){ 349. if(i >= NELEM(argv)) 350. return -1; 351. if(fetchint(uargv+4\*i, (int\*)&uarg) < 0) 352. return -1; 353. if(uarg == 0){ 354. argv[i] = 0; 355. break; 356. } 357. if(fetchstr(uarg, &argv[i]) < 0) 358. return -1; 359. } 360. return exec(path, argv); 361. } 362. int 363. sys\_pipe(void) 364. { 365. int \*fd; 366. struct file \*rf, \*wf; 367. int fd0, fd1; 368. if(argptr(0, (void\*)&fd, 2\*sizeof(fd[0])) < 0) 369. return -1; 370. if(pipealloc(&rf, &wf) < 0) 371. return -1; 372. fd0 = -1; 373. if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){ 374. if(fd0 >= 0) 375. myproc()->ofile[fd0] = 0; 376. fileclose(rf); 377. fileclose(wf); 378. return -1; 379. } 380. fd[0] = fd0; 381. fd[1] = fd1; 382. return 0; 383. } 384. //#P2 시스템콜 추가 385. int 386. sys\_date(void) 387. { 388. struct rtcdate\* rt = 0; 389. argptr(0, (void\*)&rt, sizeof(struct rtcdate\*)); //rtcdate 초과하는지 체크 및 변수 받아오기 390. if (rt == 0) 391. return -1; 392. cmostime(rt); 393. return 0; 394. } 395. //#define DEBUG 396. int 397. sys\_alarm(void) 398. { 399. int timer; 400. uint ticks\_convert = 100; //1 tick 은 대략 10ms 401. argint(0, &timer); 402. //Timer 가 0보다 작은 경우 에러처리 403. if (timer < 0) 404. return -1; 405. myproc()->alarm\_timer = (uint)timer \* ticks\_convert; //Timer 설정 406. #ifdef DEBUG 407. cprintf("timer set : %d\n", myproc()->alarm\_timer); 408. #endif 409. return 0; 410. } |

### trap.c

|  |
| --- |
| 1. #include "types.h" 2. #include "defs.h" 3. #include "param.h" 4. #include "memlayout.h" 5. #include "mmu.h" 6. #include "proc.h" 7. #include "x86.h" 8. #include "traps.h" 9. #include "spinlock.h" 10. #include "date.h" 11. #define P3\_TIMER 12. // Interrupt descriptor table (shared by all CPUs). 13. struct gatedesc idt[256]; 14. extern uint vectors[]; // in vectors.S: array of 256 entry pointers 15. struct spinlock tickslock; 16. uint ticks; 17. void 18. tvinit(void) 19. { 20. int i; 21. for(i = 0; i < 256; i++) 22. SETGATE(idt[i], 0, SEG\_KCODE<<3, vectors[i], 0); 23. SETGATE(idt[T\_SYSCALL], 1, SEG\_KCODE<<3, vectors[T\_SYSCALL], DPL\_USER); 24. initlock(&tickslock, "time"); 25. } 26. void 27. idtinit(void) 28. { 29. lidt(idt, sizeof(idt)); 30. } 31. //PAGEBREAK: 41 32. /\*\* 33. \* 트랩명령어를 처리하기 위한 trap 핸들러함수 (시스템콜 포함) 34. \* @param tf : 트랩테이블 (보통 인자로 넘겨줄 때 proc->tf) 35. \* @return 없음 36. \* @warning tf->trapno 는 trap 함수에 어떻게 왔는지 알려주는 flag와 같음 37. \* @warning trap.c 38. \*/ 39. void 40. trap(struct trapframe \*tf) 41. { 42. //트랩 번호가 시스템 콜일 경우 내부에서 syscall() 호출 43. if(tf->trapno == T\_SYSCALL){ 44. //프로세스가 죽어있으면 프로세스 종료 45. if(myproc()->killed) 46. exit(); 47. myproc()->tf = tf; 48. syscall(); //syscall 호출 49. if(myproc()->killed) 50. exit(); 51. return; 52. } 53. switch(tf->trapno){ 54. case T\_IRQ0 + IRQ\_TIMER: //Timer Interrupt 의 경우 55. //첫 번째 cpu 만 Timer\_Tcks 를 늘려주고있음 56. if(cpuid() == 0){ 57. acquire(&tickslock); 58. ticks++; //ticks 증가! (tcks 호출마다 증가) 59. //#P2 alarmticks 설정 시 증가 60. if (myproc() && myproc()->alarm\_timer != 0xFFFFFFFF) 61. myproc()->alarmticks++; 62. wakeup(&ticks); 63. release(&tickslock); 64. } 65. //두 번째 CPU도 늘려주고싶다면 설정할 것 66. lapiceoi(); 67. break; 68. case T\_IRQ0 + IRQ\_IDE: 69. ideintr(); 70. lapiceoi(); 71. break; 72. case T\_IRQ0 + IRQ\_IDE+1: 73. // Bochs generates spurious IDE1 interrupts. 74. break; 75. case T\_IRQ0 + IRQ\_KBD: 76. kbdintr(); 77. lapiceoi(); 78. break; 79. case T\_IRQ0 + IRQ\_COM1: 80. uartintr(); 81. lapiceoi(); 82. break; 83. case T\_IRQ0 + 7: 84. case T\_IRQ0 + IRQ\_SPURIOUS: 85. //무슨 인터럽트인지 모르겠음.. 86. cprintf("cpu%d: spurious interrupt at %x:%x\n", 87. cpuid(), tf->cs, tf->eip); 88. lapiceoi(); 89. break; 90. //PAGEBREAK: 13 91. default: 92. if(myproc() == 0 || (tf->cs&3) == 0){ 93. // In kernel, it must be our mistake. 94. cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n", 95. tf->trapno, cpuid(), tf->eip, rcr2()); 96. panic("trap"); 97. } 98. // In user space, assume process misbehaved. 99. cprintf("pid %d %s: trap %d err %d on cpu %d " 100. "eip 0x%x addr 0x%x--kill proc\n", 101. myproc()->pid, myproc()->name, tf->trapno, 102. tf->err, cpuid(), tf->eip, rcr2()); 103. myproc()->killed = 1; 104. } 105. // Force process exit if it has been killed and is in user space. 106. // (If it is still executing in the kernel, let it keep running 107. // until it gets to the regular system call return.) 108. if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER) 109. exit(); 110. //주의! xv6는 TIME Ticks 가 1개씩 증가할 때 마다 스케쥴러가 교체되는 RR 형태의 스케쥴러를 채용 중! 111. // RR ticks 주기를 늘리고 싶다면 해당 함수를 손 볼것 112. if(myproc() && myproc()->state == RUNNING && 113. tf->trapno == T\_IRQ0+IRQ\_TIMER) 114. yield(); 115. // Check if the process has been killed since we yielded 116. if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER) 117. exit(); 118. //프로세스 alarm\_timer 설정이 되었다면 timer() check 119. if (myproc() && myproc()->alarm\_timer != 0xFFFFFFFF) { 120. if (myproc()->alarmticks >= myproc()->alarm\_timer) { //timer 초과여부 확인 121. cprintf("SSU\_Alarm!\n"); 122. #ifdef P3\_TIMER 123. //과제 명세에 종료 전 출력부분이 있어 수정 124. struct rtcdate r; 125. cmostime(&r); 126. cprintf("Current time : %d-%d-%d %d:%d:%d\n", r.year, r.month, r.day, r.hour, r.minute, r.second); 127. #endif 128. exit(); //프로세스 종료 129. } 130. } 131. } |

### user.h

|  |
| --- |
| 1. struct stat; 2. struct rtcdate; 3. // system calls 4. int fork(void); 5. int exit(void) \_\_attribute\_\_((noreturn)); 6. int wait(void); 7. int pipe(int\*); 8. int write(int, const void\*, int); 9. int read(int, void\*, int); 10. int close(int); 11. int kill(int); 12. int exec(char\*, char\*\*); 13. int open(const char\*, int); 14. int mknod(const char\*, short, short); 15. int unlink(const char\*); 16. int fstat(int fd, struct stat\*); 17. int link(const char\*, const char\*); 18. int mkdir(const char\*); 19. int chdir(const char\*); 20. int dup(int); 21. int getpid(void); 22. char\* sbrk(int); 23. int sleep(int); 24. int uptime(void); 25. int date(struct rtcdate\*); //date 시스템콜 추가 26. int alarm(int); //alarm 시스템 콜추가 27. // ulib.c 28. int stat(const char\*, struct stat\*); 29. char\* strcpy(char\*, const char\*); 30. void \*memmove(void\*, const void\*, int); 31. char\* strchr(const char\*, char c); 32. int strcmp(const char\*, const char\*); 33. void printf(int, const char\*, ...); 34. char\* gets(char\*, int max); 35. uint strlen(const char\*); 36. void\* memset(void\*, int, uint); 37. void\* malloc(uint); 38. void free(void\*); 39. int atoi(const char\*); |

### user.S

* alarm 추가.

|  |
| --- |
| 1. #include "syscall.h" 2. #include "traps.h" 3. #define SYSCALL(name) \ 4. .globl name; \ 5. name: \ 6. movl $SYS\_ ## name, %eax; \ 7. int $T\_SYSCALL; \ 8. ret 9. SYSCALL(fork) 10. SYSCALL(exit) 11. SYSCALL(wait) 12. SYSCALL(pipe) 13. SYSCALL(read) 14. SYSCALL(write) 15. SYSCALL(close) 16. SYSCALL(kill) 17. SYSCALL(exec) 18. SYSCALL(open) 19. SYSCALL(mknod) 20. SYSCALL(unlink) 21. SYSCALL(fstat) 22. SYSCALL(link) 23. SYSCALL(mkdir) 24. SYSCALL(chdir) 25. SYSCALL(dup) 26. SYSCALL(getpid) 27. SYSCALL(sbrk) 28. SYSCALL(sleep) 29. SYSCALL(uptime) 30. SYSCALL(date) 31. SYSCALL(alarm) |