**191028 Ch7. Event Driven Programming**

* **Space Programming: the curses library**
  + The curses library: terminal을 제어, cursor의 위치를 조절, text의 형태 제어
  + basic curses functions
    - initscr() -초기화
    - endwin() – turn off curses and reset tty terminal setting 초기화
    - refresh() – 화면에 출력
    - move(r, c) – cursor 이동
    - addstr(s), addch(c) – string 또는 char 그리기
    - clear()
    - standout(), standend(): standout mode를 키고 끔 (text 강조)
  + Curses는 두 버전의 화면을 저장(virtual, updated) – 변화된 부분만 전송
* **Time Programming: sleep**
  + 움직이는 물체: 특정한 시간 간격마다 출력 – timer, interval 설정
* **Programming with Time: Alarms, Interval Timers**
  + sleep(n) - process를 특정 시간 n초만큼 block 후 alarm signal 발생
    - signal(SIGALRM, handler);

alarm(n); - n초 뒤에 alarm signal 발생

pause(); - signal 발생 전까지 process block

* + - 단점: 시간 설정의 제약 (1초 단위)
  + Interval Timer 3가지
    - Real Timer: 실제 실행 시간 = Elapsed time = CPU time + IO time + waiting time
    - Virtual Timer: process에 의한 시간 = CPU time (user mode)
    - Profile Timer: process & system calls(kernel)의 동작시간 = user + kernel mode time
    - Virtual & Profile timer 구분 이유: 최적화 기준을 설정하기 위함
  + Interval Timer 정하기
    - 1. initial interval(초기 주기) & 2. repeating interval(반복 주기)
    - itimerval 구조체에 저장 – timeval 구조체 - microsecond까지 저장
    - getitmer(timer, struct itimerval \*val) – 현재 설정 값 불러오기

setitmer(timer, struct itimerval \*newval, struct itimerval \*oldval) – 값 설정

* + timer의 역할: suspend execution, schedule future action
* **Signal Handling I: Using signal**
  + Signal: 프로그램 동작 중에 발생하는 interruption
    - 종류: keystroke / Illegal process / elapsed timer등
    - 반응: Default action, signal / Ignore signal / Invoke function
  + 문제: 처음 signal 발생 후 handler 실행 중 다른 signal 발생하는 경우 그냥 지나 감
  + 한계: 어떤 signal이 발생했는지는 알 수 있지만, signal이 발생한 이유는 모름
  + inthandler와 signal 사이에 발생하는 경우 문제 발생
  + SIGQUIT을 무시만 할 수 있고, block(지연)할 수 있는 방법이 없음.
* **Signal Handling II: sigaction**
  + **sigaction**(signum, action, prevaction) – signal 처리방식을 더 세밀하게 정해 줌
  + 사용하는 이유
    - signal() -> 특정 signal 동작을 정하는데 한계
    - signal -> handler 발생 -> 원인 알기 위해
    - sys. call 호출, signal 발생 -> return 안하고 handler 들어갔다가 다시 돌아오도록
  + struct sigaction – signal의 처리방법, 동작 등 기술
    - void **\*sa\_handler** – signal 발생시 호출할 값 – SIG\_DFL, SIG\_IGN, function
    - void (**\*sa\_sigaction**)(int signum, siginfo\_t \*, void \*) – 새로운 handler
      * 두 handler 중 한 function을 통해 signal에 대한 정보(발생 이유)를 알 수 있음. -> signal() sys. call 과의 차이점
    - sigset\_t **sa\_mask** – block 할 signal을 저장
    - int **sa\_flags**: signal handling 시의 동작을 세밀하게 기술
      * SA\_RESETHAND – mousetrap mode – handler 실행 이후 signal 재발생 시 default action 수행, 동일한 handler 실행 위해 초기화
      * SA\_NODEFER – block 안하고 signal 다시 실행하도록 해줌
      * SA\_RESTART – return 안하고 restart
      * SA\_SIGINFO – hander 2개 중에 선택
* **Protecting Data from Corruption**
  + Critical Sections: interrupt가 생기는 경우 망가질 수 있는 중요한 부분
    - 보호하는 방법: signal을 block 또는 ignore 한다.
  + **sigprocmask**(how, sigset\_t \*sigs, sigset\_t \*prev) – 특정 signal을 block
    - how: SIG\_BLOCK, SIG\_UNBLOCK, SIG\_SET
    - \*sigs: list of signals to use
    - \*prev: previous signal mask
  + building signal sets with sigsetops
    - sigemptyset(\*setp) – signal 모음(setp)을 비워줌
    - sigfillset(\*setp) – 모든 signal을 setp list에 넣음
    - sigaddset(\*setp, signum) – setp 모음에서 signum을 넣음
    - sigdelset(\*setp, signum) – setp 모음에서 signum을 뺌
  + Reentrant Code: 이미 실행 중에 다시 실행되어도 문제 발생 X 함수
    - recursive – set SA\_NODEFER flag
    - blocking – clear SA\_NODEFER flag
* **Kill: Sending Signals from a Process**
  + 한 process가 다른 process에게 signal을 보낼 때 사용
    - ex) inter-process communication
  + **kill**(pid, sig) – process에 signal을 보냄
    - pit\_t pid: process id of target
    - int sig: signal
  + 특징)
    - 조건: 전송하는 대상과 같은 user ID를 갖거나, 관리자(superuser)여야 함
    - 자기 자신에게도 보낼 수 있음
    - 모든 종류의 signal을 보낼 수 있음. – keyboard, timer, kernel
    - custom 가능한 default signal: SIGUSR1, SIGUSR2
* **Signals on Input: Asynchronous I/O**
  + 이전까지: timer 통해 user input 기다림 -> event driven X

user input 들어올 때마다 action -> event-driven O

목표: SIGIO signal 발생하면 동작하도록 만들어주기

* + pause() 기능: signal 발생 전까지 process를 ‘suspend’상태로 만들어 CPU가 다른 process를 실행할 수 있도록 해줌.
  + 방법1: Using O\_ASYNC

1. ‘handler’ 만들기
2. fcntl (file attribute 바꿔줌) 이용하여 F\_SETOWN (SIGIO) 설정 – input notification signal 전송
3. input signal 켜기 – O\_ASYNC (기다리지 않고 바로 return)를 file descriptor 0(stdin)에 설정
4. signal 기다리기 위해 pause 반복
   * 방법2: Using aio\_read
5. ‘handler’ 만들기 (on\_input) – input이 들어왔을 때 실행
6. aiocb 구조체 – 기다릴 input & 전송할 signal 저장
7. read 명령으로 aio\_read에게 구조체 전달
8. write handler – aio\_return
   * Asynchronous I/O가 필요한 이유?
     + 의미 없는 조건 검사(pause 실행을 위한)를 반복할 필요 X

**191104-1111 Ch8. Processes and Programs – Studying sh**

* **Process**
  + Program: 기계어 명령의 모음
  + Process: 실행중인 Program
* **Learning about Processes with ps**
  + “ps” command: 현재 process들을 나열
    - PID: process ID
    - TTY: user가 로그인 되어있는 terminal의 이름
    - TIME: process가 실행된 CPU time
    - CMD: process를 실행한 command의 이름
    - ‘-a’: 다른 user와 terminal의 process까지 나열
* **The Shell: A Tool for Process and Program Control**
  + Shell: process를 관리하고 program을 실행하는 program
    - 기능
      * user input ‘Interpreter’
      * program ‘Launcher’
      * input & output ‘Manager’
      * programming language
* **How the Shell Runs Programs**
  + 동작 순서)
    1. user의 입력
    2. program 실행 위해 새로운 process 생성
    3. disk에서 process로 program 전달
    4. process 실행
  + **execvp**(\*file, \*argv) – file을 실행
    - \*file: 실행할 파일 이름
    - argv: program에 전달할 문자열
    - 실행순서)
      1. kernel이 disk에서 process로 program 전달
      2. kernel이 arglist를 process로 복사
      3. kernel이 main(argc, argv) 실행
    - 문제: 현재 process의 shell program image(code)를 날리고 새로운 command로 바꾸고 종료 해버림
    - 해결: 새로운 process를 만들어서 program을 실행하도록 함 –> fork()
  + **fork**() – 똑같은 process를 만듦
    1. 새로운 memory 공간 할당
    2. original process(parent)를 new process(child)로 복사
    3. 새로운 process를 running process에 추가
    4. fork() 이후의 작업들 실행
    - return: child(0), parent(pid)
  + child process가 실행 종료될 때까지 parent는 기다려야 함 -> wait()
    - 종료 전에 parent가 종료되면 – ‘고아’ -> init process가 자동으로 입양
    - **wait**(\*statusptr) – process가 종료될 때까지 기다림
      * \*statusptr: child의 성공 여부 확인 (16bit)
        + exit value(8) + core dump flag(1) + signal number(7)
      * return: 종료된 process
      * child가 종료될 때까지 parent는 pause
      * child가 exit()을 하면 parent가 wake하고 child가 준 값을 전달
      * 2가지 기능: notification & communication
  + zombie process: child process가 wait X -> kernel memory 공간이 남아있음

**191111-1118 Ch10. I/O Redirection, Pipes**

* **The Shell**
  + 두번째 기능: Manage Input and Output
* ‘>’ : 실행 결과가 화면이 아닌 파일로 출력
* ‘|’ : pipe를 통해 출력값이 다음 파일의 input으로 들어감
* **A Shell application: Watch for Users**
  + shell script: command line 나열
  + who command를 이용한 application
    - ‘comm’: 두개의 list를 비교하여 1(prev만), 2(curr만), 3(prev&curr 동시)을 출력
* **Facts about Standard I/O and Redirection**
  + Unix I/O Redirection – based on the principle of standard streams of data
  + Fact 1: Three Standard File Descriptors

0: Standard Input – stream of data to process

1: Standard Output – stream of result data

2: Standard Error – stream of error msg

* + - output은 보통 ‘이름’이 아닌, file descriptor 1로 출력
      * 해결: fd1을 또다른 file로 출력하도록 ‘redirection’ 해줘야 함
        + $ cmd > filename
  + Fact 2: The “Lowest-Available-fd” Principle
    - file descriptor: 배열의 index
    - file을 open할 때, 배열의 lowest spot을 받음
    - 사용하지 않는 fd 중 index#가 가장 작은 것부터 할당
    - 각 process 별로 fd table이 있음 -> file access (0, 1, 2)
    - fork()로 child fd 생성 (parent fd 복사)
* **How to Attach stdin to a File**
  + ‘<’ : file becomes source for standard input
  + Method 1: Close then Open
    - close(0) – file descriptor의 0번을 닫는다
    - open(filename, O\_RDONLY) – file과 연결시켜 stdin으로 받는다
  + Method 2: open..close..dup..close
    - dup, dup2: file descriptor를 복사함
    - dup(oldfd) – return newfd
    - dup2(oldfd, newfd) – return newfd
* **Redirecting I/O for Another Program: who > userlist**

**191118-25 Ch11. Connecting to Processes Near and Far – Servers and Sockets**

* **bc: a Unix Calculator**
* **popen: Making Process Look like Files**
  + popen() – 프로세스 생성 -> 프로그램 실행 -> stdout 값을 현재 프로세스에서 읽음
  + pclose() 필수 – child process가 끝날 때까지 wait
  + 작동원리
    - pipe, fork를 통해 Parent는 Reading end, Child는 Writing end로 사용
    - **Parent**
      1. close(p[1]) - writing end 닫기
      2. fp = fdopen(p[0], “r”) – reading end의 fd 불러옴
      3. return fp
    - **Child**
      1. close(p[0]) – reading end 닫기
      2. dup2(p[1], 1) – writing end를 fd1으로 복사
      3. close(fd1) – writing end 닫기
      4. execl(“/bin/sh”, “sh”, “-c”, cmd, NULL) – 명령어 입력, cmd에 input
* **Sockets: Connecting to Remote Processes**
  + Pipe의 단점:
    - 관계 있는 process 사이에서만 만들 수 있음
    - 동일한 컴퓨터 내에서 실행중인 process 사이에서만 가능
    - 결론) socket 사용 -> 관련이 없어도, 다른 컴퓨터 간에 연결
  + Basic Concepts
    - Server: request를 받고, 처리하고, 다름 request를 계속 받는 program
    - Hostname: 컴퓨터마다 가지고 있는 이름
    - Port Number: 컴퓨터와 연결된 여러 server 중 어떤 것과 상호작용할지 정해줌

(ex) 웹사이트 서버: 80, SSH: 22)

* + - Address: host마다 가지고 있는 주소
    - Protocol: 통신 조약
  + Time server

1. Get a phone line – socket()
2. Assign a number – bind()
3. Allow incoming calls – listen()
4. Wait for a call – accept()
5. Transfer data – read() / write()
6. Hang up – close()
7. **socket**(domain, type, protocol) – 소켓 생성
   * domain: PF\_INET, type: SOCK\_STREAM, protocol: 0
8. **bind**(sockfd, \*addr, addrlen) – 주소 할당
   * sockfd: file descriptor
   * \*addr: IP주소, port # 저장되어 있는 structure pointer
9. **listen**(sockfd, backlog) – socket을 passive로 만듦 (connection 기다림)
   * backlog: 최대 대기 call 수 제한 – kernel memory 소진 방지
   * backlog 이상의 connection request가 들어온 경우 – ERROR
10. **accept**(sockfd, \*addr, \*addrlen) – connection 받아줌
    * \*addr: client의 주소
    * client의 connect() request와 연결, synchronize
    * connection이 없으면 – blocked
    * Return: 새로운 active socket 생성 – file descriptor 반환
11. **read**(), **write**(),
12. **close**()
    * Time client
    * 1. Get a phone line – socket()
    * 2. Call the server – connect()
    * 3. Transfer data – read() / write()
    * 4. Hang up – close()
13. socket()
14. connect(sockfd, \*addr, addrlen) – active socket을 연결
    * sockfd: active socket의 file descriptor
    * addr: listening socket의 주소 저장되어 있는 구조체

**191125-1202 Ch14. Threads, Concurrent Functions**

* **Doing Several Things at Once**
  + fork & exec – 여러 program 동시 실행
  + 하나의 program 안의 여러 function 사용 / 동일한 function을 동시에 여러 번 실행 -> Thread 사용
  + **Process**의 구성
    - Address space – code, data 저장
      * 서로 다른 process 간에는 data 공유 X (adr. space 다르기 때문)
    - Thread state(=Execution state)- PC, stack pointer, register 등 저장
    - OS resources
  + Concurrency vs. Parallelism
    - Concurrent Execution: 여러 task들을 짧은 시간 간격동안 조금씩 번갈아 실행해서 동시에 실행되는 것처럼 보이는 것
    - Parallelism – multi-core를 통해 task들이 실제로 동시에 실행
  + **Thread**
    - Process(program code, data, files, address space, OS resources)는 공유
    - Execution State(stack, stack pointer, PC, registers)는 개별로 가짐
    - 하나의 process – 여러 개의 thread – 동시에 하나의 function 실행
  + Thread vs. Process
    - **Process**: address space, general process attributes 정의
      * heavy-weight operation
      * own memory space
      * process 간의 교류 느림 – memory address가 다르기 때문
      * 다른 process간에 memory를 공유 X
    - **Thread**: sequential execution stream 정의
      * light-weight operations
      * 속해 있는 process의 memory를 사용, 공유
      * thread 간의 교류 빠름 – memory(전역변수 포함 등) 공유
      * address space 안에서 여러 개의 thread를 가질 수 있음
      * thread 간의 data 공유 – 효율
      * scheduling의 단위로 사용
* **Thread Creation**
  + **pthread\_create**(\*thread, \*attr, \*func, \*arg) – thread 생성
    - \*thread: pthread\_t pointer
    - \*attr: pthread\_attr\_t pointer
    - func: 이 thread가 실행할 function
    - arg: func에 전달될 arguments
  + **pthread\_join**(thread, \*\*retval) - thread가 종료될 때까지 calling thread를 block 한다
    - thread: wait할 thread
    - \*\*retval: thread에게서 받을 return value
* **Inter-thread Cooperation**
  + (pipe, socket, signal, exit/wait) 등은 system call이기 때문에 느리다.
  + Thread는 single process에서 function들을 execute하기 때문에 address space(global variable 등)를 공유
  + memory으로의 동시 접근은 강력하지만 위험하다
    - thread의 실행 순서를 정할 수 X -> 결과 예상 X, 실행 환경에 따라 다른 결과
  + twordcount1.c – thread 실행 순서 지정 X
    - 문제: 여러 core들이 memory에서 동시에 데이터 불러오고 동시에 update
    - 의도: 하나의 thread 실행이 끝난 후 다음 thread 실행
      * 방법: 한 번에 하나의 thread에서만 update 하도록 -> “**Mutex**”
  + twordcount2.c – 2 Thread + 1 Counter + 1 Mutex
    - Mutex: Mutual Exclusion lock
      * ‘lock’을 가져간 thread만 변수를 update 할 수 있도록 제어
    - **pthread\_mutex\_lock**(\*mutex) – key를 받음
      * \*mutex: 막아 놓을 대상
    - **pthread\_mutex\_unlock**(\*mutex) – key를 반납
    - 문제: mutex 사용은 프로그램을 느리게 만듦
  + twordcount3.c – 2 Thread + 2 Counter + Multiple Arguments
    - 각 thread에게 counter를 제공
    - struct arg\_set: Multiple Arguments를 위한 구조체
    - local struct에게 pointer를 전달하므로 mutex나 global variable이 필요 X
* **Inter-thread Notification**
  + thread가 동작을 마쳤을 때 original thread에게 완료를 알리는 방법?
  + **pthread\_cond\_wait**(\*cond, \*mutex) – thread를 block한다
    - \*cond: condition variable pointer
    - \*mutex: 하나의 thread만 cond. var.을 set 하도록
    - condition variable이 set 될 때까지 sleep 상태
    1. 함수 호출 전 mutex lock
    2. wait -> 함수 호출 -> mutex release(unlock)
    3. 호출한 thread -> block
  + **pthread\_cond\_signal**(\*cond) – condition variable을 기다리는 thread를 unblock
  + twordcount4.c
  + tanimate.c
    - mutex를 통해 하나의 thread가 메시지 출력 마칠 때까지 lock 시켜줌