#1

0.ALP textbook 실습

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
  pid_t child_pid;
  printf ("the main program process id is %d\n", (int) getpid ());
  child_pid = fork ();
  if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process id is %d\n", (int) child_pid);
  }
  else
    printf ("this is the child process, with id %d\n", (int) getpid ());
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
/* Spawn a child process running a new program.  PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list.  Returns the process id of
   the spawned process.  */
int spawn (char* program, char** arg_list)
{
  pid_t child_pid;
  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program, arg_list);
```

```
    /* The execvp function returns only if an error occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}
int main ()
{
  int child_status;
  /* The argument list to pass to the "ls" command.  */
  char* arg_list[] = {"ls","-l","/",NULL};
     /* argv[0], the name of the program.  */
     /* The argument list must end with a NULL.  */

  /* Spawn a child process running the "ls" command.  Ignore the
     returned child process id.  */
  spawn ("ls", arg_list);
  wait(&child_status);
  if(WIFEXITED(child_status))
    printf("the child process exited normally,with exit code %d\n",
WEXITSTATUS(child_status));
  else
    printf("the child process exited abnormally");
  // printf ("done with main program\n");
  return 0;
}
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
  pid_t child_pid;
  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* This is the parent process.  Sleep for a minute.  */
    sleep (60);
  }
  else {
    /* This is the child process.  Exit immediately.  */
    exit (0);
  }
  return 0;
}
```

1.Wating for a Process

- Sometimes, we would like to find out when a child process has finished.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

- 하위 프로세스 하나가 중지 될 때 까지 상위 프로세스가 일시 중지 되도록 함 / 자식 프로세스의 pid를 반환함
- 정상적으로 종료된 프로세스..리턴 혹은 exit값을 전달 되게 함 상태정보
- 상태정보들임

Macro	Definition
WIFEXITED(stat_val)	Non-zero if the child is terminated normally.
WEXITSTATUS(stat_val)	If WIFEXITED is non-zero, this returns child exit code.
WIFSIGNALED(stat_val)	Non-zero if the child is terminated on an uncaught signal.
WTERMSIG(stat_val)	If WIFSIGNALED is non-zero, this returns a signal number.
WIFSTOPPED(stat_val)	Non-zero if the child has stopped.
WSTOPSIG(stat_val)	If WIFSTOPPED is non-zero, this returns a signal number.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    pid_t pid;
    char *message;
    int n;
    int exit_code;
    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            message = "This is the child";
            n = 5;
            exit_code = 37;
```

```
            break;
        default:
            message = "This is the parent";
            n = 3;
            exit_code = 0;
            break;
    }
    for(; n > 0; n--)
    {
        puts(message);
        sleep(1);
    }
    if (pid != 0) {
        int stat_val;
        pid_t child_pid;
        child_pid = wait(&stat_val);
        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit(exit_code);
}
```

2. Signal

Signal Name	Description
SIGABORT	*Process abort.
SIGALRM	Alarm clock.
SIGFPE	*Floating point exception.
SIGHUP	Hangup.
SIGILL	*Illegal instruction.
SIGINT	Terminal interrupt.
SIGKILL	Kill (can't be caught or ignored).
SIGPIPE	Write on a pipe with no reader.
SIGQUIT	Terminal quit.
SIGSEGV	*Invalid memory segment access.
SIGTERM	Termination.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

- 이 다소 복잡한 선언은 신호가 sig와 func의 두 매개 변수를 취하는 함수라는 것을 나타냅니다.
- 잡히거나 무시 될 신호는 sig 인수로 주어집니다.
- 지정된 신호가 수신 될 때 호출되는 함수는 func로 주어집니다.
- 이 함수는 단일 int 인수 (수신 된 신호)를 취해 void 유형이어야합니다.
- 신호 함수 자체는이 신호를 처리하도록 설정된 함수의 이전 값 또는이 두 특수 값 중 하나 인 같은 유형의 함수를 반환합니다.

SIG_IGN	Ignore the signal.
SIG_DFL	Restore default behavior.

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig) {
  printf("OUCH! - I got signal %d\n", sig);
  (void) signal(SIGINT, SIG_DFL);
}
int main()
{
  (void) signal(SIGINT, ouch);

  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Sending Signals

- 프로세스는 kill을 호출하여 자신을 포함한 다른 프로세스에 신호를 보낼 수 있습니다.
- 대상 프로세스가 다른 사용자의 소유이기 때문에 일반적으로 프로그램에 신호를 보낼 수있는 권한이 없으면 호출이 실패합니다.
- 이것은 동일한 이름의 쉘 명령에 해당하는 프로그램입니다.

```c
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- kill 함수는 지정된 시그널 인 sig를 식별자가 pid로 주어지는 프로세스로 보낸다.
- 성공하면 0을 반환합니다.
- 신호를 보내려면 보내는 프로세스에 권한이 있어야합니다.
- 일반적으로 이것은 두 프로세스가 동일한 사용자 ID를 가져야한다는 것을 의미합니다.
- 즉 수퍼 유저가 모든 프로세스에 신호를 보낼 수는 있지만 자신의 프로세스 중 하나에만 신호를 보낼 수 있습니다.
- kill은 실패하고, -1을 반환하고 주어진 신호가 유효한 값 (errno는 EINVAL로 설정)이면 errno를 설정한다
- 권한 (EPERM)이 없거나 지정된 프로세스가 존재하지 않으면 (ESRCH) errno를 설정합니다.
- 신호는 우리에게 유용한 자명종 시설을 제공합니다. 알람 함수 호출은 프로세스가 미래의 어느 시점에 SIGALRM 신호를 스케쥴링하는 데 사용될 수 있습니다.

```c
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
```

```c
static int alarm_fired = 0;
void ding(int sig)
{
  alarm_fired = 1;
}
int main()
{
  pid_t pid;
  printf("alarm application starting\n");
  pid = fork();
  switch(pid)
  {
    case -1:
    /* Failure */
      perror("fork failed");
      exit(1);
    case 0:
    /* child */
      sleep(5);
      kill(getppid(), SIGALRM);
      exit(0);
    }
/* if we get here we are the parent process */
  printf("waiting for alarm to go off\n");
  (void) signal(SIGALRM, ding);
  pause(); //시그널 수신까지 기다림
  if (alarm_fired)
    printf("Ding!\n");

  printf("done\n");
  exit(0);
}
```

- 이 프로그램은 신호가 발생할 때까지 프로그램이 일시 중지되도록하는 새로운 함수 pause씀. 신호를 받으면 설정된 핸들러가 실행되고 정상적으로 실행이 계속됩니다.
- 신호에 의해 인터럽트되었을 때 errno를 EINTR로 설정하고 -1을 반환합니다 (다음 수신 신호가 프로그램을 종료시키지 않는 경우). - 인터럽트에 인한 중단
- 신호를 기다릴 때 sigsuspend를 사용하는 것이 더 일반적입니다. 우리는 1 분 안에 만날 것입니다.
- 알람 시계 시뮬레이션 프로그램은 포크를 통해 새로운 프로세스를 시작합니다. 이 하위 프로세스는 5 초 동안 대기 한 다음 SIGALRM을 부모에게 보냅니다.
- 부모는 SIGALRM을 잡아서 신호가 수신 될 때까지 일시 중지합니다. 우리는 시그널 핸들러에서 직접 printf를 호출하지 않고 플래그를 설정 한 다음 나중에 플래그를 확인합니다.

4. A Robus Signals Interface

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act, struct sigaction
*oact);
void (*) (int) sa_handler/* function, SIG_DFL or SIG_IGN*/
sigset_t sa_mask; /* signals to block in sa_handler */
int sa_flags /* signal action modifiers*/
```

지정된 시그날의 수신시 취할 액션을 정의하는데 사용되는 시그날 싱 구조는 signal.h에 정의되어 있

- sigaction 함수는 시그널과 연관된 액션을 설정한다.
- oact가 null이 아닌 경우, sigaction은 이전 신호 액션을 그것이 참조하는 위치에 기록합니다.
- act가 null의 경우, 이것은 모두 sigaction가합니다. act가 null이 아닌 경우 지정된 신호에 대한 작업이 설정됩니다.
- 인수 act가 가리키는 sigaction 구조 내에서 sa_handler는 signal sig가 수신 될 때 호출되는 함수에 대한 포인터입니다.
- 이것은 이전에 우리가 보았던 함수 func가 신호를 전달한 것과 매우 흡사합니다.
- sa_handler 필드에서 특수 값 SIG_IGN과 SIG_DFL을 사용하여 신호가 무시되거나 조치가 기본값으로 복원됨을 나타낼 수 있습니다.
- sa_mask 필드는 sa_handler 함수가 호출되기 전에 프로세스의 시그널 마스크에 추가 될 신호들의 집합을 지정한다. 신호는 차단되어 프로세스로 전달되지 않습니다. . sa_mask 필드를 사용하면이 경쟁 조건을 제거할 수 있습니다. 시그널 핸들러 함수가 실행되는 동안 블럭되어야 하는 시그널 마스크 제공
- 그러나 sigaction에 의해 설정된 핸들러로 잡힌 신호는 기본적으로 리셋되지 않으며 시그널로 이전에 보았던 동작을 얻으려면 sa_flags 필드가 SA_RESETHAND 값을 포함하도록 설정되어야합니다.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig) {
  printf("OUCH! - I got signal %d\n", sig);
}
int main() {
  struct sigaction act;
  act.sa_handler = ouch;//
  sigemptyset(&act.sa_mask);//sa_mask 사용시 경쟁조건 제거라고? 차단 안됨?..
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);//이전 액션이 없으면서 sa_flags가 0
  while(1) {
    printf("Hello World!\n");
    sleep(1);
  }
}
```

```
```

```c
/*Signal Example with wait() and waitpid() SIGUSR1, SIGUSR2, and SIGINT*/
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>
#include <errno.h>

static void signal_handler(int);

int i, pid1, pid2, status;//전역변수선언

int main( int argc, char *argv[], char *env[] )
{
  int exit_status;
  //SIGUSR1 에대한 핸들러 설정 , 만약 에러시 출력
  if( signal( SIGUSR1, signal_handler) == SIG_ERR )
    printf("P?ent: Unable to create handler for SIGUSR1\n");
  //SIGUSR2 에대한 핸들러 설정 만약 에러시 출력
  if( signal( SIGUSR2, signal_handler) == SIG_ERR )
    printf("P?ent: Unable to create handler for SIGUSR2\n");

  //부모의 pid를 pid1에 넣고 출력
  printf( "Parent pid = %d\n", pid1=getpid());

  //fork를 하고 pid2 자식의 pid를 넣는다, 자식의 pid2는 0이다
  if( (pid2 = fork()) == 0 )
  { //자식 프로세스의 경우 이 조건문에 들어가게 된다.
    //자신의 pid를 출력
    printf( "Child pid = %d\n", getpid() );
    printf( "Child: sending parent SIGUSR1\n", getpid() );
    //pid1(부모에게) 에게 SIGUSR1을 보낸다
    kill( pid1, SIGUSR1 );
    for( ;; );
    /* loop forever */
  }
  else {
    /*
    * This waits for ANY child to die. It doesn't matter if the child
    * dies normally or from a signal. The status information is then
    * stored in the status integer.
    *
    * If you want to wait on a particular process use waitpid():
    * waitpid( childPID, &status, 0 );
    * is the common usage.
    *
    * Solaris acts weirdly when a signal is given to the parent process.
    * Therefore we place the wait() inside a while loop so that wait()
    * will not return before the child has died. */
    /* while( (wait( &status ) == -1) && (errno == EINTR) ) {} */
```

```
wait(&status);

/*
* The information in status is *NOT* the return code!! To make use
* of the information we must macros to extract the needed
* information.
*/
/* WIFEXITED() determines if the process exited normally (returned a
* number). This can be done through a return or exit()
*/

if( WIFEXITED( status ) )
{//자식이 정상적으로 종료 되었다면 Non-zero를 반환
  /*
  * Now we know the process exited properly so we can get the
  * return value
  *
  * Note that WEXITSTATUS only retuns the lower 8 bits! That means
  * that if we ever expect a negative number then we have to count
  * the 8th bit as a sign bit.
  */

  exit_status = WEXITSTATUS( status );
  //정상 종료 되었을 때의 반환을 넣어준다.
  /*
  * Since we expect negative numbers...
  *
  * If the exit_status is greater than 2^7 (128), th? the eigth bit
  * is a 1, so we subtract 2^8 (256) from it to make it look like
  * a negative number. */
  if( exit_status > 128 )
  {
    exit_status -= 256;
  }
  printf( "Child return - %d\n", WEXITSTATUS( status ) );
}
else
{
  /* Well it didn't exit properly. Was it a signal? */
  if( WIFSIGNALED( status ) )
  {
    /*
    * Yes. A signal killed the child process. Now we can extract
    * the signal information from status
    */
    printf( "Child died on signal - %d\n", WTERMSIG( status ));
  }
}
```

```
    /*
    * There are two other macros most UNIXes use. They are:
    *WIFSTOPPED() and WSTOPSIG(). See the man pages on the dells for
    *more information.
    *To wait on a particular pid - see waitpid()
    */
  }
  return 0;
}



static void signal_handler(int signo)
{
  /* signo contains the signal number that was received */
  //매개변수 signo에 시그널 넘버가 담겨서 온다.
  switch( signo )
  {
    /* Signal is a SIGUSR1 */
    case SIGUSR1:
      //SIGUSR1의 시그널을 받은 프로세스의 pid를 출력한다.
      printf( "Process %d: received SIGUSR1 \n", getpid() );

      if(pid1==getpid()) /* it is the parent */
      {//부모일 때 이 조건문에 들어오게 된다.
        printf( "Process %d is passing SIGUSR1 to %d...\n", getpid(),pid2
);
        //pid2에게 SIGUSR1을 보낸다.
        kill( pid2, SIGUSR1 );
      }
      else /* it is the child */
      {//자식일 때 이 조건문에 들어오게 된다.
        printf( "Process %d is passing SIGUSR2 to itself...\n", getpid());
        //자신에게 SIGUSR2를 보낸다.
        kill(getpid(), SIGUSR2);
      }
      break;

      /* It's a SIGUSR2 */
    case SIGUSR2:
      printf( "Process %d: received SIGUSR2 \n", getpid() );
      if(pid1==getpid())
      {//부모일 때 이 조건문에 들어오게 된다
        printf( "Process %d is passing SIGUSR2 to %d...\n", getpid(),pid2
);
        kill( pid2, SIGUSR2 );
        //자식에게 SIGUSR2를 보낸다
      }
```

```
      else /* it is the child */
      {//자식일 때 이 조건문에 들어오게 된다
        exit(1);
        printf( "Process %d will terminate itself using SIGINT\n",
getpid());
        kill(getpid(), SIGINT);
        //자신에게 SIGINT를 보내게 된다.
      }
      break;
    default: break;
    }
  return;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
// This function will handle a signal.
void HandleSignal(int sig, siginfo_t *si, void *context);

int main(int argc, char *argv[])
{
  struct sigaction sVal;
  pid_t myPID;
  pid_t myG_PID;
  // Specify that we will use a signal handler that takes three arguments
  // instead of one, which is the default.
  sVal.sa_flags = SA_SIGINFO;

  // Indicate which function is the signal handler.
  sVal.sa_sigaction = HandleSignal;

  myPID = getpid();
  myG_PID = getpgid(myPID);
  printf("\nMy process id = %d.\n", myPID);
  printf("My process group id = %d.\n", myG_PID);

  if(fork() == 0) {//자식 프로세스일때.
    myPID = getpid();
    myG_PID = getpgid(myPID);
    printf("\nChild: My process id = %d.\n", myPID);
    printf("Child: My process group id = %d.\n", myG_PID);
    // Create a new process group that contains this process
```

```
    setpgid(0,0); //
    myPID = getpid();
    myG_PID = getpgid(myPID);
    printf("\nChild: My process id = %d.\n", myPID);
    printf("Child: My process group id = %d.\n", myG_PID);
  }
  else
  {
    // Register for SIGINT
    sigaction(SIGINT, &sVal, NULL);
    // Register for SIGCHLD
    sigaction(SIGCHLD, &sVal, NULL);

    myPID = getpid();
    myG_PID = getpgid(myPID);

    printf("\nParent: My process id = %d.\n", myPID);
    printf("Parent: My process group id = %d.\n", myG_PID);
    while(1) {}
  }
  return(0);
}

void HandleSignal(int sig, siginfo_t *si, void *context) {
  switch(sig)
  {
    case SIGINT:
      printf("\nControl-C was pressed: mypid = %d, mypgid = %d\n",getpid(),
getpgid(getpid()));
      _exit(0);
      break;
    case SIGCHLD:
      printf("\nSIGCHLD. mypid = %d, mypgid = %d\n", getpid(),
getpgid(getpid()));
      if(si->si_code == CLD_EXITED || si->si_code == CLD_KILLED) {
        printf("Process %d is done!\n", si->si_pid);
      }
        break;
  }
}
```

#2

0.ALP textbook 실습

- 스레드 함수는 void * 유형의 단일 매개 변수를 사용하고 void * return 유형을가집니다.
- 마찬가지로 프로그램은 반환 값을 사용하여 기존 스레드의 데이터를 작성자에게 다시 전달할 수 있습니다.

1. pthread_t 변수에 대한 포인터. 새 스레드의 스레드 ID는
2. 스레드 속성 객체에 대한 포인터.이 객체는 스레드가 프로그램의 다른 부분과 상호 작용하는 방법에 대한 세부 사항을 제어합니다. 스레드 속성으로 NULL을 전달하면 기본 스레드 속성으로 스레드가 생성됩니다.
3. 스레드 함수에 대한 포인터.이 형식의 일반 함수 포인터입니다. void * (*) (void *)
4. void * 유형의 스레드 인수 값. 당신이 무엇을 건지는 것은 단순히 thread가 실행을 개시 할 때의 thread 함수의 인수

pthread_create에 대한 호출이 즉시 반환되고 원래 스레드는 호출에 이어 명령어를 계속 실행합니다. 한편 새 스레드는 스레드 함수를 실행하기 시작합니다. Linux는 두 스레드를 비동기 적으로 스케줄하며 프로그램은 두 스레드에서 명령어가 실행되는 상대 순서에 의존하지 않아야합니다.

```
#include <pthread.h>
#include <stdio.h>
/* Parameters to print_function.  */
struct char_print_parms
{
  /* The character to print.  */
  char character;
  /* The number of times to print it.  */
  int count;
};
/* Prints a number of characters to stderr, as given by PARAMETERS,
   which is a pointer to a struct char_print_parms.  */
void* char_print (void* parameters)
{
  /* Cast the cookie pointer to the right type.  */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;
  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}
/* The main program.  */
int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
  /* Create a new thread to print 30000 x's.  */
  thread1_args.character = 'x';
```

```
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
  /* Create a new thread to print 20000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);

  /* Now we can safely return.  */
  return 0;
}
```

pthread_join에게 건네주는 2 번째의 인수가 null가 아닌 경우, thread의 반환 값은 그 인수가 가리키는 위치에 배치됩니다. thread의 반환 값은 thread 인수와 같이 void * 형입니다.

하나의 int 나 다른 작은 숫자를 돌려 주려면 값을 void *로 캐스팅 한 다음 pthread_join.1을 호출 한 후 해당 유형으로 다시 캐스팅하면됩니다. 목록 4.4의 프로그램은 별도의 스레드에서 n 번째 소수를 계산합니다. 이 스레드는 원하는 소수를 스레드 반환 값으로 반환합니다. 반면에 주 스레드는 다른 코드를 자유롭게 실행할 수 있습니다. compute_prime에 사용 된 연속 나누기 알고리즘은 매우 비효율적입니다.

```
#include <pthread.h>
#include <stdio.h>
/* Compute successive prime numbers (very inefficiently).  Return the
   Nth prime number, where N is the value pointed to by *ARG.  */
void* compute_prime (void* arg)
{
  int candidate = 2;
  int n = *((int*) arg);

  while (1) {
    int factor;
    int is_prime = 1;
    /* Test primality by successive division.  */
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
        is_prime = 0;
        break;
      }
    /* Is this the prime number we're looking for?  */
    if (is_prime) {
      if (--n == 0)
        /* Return the desired prime number as the thread return value.  */
```

```
          return (void*) candidate;
        }
        ++candidate;
      }
      return NULL;
    }

    int main ()
    {
      pthread_t thread;
      int which_prime = 5000;
      int prime;

      /* Start the computing thread, up to the 5000th prime number.  */
      pthread_create (&thread, NULL, &compute_prime, &which_prime);
      /* Do some other work here...  */
      /* Wait for the prime number thread to complete, and get the result.  */
      pthread_join (thread, (void*) &prime);
      /* Print the largest prime it computed.  */
      printf("The %dth prime number is %d.\n", which_prime, prime);
      return 0;
    }
```

1. Thread Attributes

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int
*detachstate);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct
sched_param *param); int pthread_attr_getschedparam(const pthread_attr_t
*attr, struct sched_param *param); int
pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
int pthread_attr_setstacksize(pthread_attr_t *attr, int scope);
int pthread_attr_getstacksize(const pthread_attr_t *attr, int *scope);
```

```
#include <stdio.h>
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void *thread_function(void *arg);
char message[] = "Hello World"; int thread_finished = 0;
int main()
{
    int res;
    pthread_t a_thread;
    pthread_attr_t thread_attr;
    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached attribute failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, &thread_attr, thread_function,
(void)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    (void)pthread_attr_destroy(&thread_attr);
    while(!thread_finished) {
        printf("Waiting for thread to say it's finished...\n");
        sleep(1);
    }
    printf("Other thread finished, bye!\n");
    exit(EXIT_SUCCESS);
}
void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(4);
    printf("Second thread setting finished flag, and exiting now\n");
    thread_finished = 1;
    pthread_exit(NULL);
}
```

2.Threads in Abundance

```c
#include <stdio.h>
#include <unistd.h>
```

```c
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 6

void *thread_function(void *arg);
int main() {
    int res;
    pthread_t a_thread[NUM_THREADS];
    void *thread_result;
    int lots_of_threads;
    for(lots_of_threads = 0; lots_of_threads < NUM_THREADS;
lots_of_threads++)
    {
        res = pthread_create(&(a_thread[lots_of_threads]), NULL,
thread_function, (void*)&lots_of_threads);
        if (res != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
    sleep(1);
    }
    printf("Waiting for threads to finish...\n");
    for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0;
lots_of_threads--)     {
        res = pthread_join(a_thread[lots_of_threads], &thread_result);
        if (res == 0) {
            printf("Picked up a thread\n");
        }
        else {
            perror("pthread_join failed");
        }
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int my_number = *(int *)arg;
    int rand_num;

    printf("thread_function is running. Argument was %d\n", my_number);

    rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));

    sleep(rand_num);

    printf("Bye from %d\n", my_number);

    pthread_exit(NULL);
```

```
}

미묘한 버그 ?
```

```
/*슬립을 사용하지 않고 프로그램을 실행하려고하면 동일한 인수로 시작되는 일부 스레드를 비롯한 이상한 결과가 나타날 수 있습니다. 이것이 일어날 수있는 이유를 발견 했습니까? 스레드는 스레드 함수에 대한 인수에 대한 지역 변수를 사용하여 시작됩니다. 이 변수는 루프에서 업데이트됩니다. 문제가되는 행은 다음과 같습니다.*/
for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++)
{
    res = pthread_create(&(a_thread[lots_of_threads]),
NULL,thread_function, (void *)&lots_of_threads);
}

/*주 스레드가 충분히 빠르게 실행되면 일부 스레드의 인수 (lots_of_threads)가 변경 될 수 있습니다. 이와 같은 동작은 공유 변수 및 다중 실행 경로로주의를 기울이지 않을 때 발생합니다. 문제를 해결하려면 다음과 같이 값을 직접 전달해야합니다.*/

res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function,
(void *)lots_of_threads);


void *thread_function(void *arg) { int my_number = (int)arg;
```

3. Synchronization with Mutexes

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr); int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
```

```c
void *thread_function(void *arg);
pthread_mutex_t work_mutex; /* protects both work_area and time_to_exit */

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;


int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&work_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    pthread_mutex_lock(&work_mutex);
    printf("Input some text. Enter 'end' to finish\n");
    while(!time_to_exit)
    {
        fgets(work_area, WORK_SIZE, stdin);

        pthread_mutex_unlock(&work_mutex);

        while(1) {

            pthread_mutex_lock(&work_mutex);
            if (work_area[0] != '\0') {
                pthread_mutex_unlock(&work_mutex);
                sleep(1);
            }
            else {
                break;
            }
        }
    }
    pthread_mutex_unlock(&work_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);

    if (res != 0) {
```

```
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }

    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

Semaphore

세마포어 함수는 대부분의 스레드 특정 함수처럼 sem*와 함께 pthread*로 시작하지 않습니다. 스레드에서 사용되는 네 가지 기본 세마포어 함수가 있습니다. 그들은 모두 아주 간단합니다. 세마포어는 다음과 같이 선언 된 sem_init 함수로 생성됩니다.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem); int sem_post(sem_t * sem);
int sem_destroy(sem_t * sem);
```

```
#include <stdio.h>
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = sem_init(&bin_sem, 0, 0);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Input some text. Enter 'end' to finish\n");

    while(strncmp("end", work_area, 3) != 0) {
        fgets(work_area, WORK_SIZE, stdin);
        sem_post(&bin_sem);
        }
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}
void *thread_function(void *arg) {
    sleep(1);
    sem_wait(&bin_sem);
    while(strncmp("end", work_area, 3) != 0)
    {
        printf("You input %d characters\n", strlen(work_area) -1);
        sem_wait(&bin_sem);
    }
    pthread_exit(NULL);
```

```
}
```

우리가 세마포어를 초기화 할 때 그 값을 0으로 설정한다.

따라서 threads 함수가 시작될 때 sem_wait를 호출하면 세마포어가 블록되지 않고 0이 될 때까지 기다린다.

주 스레드에서는 텍스트가 생길 때까지 기다린 다음 sem_post로 세마포어를 증가시킵니다.

그러면 즉시 다른 스레드가 해당 sem_wait에서 돌아가 실행을 시작할 수 있습니다.

문자 수를 계산하면 다시 sem_wait를 호출하고 주 스레드가 sem_post를 다시 호출하여 세마포어를 증가시킬 때까지 차단됩니다.

일반적인 문제의 원인은 미묘한 타이밍 오류를 간과하는 것입니다.

프로그램을 thread4a.c로 약간 수정하여 키보드의 텍스트 입력이 언젠가 자동으로 사용 가능한 텍스트로 대체되는 것처럼 가장하십시오.

우리는 메인의 읽기 루프를 다음과 같이 수정합니다

```
printf("Input some text. Enter 'end' to finish\n");
while(strncmp("end", work_area, 3) != 0) {
    if (strncmp(work_area, "FAST", 4) == 0) {
        sem_post(&bin_sem);
        strcpy(work_area, "Wheeee...");
    } else {
        fgets(work_area, WORK_SIZE, stdin);
    }
sem_post(&bin_sem);
}
/*
이제 우리가 'FAST'를 입력하면 프로그램은 sem_post를 호출하여 문자 카운터가 실행되도록하지만
work_area를 다른 것으로 즉시 업데이트합니다.*/
```

Shared Memory

- shmctl ("Shared Memory ConTroL") 호출은 공유 메모리 세그먼트에 대한 정보를 반환하고 수정할 수 있습니다
- 첫 번째 매개 변수는 공유 메모리 세그먼트 식별자입니다
- 공유 메모리 세그먼트에 대한 정보를 얻으려면 IPC_STAT를 두 번째 인수로 전달하고 struct shmid_ds에 대한 포인터를 전달하십시오.
- 세그먼트를 제거하려면 IPC_RMID를 두 번째 인수로 전달하고 세 번째 인수로 NULL을 전달하십시오. 세그먼트가 연결된 마지막 프로세스가 마지막으로 분리하면 세그먼트가 제거됩니다.

- 공유 메모리 세그먼트의 총 수에 대한 시스템 전체 제한을 위반하지 않도록 각 공유 메모리 세그먼트는 shmctl을 사용하여 명시 적으로 할당을 해제해야합니다.
- exit 및 exec를 호출하면 메모리 세그먼트가 분리되지만 할당을 해제하지는 않습니다.

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main () {
    int segment_id;
    char* shared_memory;
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = 0x6400;

    /* Allocate a shared memory segment. */
    segment_id = shmget (IPC_PRIVATE, shared_segment_size,IPC_CREAT |
IPC_EXCL | S_IRUSR | S_IWUSR);

    /* Attach the shared memory segment. */
    shared_memory = (char*) shmat (segment_id, 0, 0);
    printf ("shared memory attached at address %p\n", shared_memory);

    /* Determine the segment's size. */
    shmctl (segment_id, IPC_STAT, &shmbuffer);
    segment_size = shmbuffer.shm_segsz;
    printf ("segment size: %d\n", segment_size);

    /* Write a string to the shared memory segment. */
    sprintf (shared_memory, "Hello, world.");

    /* Detach the shared memory segment. */
    shmdt (shared_memory);

    /* Reattach the shared memory segment, at a different address. */
shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
    printf ("shared memory reattached at address %p\n",
shared_memory);
    /*Print out the string from shared memory. */
    printf ("%s\n", shared_memory);

    /* Detach the shared memory segment. */
    shmdt (shared_memory);

    /* Deallocate the shared memory segment. */
    shmctl (segment_id, IPC_RMID, 0);
return 0;
}
```

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
/* Wait on a binary semaphore. Block until the semaphore value is positive,
then decrement it by 1. */
int binary_semaphore_wait (int semid){
    struct sembuf operations[1];

    /* Use the first (and only) semaphore. */
    operations[0].sem_num = 0;

    /* Decrement by 1. */
    operations[0].sem_op = -1;

    /* Permit undo'ing. */
    operations[0].sem_flg = SEM_UNDO;
    return semop (semid, operations, 1);
}
/* Post to a binary semaphore: increment its value by 1. This returns
immediately. */
int binary_semaphore_post (int semid) {
    struct sembuf operations[1];
    /* Use the first (and only) semaphore. */
    operations[0].sem_num = 0;

    /* Increment by 1. */
    operations[0].sem_op = 1;
    /* Permit undo'ing. */
    operations[0].sem_flg = SEM_UNDO;
    return semop (semid, operations, 1);
}
```

- sem_num : 세마푸어 세트의 세마푸어 번호
- sem_op : 세마포어 연산을 지정하는 정수입니다. sem_op가 양수인 경우 해당 숫자는 즉시 세마포 값에 추가됩니다. sem_op이 음수이면, 해당 숫자의 절대 값이 세마포어 값에서 뺍니다. 이렇게하면 세마포어 값이 음수가됩니다. 세마포어 값이 sem_op의 절대 값만큼 커질 때까지 호출이 차단됩니다 (다른 프로세스에서이 값을 증가시키기 때문입니다). sem_op이 0이면 세마포어 값이 0이 될 때까지 연산이 차단됩니다.
- sem_flg : 플래그 값입니다. IPC_NOWAIT를 지정하면 작업이 차단되지 않습니다. 작업이 차단 된 경우 semop을 호출하면 대신 실패합니다. SEM_UNDO를 지정하면 프로세스가 종료 될 때마다 Linux가 자동으로 세마포어의 작업을 취소합니다.

Mapped Memory

MAP_FIXED—이 플래그를 지정하면, Linux는 힌트로 취급하지 않고 파일을 맵핑하기 위해 요청한 주소를 사용합니다.이 주소는 페이지 정렬되어야합니다.

n MAP_PRIVATE— 메모리 범위에 대한 쓰기는 첨부 된 파일에 다시 기록하지 말고 파일의 개인 사본에 기록해야합니다. 다른 프로세스는 이러한 기록을 보지 않습니다. 이 모드는 MAP_SHARED와 함께 사용할 수 없습니다.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
/* Return a uniformly random number in the range [low,high]. */
int random_range (unsigned const low, unsigned const high) {
    unsigned const range = high - low + 1;
    return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}
int main (int argc, char* const argv[]) {
    int fd;
    void* file_memory;
/* Seed the random number generator. */
    srand (time (NULL));
/* Prepare a file large enough to hold an unsigned integer. */
    fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    lseek (fd, FILE_LENGTH+1, SEEK_SET);
    write (fd, "", 1);
    lseek (fd, 0, SEEK_SET);
    /* Create the memory mapping. */
    file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
    close (fd);
    /* Write a random integer to memory-mapped area. */
    sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
    /* Release the memory (unnecessary because the program exits). */
    munmap (file_memory, FILE_LENGTH);

    return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
int main (int argc, char* const argv[]) {
    int fd;
    void* file_memory;
    int integer;
    /* Open the file. */
    fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
    /* Create the memory mapping. */
    file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,MAP_SHARED,
fd, 0);
    close (fd);
    /* Read the integer, print it out, and double it. */
    scanf (file_memory, "%d", &integer);
    printf ("value: %d\n", integer);
    sprintf ((char*) file_memory, "%d\n", 2 * integer);
    /* Release the memory (unnecessary because the program exits). */
    munmap (file_memory, FILE_LENGTH);
    return 0;
}

msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

검증이 되면 씀

```
int pipe_fds[2];
int read_fd;
int write_fd;
pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];

int main () {
    int fds[2];
    pid_t pid;
    /* Create a pipe. File descriptors for the two ends of the pipe are
placed in fds. */
    pipe (fds);
    /* Fork a child process. */
    pid = fork ();
    if (pid == (pid_t) 0) {
        FILE* stream;
```

```
        /* This is the child process. Close our copy of the write end of
        the file descriptor. */
        close (fds[1]);
        /* Convert the read file descriptor to a FILE object, and read
        from it. */
        stream = fdopen (fds[0], "r");
        reader (stream);
        close (fds[0]);
    }
    else {
        /* This is the parent process. */
        FILE* stream;
        /* Close our copy of the read end of the file descriptor. */
        close (fds[0]);
        /* Convert the write file descriptor to a FILE object, and write
    to it. */
        stream = fdopen (fds[1], "w");
        writer ("Hello, world.", 5, stream);
        close (fds[1]);
    }
    return 0;
}
```

socket—Creates a socket closes—Destroys a socket connect—Creates a connection between two sockets

bind—Labels a server socket with an address listen—Configures a socket to accept conditions accept—Accepts a connection and creates a new socket for the connection

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
/* Read text from the socket and print it out. Continue until the socket
closes. Return nonzero if the client sent a "quit" message, zero otherwise.
*/
int server (int client_socket) {
    while (1) {
        int length; char* text;
    /* First, read the length of the text message from the socket. If read
returns zero, the client closed the connection. */
```

```
        if (read (client_socket, &length, sizeof (length)) == 0)
            return 0;
        /* Allocate a buffer to hold the text. */
        text = (char*) malloc (length);
        /* Read the text itself, and print it. */
        read (client_socket, text, length);
        printf ("%s\n", text);

    /* Free the buffer. */
        free (text);
    /* If the client sent the message "quit," we're all done. */
        if (!strcmp (text, "quit"))
            return 1;
    }
}
int main (int argc, char* const argv[]) {
    const char* const socket_name = argv[1];
    int  socket_fd;
    struct sockaddr_un name;

    int client_sent_quit_message;

    /* Create the socket. */
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
    /* Indicate that this is a server. */
    name.sun_family = AF_LOCAL;
    strcpy (name.sun_path, socket_name);
    bind (socket_fd, &name, SUN_LEN (&name));
    /* Listen for connections. */

    listen (socket_fd, 5);
    /* Repeatedly accept connections, spinning off one server() to deal
    with each client. Continue until a client sends a "quit" message. */

    do {

        struct sockaddr_un client_name;
        socklen_t client_name_len;

        int client_socket_fd;
        /* Accept a connection. */
        client_socket_fd = accept (socket_fd, &client_name,
&client_name_len);

        /* Handle the connection. */
        client_sent_quit_message = server (client_socket_fd);
        /* Close our end of the connection. */
        close (client_socket_fd);
    }while (!client_sent_quit_message);
```

```
    /* Remove the socket file. */

    close (socket_fd);
    unlink (socket_name);
    return 0;
}
```

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Write TEXT to the socket given by file descriptor SOCKET_FD. */
void write_text (int socket_fd, const char* text)
{
    /* Write the number of bytes in the string, including NUL-termination.
*/
    int length = strlen (text) + 1;
    write (socket_fd, &length, sizeof (length)); /* Write the string. */
    write (socket_fd, text, length);
}
int main (int argc, char* const argv[]) {
    const char* const socket_name = argv[1];
    const char* const message = argv[2];
    int socket_fd;
    struct sockaddr_un name;
    /* Create the socket. */
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
    /* Store the server's name in the socket address. */
    name.sun_family = AF_LOCAL;
    strcpy (name.sun_path, socket_name);
    /* Connect the socket. */
    connect (socket_fd, &name, SUN_LEN (&name));
    /* Write the text on the command line to the socket. */
    write_text (socket_fd, message);
    close (socket_fd);
    return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
/* Print the contents of the home page for the server's socket. Return an
indication of success. */
void get_home_page (int socket_fd) {
    char buffer[10000];
    ssize_t number_characters_read;
    /* Send the HTTP GET command for the home page. */
    sprintf (buffer, "GET /\n");
    write (socket_fd, buffer, strlen (buffer));
    /* Read from the socket. The call to read may not return all the data
at one time, so keep trying until we run out. */
    while (1) {
        number_characters_read = read (socket_fd, buffer, 10000);
        if (number_characters_read == 0)
            return;
/* Write the data to standard output. */
        fwrite (buffer, sizeof (char), number_characters_read, stdout);
    }
}
int main (int argc, char* const argv[]) {
    int socket_fd;
    struct sockaddr_in name;
    struct hostent* hostinfo;
    /* Create the socket. */
    socket_fd = socket (PF_INET, SOCK_STREAM, 0);
    /* Store the server's name in the socket address. */
    name.sin_family = AF_INET;
    /* Convert from strings to numbers. */
    hostinfo = gethostbyname (argv[1]);
    if (hostinfo == NULL)
        return 1;
    else
        name.sin_addr = *((struct in_addr *) hostinfo->h_addr); /* Web
servers use port 80. */
    name.sin_port = htons (80);
    /* Connect to the Web server */
    if (connect (socket_fd, &name, sizeof (struct sockaddr_in)) == -1)
    {
        perror ("connect");
        return 1;
```

```
    }
    /* Retrieve the server's home page. */
    get_home_page (socket_fd);
    return 0;
}
```