#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.

#2

0.ALP textbook 실습

```
#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;
}
```

```
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 30,000 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 20,000 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;
}
```

rjdjdqrp

```
#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;
```

```c
}

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;
}
```