1. asprintf() takes an unallocated char\* pointer as input parameter. After invoking asprintf(), the input pointer is allocated and the formatted string is printed into it. asprintf() takes a pointer to a pointer because it has a integer as the return value, then the string must be passed to asprintf() as a pointer in order to make changes to it.

Why: asprintf() is safer than sprint(), since it does not have the problem of buffer overflow.

How: See the example below.

#include <stdio.h>

int main()

{

char \*buffer;

int r;

r = asprintf(&buffer,"The total is %d\n",5+8);

puts(buffer);

printf("%d characters generated\n",r);

free(buffer);

return(0);

}

Note that the pointer as the first parameter of asprintf() must be freed if not used anymore.

2. When pthread\_join() is called, the thread calling it will wait until the thread as the first parameter of pthread\_join() to exit. When pthread\_exit() is called, the thread calling it will be forced to exit. Thus, they do totally different things.

![手机屏幕截图

描述已自动生成]()3. fork() creates a child process. The child process execute the same code with the parent process. We usually use the pid to control the work flow of parent process and child process.

Calling exec() will replace all the code in the process by the codes specified by the input parameter. In the following example, the printed message would be “hello world” but not “test here”. Thus in the child process we can use exec() function to execute some other codes. But remember to use pid to control it.

exec\_main.c

#include <unistd.h>

#include <stdio.h>

int main(void){

char\* const argv[]={“exec\_call”,NULL};

execv(“./exec\_call”,argv);

printf(“test here\n”);

return 0;

}

exec\_call.c

#include <stdio.h>

int main(void){

printf(“hello world\n”);

return 0;

}

Calling wait() function will block the process until the process being waited has a state change.

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

int main(){

pc=fork();

if(pc==0){

printf("This is child process with pid of %d\n",getpid());

}

Else{

pr=wait(NULL);

printf("I catched a child process with pid of %d\n"),pr);

}

printf(“hello1\n”);

}

4. (1) Find the unfreed pointers and free them. (2) check double deallocation. (3) Check the functions that uses a non-constant pointer as input.

5. When a program no longer requires certain blocks of memory, these blocks of memory can be freed, and this generates separated free memory blocks. Coalescing is a management in which two adjacent free blocks of memory are merged. The Knuth’ boundary tag is the block’s size stored at the end of the block. As the blocks are contiguous, the end of one block sits right next to the start of the next block. So the current block (apart from the first one) can look a few bytes further back to look up the size of the previous block. With this information, the allocator can now jump backward.

6. Condition variable is to avoid the unnecessary overhead for waiting and blocking. Without the condition variable, a thread must spin on waiting for a mutex to be released.

Mutex locks are not versatile since sometimes mutex locks brings the deadlock problem. We can use mutexes, condition variables, and other tools to design efficient programs.

7. The deadlock problem arises when two threads each acquires a lock and requires the lock of the other thread. The following codes give an example.

#include <unistd.h>

#include <pthread.h>

#include <string.h>

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutex2 = PTHREAD\_MUTEX\_INITIALIZER;

static int sequence1 = 0;

static int sequence2 = 0;

int func1() {

pthread\_mutex\_lock(&mutex1);

++sequence1;

sleep(1);

pthread\_mutex\_lock(&mutex2);

++sequence2;

pthread\_mutex\_unlock(&mutex2);

pthread\_mutex\_unlock(&mutex1);

return sequence1;

}

int func2() {

pthread\_mutex\_lock(&mutex2);

++sequence2;

sleep(1);

pthread\_mutex\_lock(&mutex1);

++sequence1;

pthread\_mutex\_unlock(&mutex1);

pthread\_mutex\_unlock(&mutex2);

return sequence2;

}

void\* thread1(void\* arg) {

while (1) {

int iRetValue = func1();

if (iRetValue == 100000)

pthread\_exit(NULL);

}

}

void\* thread2(void\* arg) {

while (1) {

int iRetValue = func2();

if (iRetValue == 100000)

pthread\_exit(NULL);

}

}

int main() {

pthread\_t tid[4];

if (pthread\_create(&tid[0], NULL, &thread1, NULL) != 0) {

\_exit(1);

}

if (pthread\_create(&tid[1], NULL, &thread2, NULL) != 0) {

\_exit(1);

}

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

pthread\_mutex\_destroy(&mutex1);

pthread\_mutex\_destroy(&mutex2);

return 0;

}

8. The reader-writer problem is to let multiple readers and a single writer to work together on a variable. The readers can read together, but the writer is exclusive. The basic idea to solve the reader-writer problem is to use several counters to track the number of working readers and writers, and use a mutex to protect the counters and the shared data. Another important point is the use of conditional wait, which ensures that only one thread can wake and execute reading or writing.

9. Mutual Exclusion means that when multiple processes wants to access the same data, at one time at most one process can actually manipulate the data.

The meaning of Progress is as follows. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded waiting. There exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted. In another way to say, after several failed attempts to try to enter the critical section, a progress will definitely get the chance to enter the critical section, which avoids starving.

10. 这个要讲favorite 和 most challenging，需要客户自己写一下了。