2024 年秋季操作系统第一次作业

Part1. 进程与线程

1. 支持多道程序设计的操作系统在运行过程中，不断地选择新进程运行来实现 CPU 的共享，但其中（）不是引起操作系统选择新进程的直接原因。

A. 运行进程的时间片用完 D

B. 运行进程出错

C. 运行进程要等待某一事件发生

D. 有新进程被创建进入就绪态

2. 若一个进程实体由 PCB、共享正文段，数据堆段和数据栈段组成，请指出下列 C 语言程序中的内容及相关数据结构各位于哪一段中。

I. 全局赋值变量（data

II. 未赋值的局部变量（D

III. 函数调用实参传递值（）D

IV. 用 malloc()要求动态分配的存储区（）C

V. 常量值（如 1995、"string"）（）rodata

VI. 进程的优先级（）A

A. PCB

B. 正文段

C. 堆段

D. 栈段

3. 以下可能导致一个进程从运行态变为就绪态的事件是（）。

A. 一次 1/0 操作结束 D

C. 运行进程结束

B. 运行进程需做 I/0 操作

D. 出现了比现在进程优先级更高的进程

4. 下列选项中，降低进程优先级的合理时机是（）。 A

A. 进程时间片用完

B. 进程刚完成 1/O 操作，进入就绪队列

C. 进程长期处于就绪队列

D. 进程从就绪态转为运行态

5. 根据进程图，以下哪个输出是不正确的（） D

A. L0, Bye, L1, Bye, L2, Bye

B. L0, L1, Bye, L2, Bye, Bye

C. L0, Bye, L1, L2, Bye, Bye

D. L0, Bye, L1, Bye, Bye, L2

6. 下列关于线程的描述中，错误的是()。

A. 内核级线程的调度由操作系统完成 ✓

B. 用户级线程间的切换比内核级线程间的切换效率高 不确定

C. 用户级线程可以在不支持内核级线程的操作系统上实现 ✓ (协程? yield/resume)

D. 操作系统为每个用户级线程建立一个线程控制块 Linux下是一对一的

7. 关于线程的以下描述中，哪一个是正确的（）

A. 线程共享进程的地址空间和资源 ✓ A

B. 每个线程用有独立的进程上下文

C. 线程的切换比进程的切换代价更大

D. 线程是独立于进程的执行单元

1. 分析程序 homework_wait.c，回答下列问题：

```
1.  /* homework_wait.c */
2.  void homework_wait() {
3.      pid_t pid[N];
4.      int i, child_status;
5.      for (i = 0; i < N; i++) {
6.          if ((pid[i] = fork()) == 0) {
7.              exit(100+i); /* Child */
8.          }
9.      }
10.     printf("hello!\n");
11.     for (i = 0; i < N; i++) { /* Parent */
12.         pid_t wpid = wait(&child_status);
13.         if (WIFEXITED(child_status))
14.             printf("Child %d terminated with exit status %d\n",
15.                    wpid, WEXITSTATUS(child_status));
16.         else
17.             printf("Child %d terminate abnormally\n", wpid);
18.     }
19. }
```

1) 注释掉第 7 行代码后，程序执行到第 10 行，输出多少个 “hello!”（用一个 N 的函数给出答案）？ N+1

2) N=2 时，程序正常运行两次，得到的结果是否相同？若不同，请解释原因；可能不相同，子进程 exit 时机和父进程 wait 时机，可能有不同的调度

3) 修改程序，使得子进程能够按照其创建的顺序退出。

2. 在以下程序中，会创建一个新进程，然后父进程会等待该进程终止。之后，父进程会创建一个新进程并重复整个过程。

修改此代码以创建两个不同的进程（并行执行），父进程会等待两个进程的终

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

#define NULL ((void *)0)

#define N 5

pid_t pid[N];

void handle_sigusr1(int sig) {
  // do nothing
}

void homework_wait() {
  struct sigaction sa;
  sa.sa_handler = handle_sigusr1;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  if (sigaction(SIGUSR1, &sa, NULL) == -1) {
    perror("sigaction");
    exit(EXIT_FAILURE);
  }

  for (int i = 0; i < N; i++) {
    if ((pid[i] = fork()) == 0) {
      pause();
      exit(100 + i);
    } else if (pid[i] < 0) {
      perror("fork");
      exit(EXIT_FAILURE);
    }
  }

  printf("hello!\n");

  for (int i = 0; i < N; i++) {
    if (kill(pid[i], SIGUSR1) == -1) {
      perror("kill");
      exit(EXIT_FAILURE);
    }

    int child_status;
    pid_t wpid = waitpid(pid[i], &child_status, 0);
    if (wpid == -1) {
      perror("waitpid");
      exit(EXIT_FAILURE);
    }

    if (WIFEXITED(child_status)) {
      printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
    } else {
      printf("Child %d terminated abnormally\n", wpid);
    }
  }
}

int main() {
  homework_wait();
  return 0;
}
```

止。

```

#include ‹sys/types.h›

#include ‹stdio.h›


int main (int argc, char *argv[])

{

    pid_t pid;

    int status;

    pid = fork ();

    if (pid!= 0)

        while (pid != wait (&status));

    else {

        sleep (5);

        exit (5) ;

    }

    pid = fork ();

    if (pid != 0)

        while (pid != wait (&status) );

    else {

        sleep (1);

        exit (1);

    }

}

```

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  pid_t pid;

  pid = fork();
  if (pid < 0) {
    perror("fork");
  } else if (pid == 0) {
    sleep(5);
    exit(5);
  } else {
    pid = fork();
    if (pid < 0) {
      perror("fork");
    } else if (pid == 0) {
      sleep(1);
      exit(1);
    } else {
      while (1) {
        int status;
        pid = wait(&status);
        if (pid == -1) {
          if (errno == ECHILD) {
            break;
          } else {
            perror("wait");
          }
        }
        printf("Child %d terminated with exit status %d\n", pid, WEXITSTATUS(status));
      }
    }
  }
}
```

3. 阅读以下代码

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2, p3;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "I "); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "LIKE "); assert(rc == 0);
    rc = pthread_create(&p3, NULL, mythread, "OS "); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    rc = pthread_join(p3, NULL); assert(rc == 0);
    printf("\nmain: end\n");
    return 0;
}
```

3! = 6

(1) 以上程序运行的输出结果共有几种?

(2) 如何修改上述程序，让其按照"main:start"、"I LIKE OS"、"main:end"的顺序输出。

Part2. 并发与同步

1. 对于两个并发进程，设互斥信号量为 mutex（初值为 1），若 mutex=-1，则（ C ）。

A. 表示没有进程进入临界区

```c
#include <assert.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>

#define NULL ((void *)0)

pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;  // mutex for stdout and I LIKE OS

int I = 0;
int LIKE = 0;
int OS = 0;

#define I_STR "I "
#define LIKE_STR "LIKE "
#define OS_STR "OS "

#define CONDITION
\
  ((strcmp(arg, I_STR) == 0 && (0 == I && 0 == LIKE && 0 == OS)) || (strcmp(arg, LIKE_STR) ==
0 && (1 == I && 0 == LIKE && 0 == OS)) || \
   (strcmp(arg, OS_STR) == 0 && (1 == I && 1 == LIKE && 0 == OS)))

void *mythread(void *arg) {
  pthread_mutex_lock(&mtx);
  while (!CONDITION) {
    pthread_cond_wait(&cv, &mtx);
  }
  assert(CONDITION);

  if (strcmp(arg, I_STR) == 0) I = 1;
  if (strcmp(arg, LIKE_STR) == 0) LIKE = 1;
  if (strcmp(arg, OS_STR) == 0) OS = 1;

  printf("%s\n", (char *)arg);

  pthread_cond_broadcast(&cv);
  pthread_mutex_unlock(&mtx);

  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2, p3;
  int rc;
  printf("main: begin\n");
  rc = pthread_create(&p1, NULL, mythread, I_STR);
  assert(rc == 0);
  rc = pthread_create(&p2, NULL, mythread, LIKE_STR);
  assert(rc == 0);
  rc = pthread_create(&p3, NULL, mythread, OS_STR);
  assert(rc == 0);
  // join waits for the threads to finish
  rc = pthread_join(p1, NULL);
  assert(rc == 0);
  rc = pthread_join(p2, NULL);
  assert(rc == 0);
  rc = pthread_join(p3, NULL);
  assert(rc == 0);
  printf("\nmain: end\n");
  return 0;
}
```

B. 表示有一个进程进入临界区

C. 表示有两个进程进入临界区

D. 表示有一个进程进入临界区，另一个进程在等待进入

2. 有两个优先级相同的并发程序 P1 和 P2，他们的执行过程如下所示。假设当前信号量 s1=0，s2=0。当前的 z=2，进程运行结束之后，x, y, z 的值分别为（ ）。

C

```c
P1: y=1;                    P2: x=1;
    y=y+2;  y=3                 x=x+1;  x=2
    z=y+1;  z=4                 P(s1);
    V(s1);                      x=x+y;  x=2+3=5
    P(s2);                      z=x+z;  z=5+4=9
    y=z+y;  y=9+3=12            v(s2);
```

A. 5, 9, 9

B. 5, 9, 4

C. 5, 12, 9

D. 5, 12, 4

3. 下列准则中，实现临界区互斥机制必须遵循的是（ ）。

D

I. 两个进程不能同时进入临界区

II. 允许进程访问空闲的临界资源

III. 进程等待进入临界区的时间是有限的

IV. 不能进入临界区的执行态进程立即放弃 CPU

A. 仅 I、IV

B. 仅 II、III

C. 仅 I、II、III

D. 仅 I、III、IV

4. 设与某资源关联的信号量初值为 3，当前值为 1．若 M 表示该资源的可用个数，N 表示等待该资源的进程数，则 M,N 分别是（ ）。

0

B

A. 0,1

B. 1,0

c. 1,2

D. 2,0

5. 有三个进程共享同一程序段，而每次只允许两个进程进入该程序段，若用 PV 操作同步机制，则信号量 S 的取值范围是（ ）。

A

A. 2,1,0,-1

B. 3,2,1,0 X

C. 2,1,0,-1,-2

D. 1,0,-1,-2

6. 若一个信号量的初值为 3，经过多次 PV 操作后当前值为-1，这表示等待进入临界区的进程数是（ ）。

C

A

A. 1

B. 2

C. 3

D. 4

7. 信号作为一般进程间通信机制的实用性是有限的，因为 _____

C

A. 它们不能在进程之间工作

B. 它们是由用户生成的

C. 它们不能直接携带信息

D. 以上都不是

1. 请说明如果 P()的信号量操作没有原子执行，那么可能会违反互斥。

2. 说明如何使用二进制信号量（信号量取值为 0 和 1）来实现 n 个进程之间的互斥？

3. 假设你有一个只有二进制信号量的操作系统，但你希望使用计数信号量。请你演示如何使用二进制信号量实现计数信号量的 P_new()和 V_new()操作。提示：可以使用两个二进制信号量，一个用来实现计数互斥，另一个用来实现阻塞。

■ wait (P)操作

```
wait(semaphore *S){
  S->value--;
  if(S->value < 0){
    add this process to S->list;
    block();
  }
}
```

■ signal (V) 操作：

```
signal(semaphore *S){
  S->value++;
  if(S->value <= 0){
    remove a process P from S->list;
    wakeup(P);
  }
}
```

1、如果不是原子的：

上面代码可被写成汇编，假设 load/store 原子

(不一定原子：内存模型、一致性相关)

1 load reg ← value(s)

2 dec reg

3 store reg → value(s)

4 bge reg, zero, 'skip

5 block( )

'skip: 6 临界

有两个线程：a、b，设 s→value = 1

有下列顺序

a1 rega = 1

b1 regb = 1

a2 rega = 0

b2 regb = 0

a3 s→value = 0

b3　　s→value = 0

a4　　rega ≥ 0 ⇒ 进入临界

b4　　regb ≥ 0 ⇒ 进入临界

与预期不符！！

2. 设有下列代码，有两线程 a b

1. P(&S)	a1	b1
2. 临界	a2	b2
3. 临界	a3	b3
4. V(&S)	a4	b4

2) 不意味着不会看到：　a2 b2 a3 b3 这样的代码

① 若 a 先 P(&S) 则有：　a4 ≤ b2

则　　a1 a2 a3 a4

　　b2 b3 b4

→ time

同时 b 先 P(&S) 的话，也不会有

a2 b2 a3 b3 之类的情况的发生

```c
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct {
  pthread_mutex_t mtx;
  int cnt;
} sem_t;

/**
 * @brief
 *
 * @param sem (return)
 * @param value
 * @return int
 */
int sem_init(sem_t **sem, int value) {
  sem_t *sem_p = malloc(sizeof(sem_t));
  if (pthread_mutex_init(&sem_p->mtx, NULL) != 0) {
    perror("pthread_mutex_init");
    return -1;
  }
  sem_p->cnt = value;
  *sem = sem_p;
  return 0;
}

int sem_p(sem_t *sem) {
  if (sem == NULL) {
    errno = EINVAL;
    perror("sem_p: sem is NULL");
    return -1;
  }

  while (1) {
    if (pthread_mutex_lock(&sem->mtx) != 0) {
      perror("pthread_mutex_lock");
      return -1;
    }

    if (sem->cnt > 0) {
      sem->cnt--;
      if (pthread_mutex_unlock(&sem->mtx) != 0) {
        perror("pthread_mutex_unlock");
        return -1;
      }
      break;
    }

    if (pthread_mutex_unlock(&sem->mtx) != 0) {
      perror("pthread_mutex_unlock");
      return -1;
    }

    usleep(1000);
  }

  return 0;
}

int sem_v(sem_t *sem) {
  if (pthread_mutex_lock(&sem->mtx) != 0) {
    perror("pthread_mutex_lock");
    return -1;
  }
  sem->cnt++;
  if (pthread_mutex_unlock(&sem->mtx) != 0) {
    perror("pthread_mutex_unlock");
    return -1;
  }
  return 0;
}

int sem_destroy(sem_t *sem) {
  if (pthread_mutex_destroy(&sem->mtx) != 0) {
    perror("pthread_mutex_destroy");
    return -1;
  }
  free(sem);
  return 0;
}

sem_t *g_sem;

#define RESOURCE_COUNT 3
#define THREAD_COUNT 10

void *thread_func(void *arg) {
  int thread_num = *(int *)arg;
  printf("thread %d trying to acquire resource...\n", thread_num);

  sem_p(g_sem);

  printf("thread %d has gotten resource! sem->cnt=%d\n", thread_num, g_sem->cnt);

  sleep(rand() % 3 + 1);

  printf("thread %d release resource.\n", thread_num);
  sem_v(g_sem);

  free(arg);
  return NULL;
}

int main() {
  pthread_t threads[THREAD_COUNT];
  int res;

  res = sem_init(&g_sem, RESOURCE_COUNT);
  if (res != 0) {
    perror("init failed");
    exit(EXIT_FAILURE);
  }

  for (int i = 0; i < THREAD_COUNT; i++) {
    int *thread_num = malloc(sizeof(int));
    if (thread_num == NULL) {
      perror("malloc failed");
      exit(EXIT_FAILURE);
    }
    *thread_num = i + 1;
    res = pthread_create(&threads[i], NULL, thread_func, thread_num);
    if (res != 0) {
      perror("pthread_create failed");
      exit(EXIT_FAILURE);
    }
  }

  for (int i = 0; i < THREAD_COUNT; i++) {
    pthread_join(threads[i], NULL);
  }

  sem_destroy(g_sem);

  printf("success!\n");
  return 0;
}
```