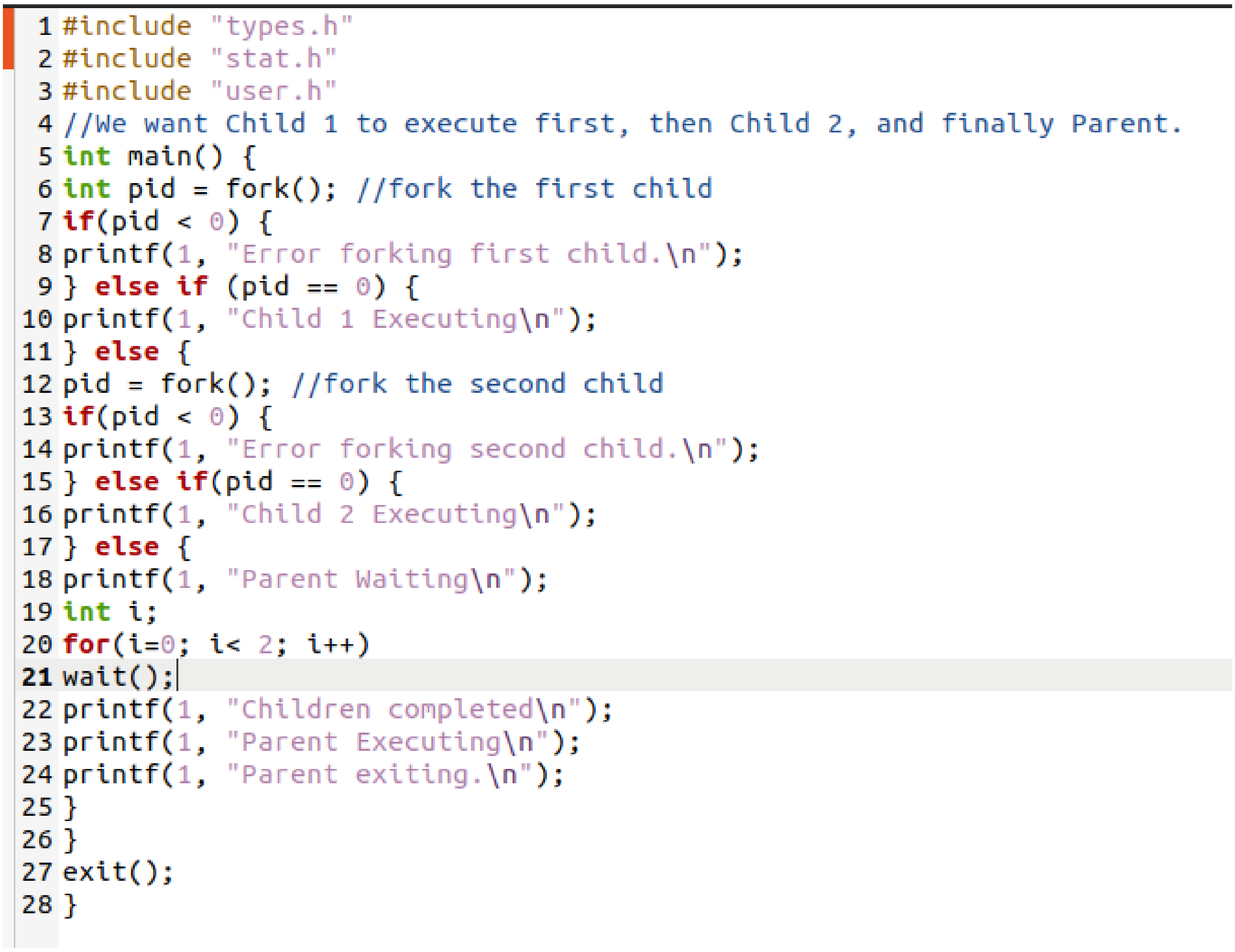
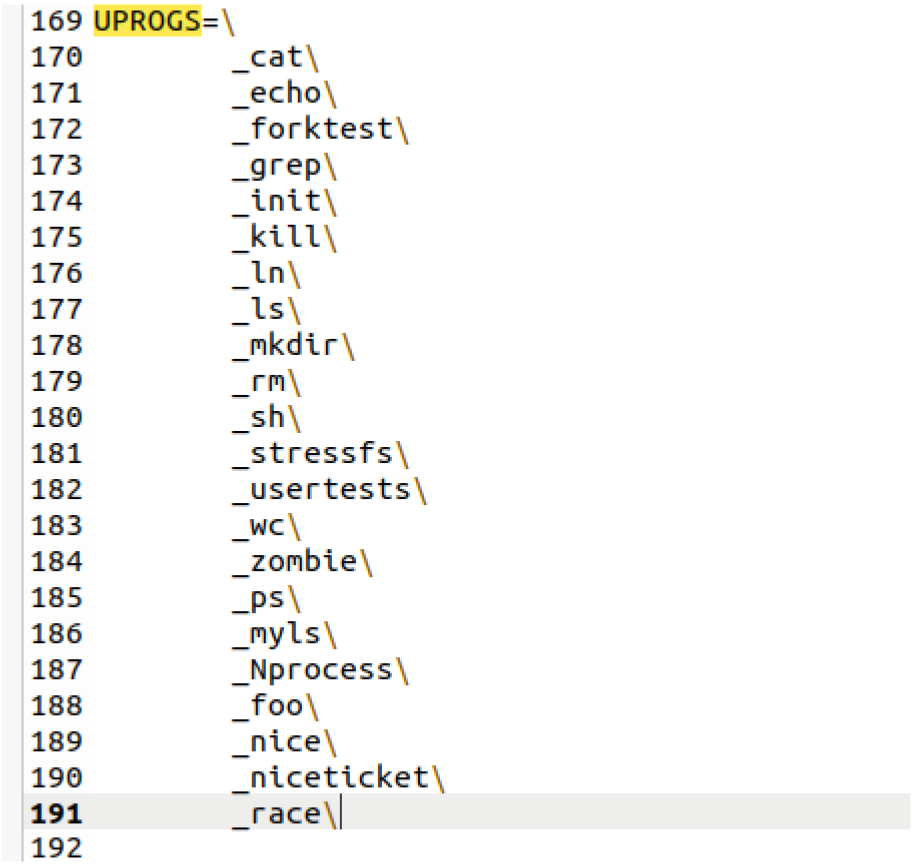
ASSIGNMENT – Process Synchronization

First create ‘race.c’ with following code:



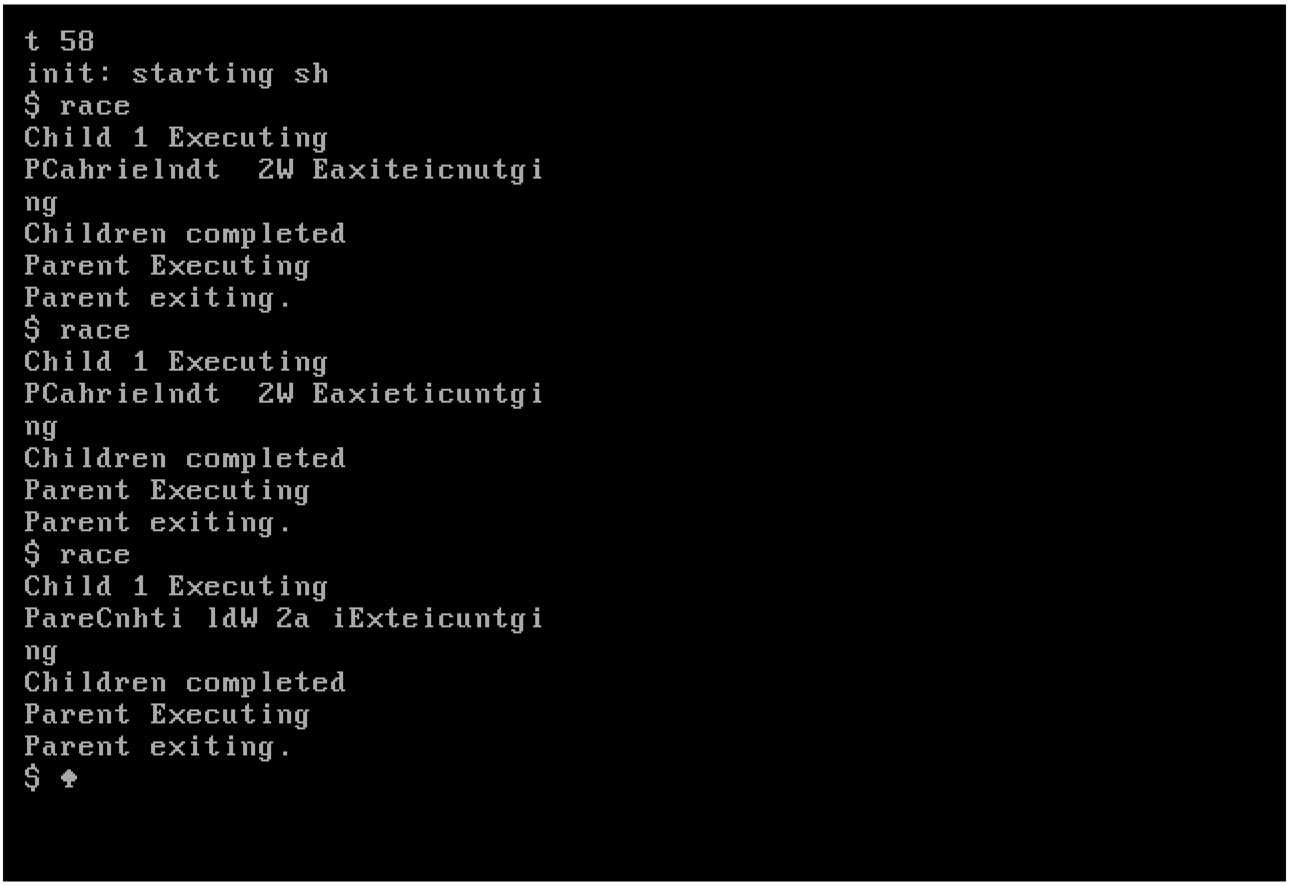
```c
#include "types.h"
#include "stat.h"
#include "user.h"
//We want Child 1 to execute first, then Child 2, and finally Parent.
int main() {
int pid = fork(); //fork the first child
if(pid < 0) {
printf(1, "Error forking first child.\n");
} else if (pid == 0) {
printf(1, "Child 1 Executing\n");
} else {
pid = fork(); //fork the second child
if(pid < 0) {
printf(1, "Error forking second child.\n");
} else if(pid == 0) {
printf(1, "Child 2 Executing\n");
} else {
printf(1, "Parent Waiting\n");
int i;
for(i=0; i< 2; i++)
wait();
printf(1, "Children completed\n");
printf(1, "Parent Executing\n");
printf(1, "Parent exiting.\n");
}
}
exit();
}
```

Add _race\ to the UPROGS variable inside your Makefile.



```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_ps\
	_myls\
	_Nprocess\
	_foo\
	_nice\
	_niceticket\
	_race\

```

Compile and run xv-6.

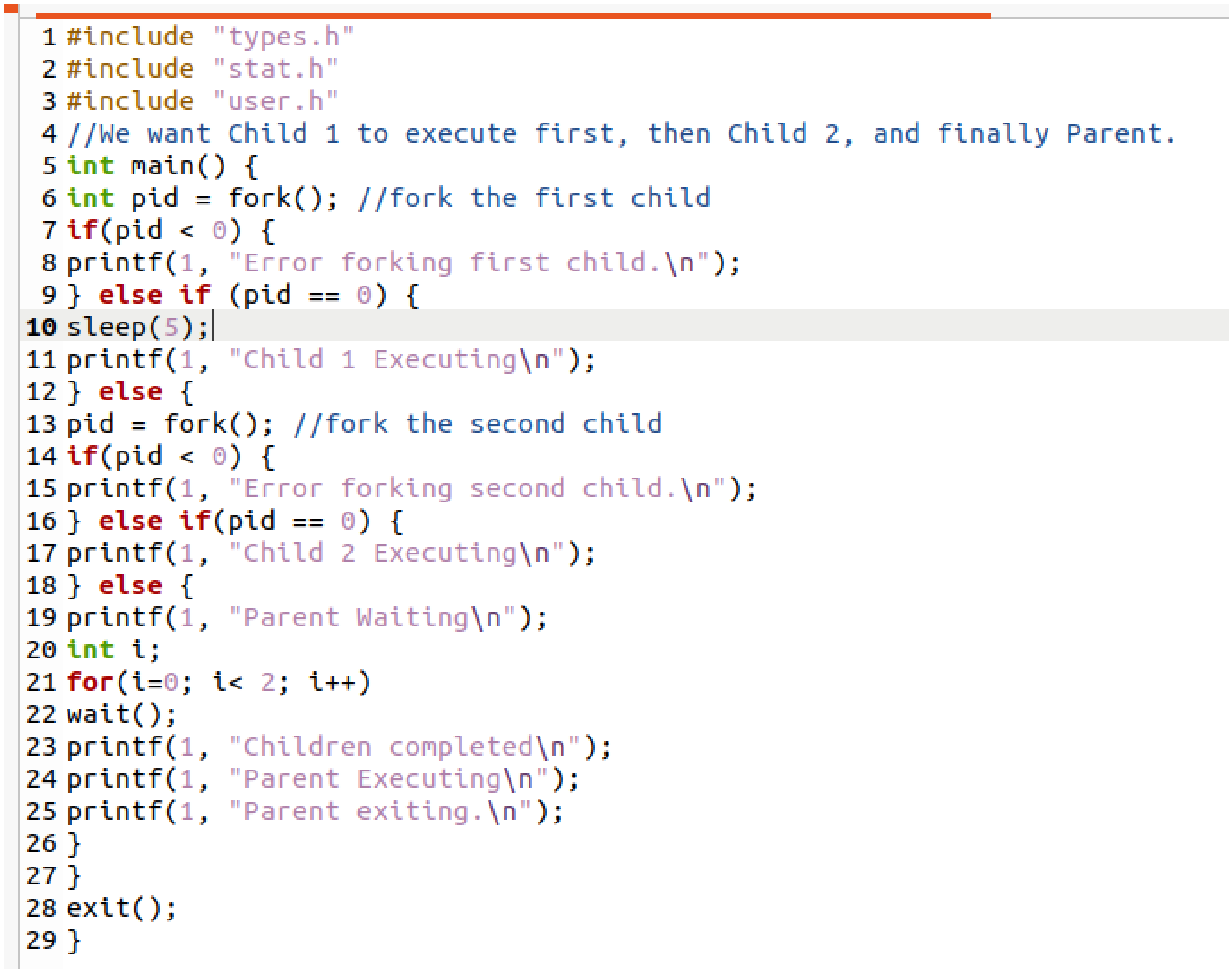


```
t 58
init: starting sh
$ race
Child 1 Executing
PCahrielndt  2W Eaxiteicnutgi
ng
Children completed
Parent Executing
Parent exiting.
$ race
Child 1 Executing
PCahrielndt  2W Eaxieticuntgi
ng
Children completed
Parent Executing
Parent exiting.
$ race
Child 1 Executing
PareCnhti ldW 2a iExteicuntgi
ng
Children completed
Parent Executing
Parent exiting.
$ ♠
```

Do you always get the same order of execution?
No, Parent or the 2nd child may execute first .

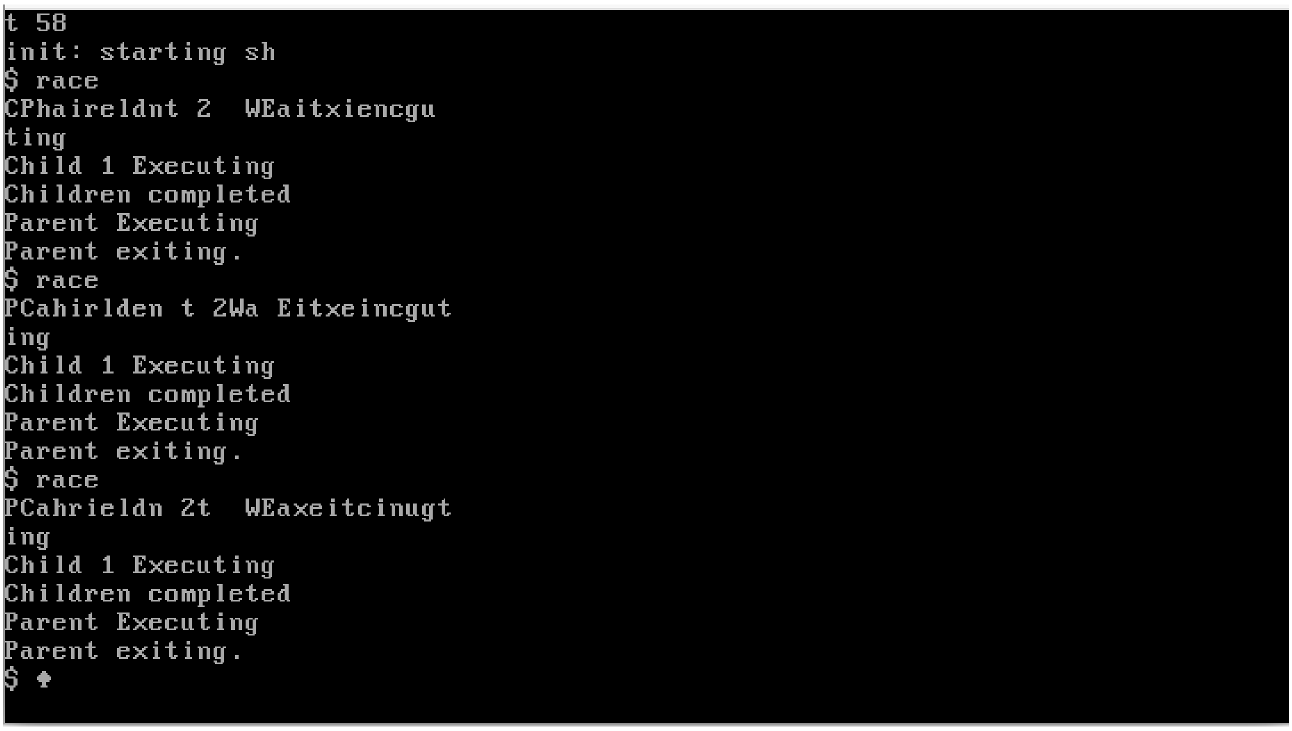
Does Child 1 always execute (print Child 1 Executing) before Child 2?
Yes, Child 1 always execute (print Child 1 Executing) before Child 2.

Add a sleep(5) line before “child 1 executing” .



```
#include "types.h"
#include "stat.h"
#include "user.h"
//We want Child 1 to execute first, then Child 2, and finally Parent.
int main() {
int pid = fork(); //fork the first child
if(pid < 0) {
printf(1, "Error forking first child.\n");
} else if (pid == 0) {
sleep(5);
printf(1, "Child 1 Executing\n");
} else {
pid = fork(); //fork the second child
if(pid < 0) {
printf(1, "Error forking second child.\n");
} else if(pid == 0) {
printf(1, "Child 2 Executing\n");
} else {
printf(1, "Parent Waiting\n");
int i;
for(i=0; i< 2; i++)
wait();
printf(1, "Children completed\n");
printf(1, "Parent Executing\n");
printf(1, "Parent exiting.\n");
}
}
exit();
}
```

Output is something like this.



```
t 58
init: starting sh
$ race
CPhaireldnt 2  WEaitxiencgu
ting
Child 1 Executing
Children completed
Parent Executing
Parent exiting.
$ race
PCahirlden t 2Wa Eitxeincgut
ing
Child 1 Executing
Children completed
Parent Executing
Parent exiting.
$ race
PCahrieldn 2t  WEaxeitcinugt
ing
Child 1 Executing
Children completed
Parent Executing
Parent exiting.
$ ♠
```

What do you notice?
We notice that Child 2 or Parent executes before child 1, due to sleep() function.

Can we guarantee that Child 1 always execute before Child 2?
No.

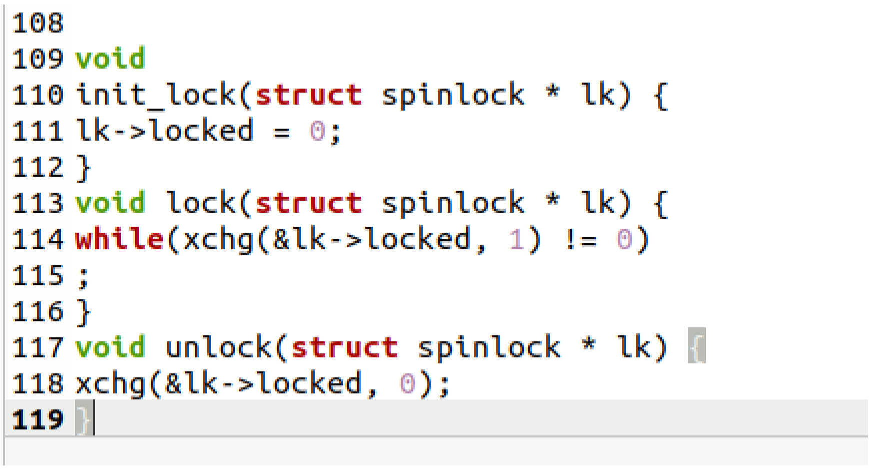
We will define a spinlock that we can use in our user-land program. Xv6 already has a spinlock (see spinlock.c) that it uses inside its kernel and is coded in somehow a complex way to handle concurrency caused by interrupts. We don't need most of the complexity, so will write our own light-weight version of spinlocks. We will put our code inside ulib.c, which includes functions accessible to user-land programs.

Inside ulib.c, add

#include "spinlock.h"

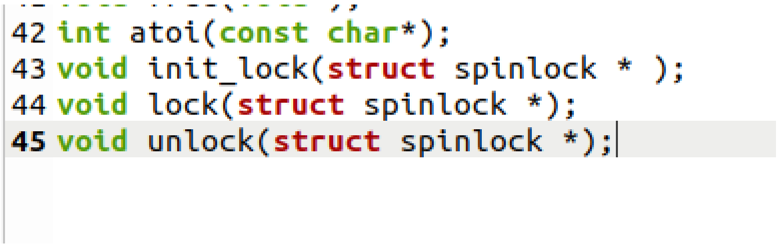
to the beginning.

Also, add the following function definitions:



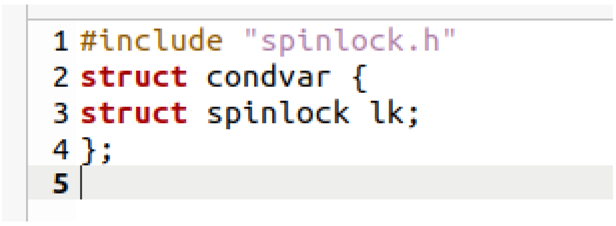
```

void
init_lock(struct spinlock * lk) {
lk->locked = 0;
}
void lock(struct spinlock * lk) {
while(xchg(&lk->locked, 1) != 0)
;
}
void unlock(struct spinlock * lk) {
xchg(&lk->locked, 0);
}
```

Inside user.h add following to end of file :



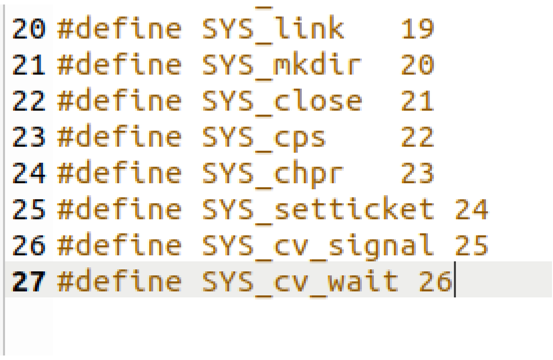
```
void free(void *);
int atoi(const char*);
void init_lock(struct spinlock * );
void lock(struct spinlock *);
void unlock(struct spinlock *);
```

Define condvar.h with following code :



```
#include "spinlock.h"
struct condvar {
struct spinlock lk;
};

```

Then add cv_signal and cv_wait in syscall.h



```
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_cps     22
#define SYS_chpr    23
#define SYS_setticket 24
#define SYS_cv_signal 25
#define SYS_cv_wait 26
```

Inside usys.S, add:

SYSCALL(cv_signal)
SYSCALL(cv_wait)

Inside syscall.c, add:
extern int sys_cv_signal(void);
extern int sys_cv_wait(void);
and
[SYS_cv_wait] sys_cv_wait,
[SYS_cv_signal] sys_cv_signal,

Inside user.h, add

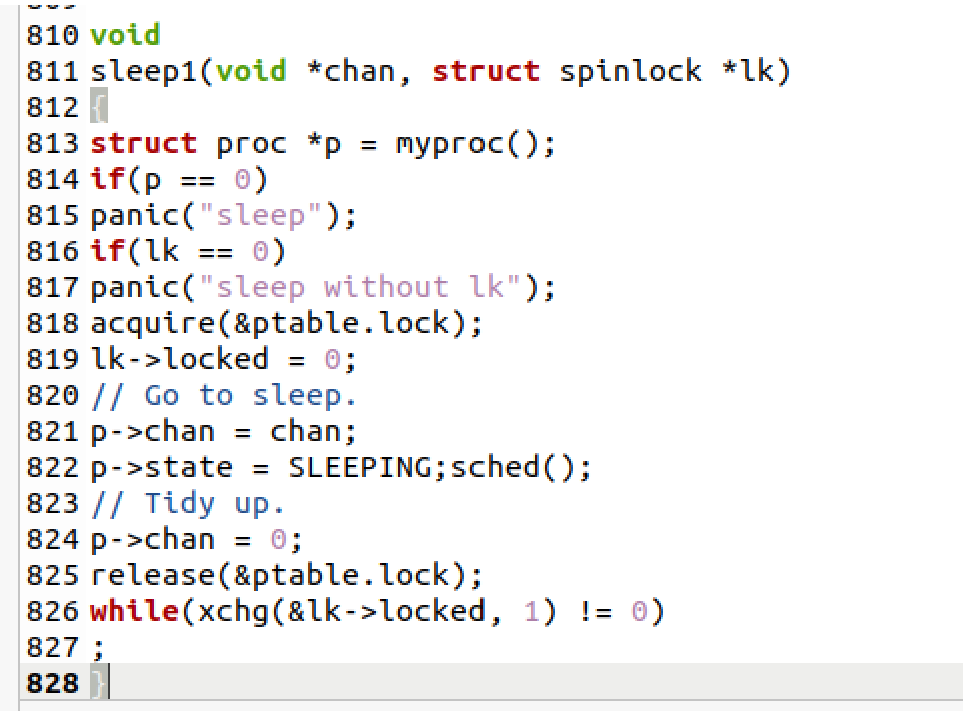
struct condvar;

to the beginning and

int cv_wait(struct condvar *);
int cv_signal(struct condvar *);

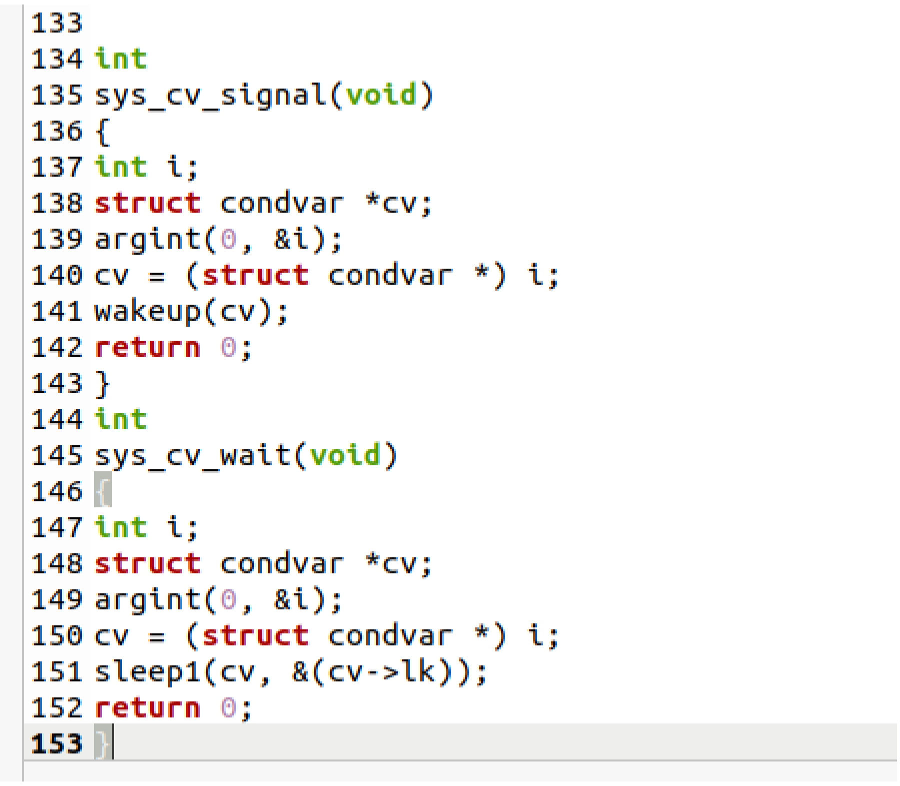
to the end of the system calls section of the file.

Inside proc.c, add the following function definition:



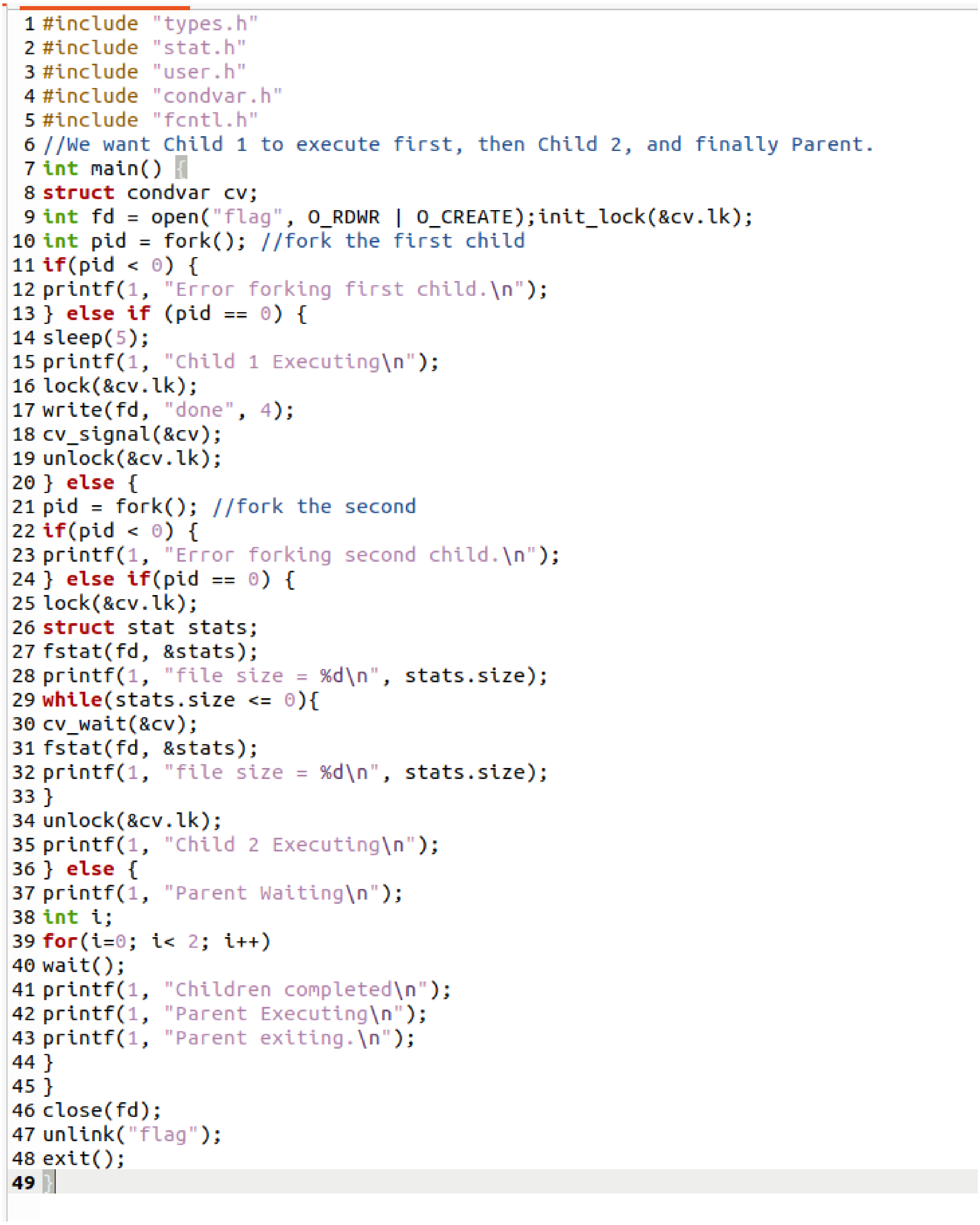
```
void
sleep1(void *chan, struct spinlock *lk)
{
struct proc *p = myproc();
if(p == 0)
panic("sleep");
if(lk == 0)
panic("sleep without lk");
acquire(&ptable.lock);
lk->locked = 0;
// Go to sleep.
p->chan = chan;
p->state = SLEEPING;sched();
// Tidy up.
p->chan = 0;
release(&ptable.lock);
while(xchg(&lk->locked, 1) != 0)
;
}
```

Add in defs.h
void
sleep1(void*, struct spinlock*);
Inside sysproc.c add
#include "condvar.h"
to the beginning of the file and the following system call functions to the end.



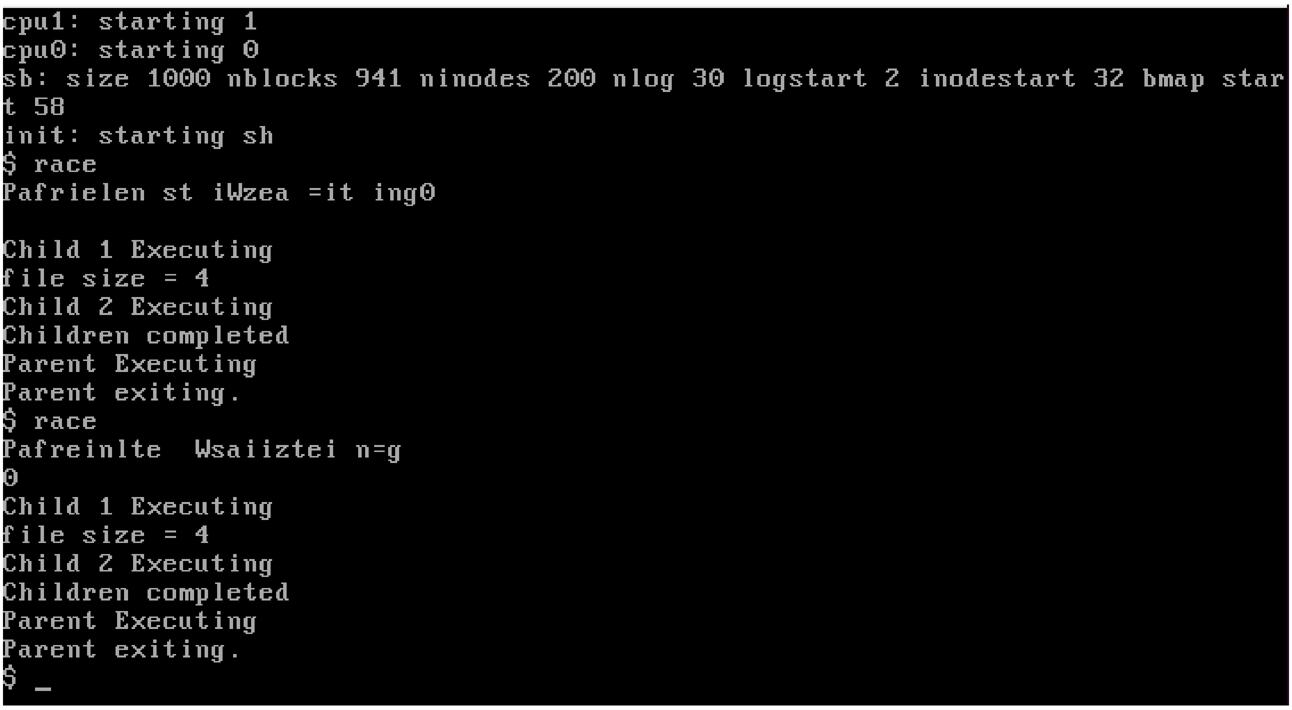
```

int
sys_cv_signal(void)
{
int i;
struct condvar *cv;
argint(0, &i);
cv = (struct condvar *) i;
wakeup(cv);
return 0;
}
int
sys_cv_wait(void)
{
int i;
struct condvar *cv;
argint(0, &i);
cv = (struct condvar *) i;
sleep1(cv, &(cv->lk));
return 0;
}
```

Modify race.c .



```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "condvar.h"
#include "fcntl.h"
//We want Child 1 to execute first, then Child 2, and finally Parent.
int main() {
struct condvar cv;
int fd = open("flag", O_RDWR | O_CREATE);init_lock(&cv.lk);
int pid = fork(); //fork the first child
if(pid < 0) {
printf(1, "Error forking first child.\n");
} else if (pid == 0) {
sleep(5);
printf(1, "Child 1 Executing\n");
lock(&cv.lk);
write(fd, "done", 4);
cv_signal(&cv);
unlock(&cv.lk);
} else {
pid = fork(); //fork the second
if(pid < 0) {
printf(1, "Error forking second child.\n");
} else if(pid == 0) {
lock(&cv.lk);
struct stat stats;
fstat(fd, &stats);
printf(1, "file size = %d\n", stats.size);
while(stats.size <= 0){
cv_wait(&cv);
fstat(fd, &stats);
printf(1, "file size = %d\n", stats.size);
}
unlock(&cv.lk);
printf(1, "Child 2 Executing\n");
} else {
printf(1, "Parent Waiting\n");
int i;
for(i=0; i< 2; i++)
wait();
printf(1, "Children completed\n");
printf(1, "Parent Executing\n");
printf(1, "Parent exiting.\n");
}
}
close(fd);
unlink("flag");
exit();
}
```

OUTPUT:



```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ race
Pafrielen st iWzea =it ing0

Child 1 Executing
file size = 4
Child 2 Executing
Children completed
Parent Executing
Parent exiting.
$ race
Pafreinlte  Wsaiiztei n=g
0
Child 1 Executing
file size = 4
Child 2 Executing
Children completed
Parent Executing
Parent exiting.
$ _
```

Question:
What is the effect of the parent process calling wait() two times?

The parent process calls wait two times for its 2 child process, if parent will not call wait, the two process will enter in zombie state and will not be terminated.

Question:
After seeing what the two system calls do, why do you think we had to add system calls for the operations on condition variables? Why not just have these operations as functions in ulib.c as we did for the spinlock?

We need system calls for sleep() and wakeup() signal.