Name: Shishu
Registration No.:2020CA089

1. **PART 1: THREE PROCESSES - A PARENT AND TWO CHILDREN**

   In the first part you will write a small user-land program that has a parent process forking two child processes, resulting in a total of three processes: the parent process and the two children.

   Put the previous code into a file named race.c inside your Xv6 source code folder. Add _race\ to the UPROGS variable inside your Makefile. Compile and run Xv6 Run the user-land program inside Xv6 by typing race at the Xv6 prompt. Notice the order of execution of the three processes. Run the program multiple times.

   • Do you always get the same order of execution?

   - Yes, order of execution's is same every time.

   • Does Child 1 always execute (print Child 1 Executing) before Child 2?

   - **No, when we add sleep statement before the child 1 print statement then first child 2 is executing.**

Race.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) {

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();
}
```

Makefile:

```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c ps.c zombie.c\
	printf.c myls.c umalloc.c foo.c nice.c race.c NProcess.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

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

```

Output:

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200
init: starting sh
$ race
Child 1 Executing
PCarhenit Waitingl
d 2 Executing
Children completed
Parent Executing
Parent exiting.
$ race
Child 1 Executing
PCarheinlt Waitingd
 2 Executing
Children completed
Parent Executing
Parent exiting.
$ race
Child 1 Executing
PCahrielndt 2  EWxaeitcinug
ting
Children completed
Parent Executing
Parent exiting.
```

After adding sleep(5); before line before the line where Child 1 prints "Child Executing".

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200
init: starting sh
$ race
PCahreintld 2 E xWaeicutitinngg

Child 1 Executing
Children completed
Parent Executing
Parent exiting.
$ race
PChairledn t2 E xWeaciting
uting
Child 1 Executing
Children completed
Parent Executing
Parent exiting.
$ race
ChilPda r2 Eexencutt inWg
aiting
Child 1 Executing
Children completed
Parent Executing
Parent exiting.
```

What do you notice?

-I notice that firstly on xv6 terminal when we type 'race' then child 1 is created after that child 2 is created and when both the children are completing then the parent executing and then exiting.

Can we guarantee that Child 1 always execute before Child 2?

-It is not guaranteed that child 1 always execute before child 2 because when we add a sleep statement before child 1 then first child 2 can be executed and after that child 1 Is executed.

2. **PART 2: SPIN LOCKS**

We will start by defining 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.

4. **Inside ulib.c, add**

```
Open ▾
#include "types.h"
#include "stat.h"
#include "fcntl.h"
#include "user.h"
#include "x86.h"
#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);


```

We are still using the `struct spinlock` defined in spinlock.h but we will only use its `locked`
field. Initializing the lock and unlocking it both work by setting `locked` to 0. Locking uses the

atomic `xchg` instruction, which sets the contents of its first parameter (a memory address) to the second parameter and returns the old value of the contents of the first parameter.

5. **Inside user.h, add the following function prototypes:**

```
void init_lock(struct spinlock * lk);
void lock(struct spinlock * lk);
void unlock(struct spinlock * lk);
```

to the end of the file.

Now, we have our spinlocks in place. We can use them inside race.c:

```
struct spinlock lk;
init_lock(&lk);
lock(&lk);
//critical section
unlock(&lk)
```

We will use condition variables to be able to make Child 2 sleep (block) until Child 1 finishes execution.

3. PART 3: CONDITION VARIABLES

We will use condition variables to ensure that Child 1 always executes before Child 2. We will add two system calls to Xv6: `cv_wait()` and `cv_signal()` to wait (sleep) on a condition variable and to wakeup (signal) all sleeping processes on a condition variable.

Recall that both waiting and signaling a condition variable has to be called after acquiring a lock (that's why we defined our spinlock in Part 2). `cv_wait` releases the lock before sleeping and reacquires it after waking up.

6. First, define the condition variable structure in condvar.h as follows.

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

A condition variable has a spin lock.

Let's then add the two system calls.

7. Inside syscall.h, add the following two lines:

```
#define SYS_cv_signal 31
#define SYS_cv_wait 32
```

Inside usys.S, add:

```
SYSCALL(cv_signal)
SYSCALL(cv_wait)
```

Inside syscall.c, add:

```
[SYS_cv_wait] sys_cv_wait,
[SYS_cv_signal] sys_cv_signal,
```

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.

Our condition variable implementation depends heavily on the sleep/wakeup mechanism implemented inside Xv6 (Please read **Sleep and Wakeup** on Page 65 of the Xv6 book). We will again define a more light-weight version of the sleep function to use our light-weight spinlocks defined in Part 2 instead of

8. Inside proc.c, add the following function definition:

```c
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)
;
}
```

After a couple of sanity checks, the function acquires the process table lock `ptable.lock` to be able to call `sched`(), which works on the process table. Then, it releases the spinlock (by setting locked to 0) and goes to sleep by setting the process state to SLEEPING, setting the channel that the process sleeps on, and switching into the scheduler by calling sched(). After the process wakes up, it releases the ptable lock and reacquires the spinlock (using the xchg instruction).

9. Inside sysproc.c add

```
#include "condvar.h"
```

to the beginning of the file and the following system call functions

```
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;
}
```

to the end. In both functions, the code starts with retrieving the argument (struct condvar *) from the stack:

```
argint(0, &i);
cv = (struct condvar *) i;
```

The address of the condition variable is used as the channel passed over to the sleep1 function defined in Step 8. The address of the condition variable is unique and this is all what we need: a unique channel number to sleep and to get waked up on.

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?

4. PART 4: USING THE CONDITION VARIABLES

We can then modify race.c to use a condition variable to guarantee process ordering.

Race.c

Open race.c ~/xv6

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "condvar.h"
//We want Child 1 to execute first, then Child 2, and finally Parent.
int main() {
struct condvar cv;
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);
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);
cv_wait(&cv);
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");
}
}
exit();
}
```

Note the highlighted parts. A condition variable is declared. Its spinlock is initialized. Then Child 1
signals the condition variable after acquiring the spinlock. Child 2 sleeps on the condition variable after acquiring the spinlock as well.

Compile and run the modified race program.

- Does Child 1 always execute before Child 2?

5. PART 5: LOST WAKEUPS

Does it happen that the program gets stuck? This is called a **deadlock**. If Child 2 gets to sleep **after** Child 1 signals, the wakeup signal is lost (i.e., never received by Child 2). In this case, Child 2 has no way of being awaked.

To solve this problem, we need to enclose the `cv_wait()` call inside a while loop. We need some form of a flag that gets set by Child 1 when it is done executing. Child 2 will then do

```
while(flag is not set)

        cv_wait();
```

This way, even if Child 1 sets the flag and signals before Child 2 executes the while loop, Child 2 will not avoid going to sleep because the flag will be set.

The flag has to be **shared** between the two processes. We will use a **file** for that. Other methods for sharing are shared memory and pipes.

To create a file,

```
int fd = open("flag", O_RDWR | O_CREATE);
```

To write into the file,

```
write(fd, "done", 4);
```

Checking the flag has to be non-blocking. The `read` system call is blocking. Reading the size of the file is not. So, we will check the flag by reading the file size. To read the size of a file,

Now, we are ready to write the while loop inside Child 2. It will loop until the file size is greater than zero, which happens when Child 1 writes "done" into the file after it finishes execution.

The new race.c is:

```c
#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");
```

Note the highlighted parts and also note that we are closing the file and deleting it before the parent exits. This is to start afresh the next time we run the program.

Compile and run race.c many times.

- Is it always the case that Child 1 executes before Child 2?
- Do you observe deadlocks?

Of course, synchronization bugs cannot be ruled out by running a program many times. Formal proof is typically the preferred way especially for safety- and mission-critical systems. There are tools that help with this kind of formal proofs.

6. SUBMISSION INSTRUCTIONS

Follow the steps of this lab and submit all the files that you had to modify in a zipped archive by uploading the zipped file to Lab 2 submission page on CourseWeb. The deadline is listed on the Lab 2 page on CourseWeb.