# Homework3

Name Yu Yang Student ID 892449550

# 5.3

Busy waiting: While a process is in its critical section, any other process that tries to enter its critical section must loop continuously.

Other kind of waiting: a process could block itself, be put in a wait queue and be awakened in the future.

Busy waiting can be avoided, but it requires overhead of putting a process in into a wait queue and waking it up again.

# 5.4

In a single-processor system, a single cpu is shared among many processes. Busy waiting of spinlock wastes CPU cycles that some other process might be able to use productively.

In a multiprocessor system, one thread can “spin” on one processor while another thread performs its CS on another processor.

# 5.10

If a user-level could disable interrupts, it can block all other processes if there is only one processor.

# 5.11

Disabling Interrupts on the processor could not influence processes executed on other processors. So it can not guarantee mutex.

On the other hand, disabling interrupts on every processor can be a difficult task and seriously diminish performance.

# 5.23

Int gate = 0;// global shared

Typedef struct{

Int value; //available num

Struct process \*waiting\_queue;

} semaphore

Wait(semaphore \*s){

While(test\_and\_set(&gate));//only the first process getting here,

//or when other process get a semaphore,

// a process could enter

If(s->value <=0){

Add this process to s->wating queue;

Block();

}else{

S->value --;

gate = 0;//this process leaves wait, let others getting through the gete

}

}

Signal(){

While(test\_and\_set(&gate));// same reason as in wait()

If(s->value <= 0 && not\_empty(s->waiting\_queue)){

Wakeup(S->waiting\_queue.remove());

}

Else{

s->value ++;

gate = 0;

}

}

# 5.28

Throughput is increased by allowing multiple reading processes and only one writing process: A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

This approach could cause writer starving.

Solution: 1. Avoid keep letting new readers go into CS: when a new reader comes when several readers are in CS and a writer is waiting, block the new reader until waiting writer finishes.

2.keeping track of the waiting time of every process.

When several readers finish or a writer finish, give access to the process has waited for the longest time.

# 5.29

The signal() operation in monitor resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed. Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore(++ operation of the semaphore value).

5.32

A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file

must be less than n. Write a monitor to coordinate access to the file.

Monitor fileAllocator{

Int sum;

Static int limit;

Condition x;

Void accessFile(int process\_num){

While(sum + process\_num >= limit)

x.wait();

sum += process\_num;

}

Void release(int process\_num){

Sum -= process\_num;

c.signal();

}

Initialization(){

Sum = 0;

Limit = n;

}

}

# 5.37

a. available\_resources

b. available\_resource -= count and available\_resources += count(because -= and += are not atomic op)

c. use semaphore with initialized value of MAX\_RESOURCES. All Ops on -= or += are modified to use semaphore.

# 6.2, 6.3, 6.11, 6.16, 6.24, 6.31

#include <stdio.h>

#include <fcntl.h>

#include <zconf.h>

int fake\_write(int fd,const void \*buffer,size\_t sz){

return sz;

}

int main() {

//open file

int file = open("/dev/null",O\_WRONLY);

char \*s = "1234567890";

int i = 0;

//write

for(i = 0; i<100000000;++i)

fake\_write(file,s,10);

}

Real 0m0.304s

User 0m0.301s

Sys 0m0.001s

Bacause it need to switch from usr mode to kernel mode, and when system call are done, it will switch back, this take much system time. And IO is slow.

# Problem 3

## exec

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <sys/wait.h>

void \*runner(void \*param){

if(fork()==0){

printf("command line param is %s\n",param);

//wait another

wait(NULL);

}else{

execl("/bin/ls","ls",NULL);

printf("exec() return -- should not be returned");

}

pthread\_exit(0);

}

int main(int argc, char \*ag[]) {

if(argc != 2){

perror("arg num is not 2");

exit(1);

}

pthread\_t pid;

pthread\_attr\_t at;

pthread\_attr\_init(&at);

pthread\_create(&pid,&at,runner,ag[1]);

pthread\_join(pid,NULL);

}

Exec doesn’t copy threads, so the printf("exec() return -- should not be returned") will not be executed.

## Fork

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <sys/wait.h>

void \*runner(void \*param){

if(fork()==0){

printf("command line param is %s\n",param);

//wait another

wait(NULL);

}else{

if(fork() == 0){

printf("fork() in thread");

}

printf("fork() in thread 2nd\n");

}

printf("after fork\n");

pthread\_exit(0);

}

int main(int argc, char \*ag[]) {

if(argc != 2){

perror("arg num is not 2");

exit(1);

}

pthread\_t pid;

pthread\_attr\_t at;

pthread\_attr\_init(&at);

pthread\_create(&pid,&at,runner,ag[1]);

pthread\_join(pid,NULL);

}

Fork copy thread, printf("fork() in thread 2nd\n"); are printed twice.

## Threadsig.c

#include <stdio.h>

#include <zconf.h>

#include <pthread.h>

#include <stdlib.h>

#include <signal.h>

int result[200];

void sig\_handle(int sig){

printf("terminated!");

};

int main() {

signal(SIGINT,sig\_handle);

sleep(100);

}

As written in the textbook:

“ When a signal is generated by an event external to a running process, that

process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>)and having a timer expire.”