David Cruz & George Suarez
CSE-460
Lab-8

1-Dining Philosophers and Deadlock:

Try the following program type ^C run it for some time to check # of philosophers and type ^\ to quit.

//dine1.cpp

```
/*
 dine1.cpp :  mutex locks eating
 Compile:  g++ -o  dine1 dine1.cpp -lSDL
 Execute:  ./dine1

*/

#include <SDL/SDL.h> #include <SDL/SDL_thread.h> #include <stdio.h>
#include <stdlib.h> #include <math.h>
#include <signal.h> #include <unistd.h>
using namespace std;

SDL_mutex *mutex;// mutex to lock eating
bool quit = false;
int nEating = 0; // number of philosophers eating

void think( int i ){
// printf("\n%d thinking", i ); SDL_Delay ( rand() % 3000);

}
void eat( int i ) {

    //  printf("\n%d eating!", i );


    SDL_Delay ( rand() % 3000);
}

void take_chops( int i ) { //printf("\nTaking chopstick %d", i );

}
void put_chops( int i ) {

    // printf("\nReleasing chopstick %d", i );
}

int philosopher( void *data ) { int i;

    i = atoi ( (char *) data );
    while ( !quit ) {


        think( i );
        SDL_LockMutex ( mutex );
        take_chops ( i );
        take_chops ( (i+1) % 5 );
        nEating++;


eat ( i );

        nEating--;
        put_chops ( i );
        put_chops ( (i+1) % 5 );


       SDL_UnlockMutex ( mutex );
   }
```

```
}
void checkCount ( int sig ) {

    if ( sig == SIGINT )
        printf("\n%d philosophers eating\n", nEating );


    else if ( sig == SIGQUIT ) {
        quit = true;


     printf("\nQuitting, please wait...\n");
   }


}

int main () {

    struct sigaction act, actq;


    act.sa_handler = checkCount;
    sigemptyset ( &act.sa_mask );
    sigaction ( SIGINT, &act, 0 );
    actq.sa_handler = checkCount;
    sigaction ( SIGQUIT, &actq, 0 );


SDL_Thread *p[5]; //thread identifiers const char *names[] = { "0", "1", "2", "3", "4" };

    mutex = SDL_CreateMutex();
    for ( int i = 0; i < 5; i++ ) {


        p[i] = SDL_CreateThread ( philosopher, (char *) names[i] );
    }


    for ( int i = 0; i < 5; i++ ) {
        SDL_WaitThread ( p[i], NULL );


    }
    SDL_DestroyMutex ( mutex );


return 0;

}
```

Dine 1-Output:

```
$ g++ -o dine1 dine1.cpp -lSDL

$ ./dine1
^C
1 philosophers eating
^C
1 philosophers eating
^C
1 philosophers eating
^C
1 philosophers eating
^C
1 philosophers eating
^C
1 philosophers eating
^C
1 philosophers eating
^C
1 philosophers eating
^\
Quitting, please wait…
```

What conclusion can you draw on the number of philosophers that can eat at one time???

- After seeing the output, we can conclude that only one philosopher is eating at a time, so only one philosopher is allowed to eat at one time.

//dine2.cpp

```
*****************************
* Same includes as Dine1.cpp *
*****************************

SDL_sem *chopLock[5]; //locks for chopsticks bool quit = false;

int nEating = 0; // number of philosophers eating

void think( int i ) {
 SDL_Delay ( rand() % 2000);


}
void eat( int i ) {

 printf("\Philosopher %d eating!\n", i );

 SDL_Delay ( rand() % 2000);
}

void take_chops( int i ) {
 printf("\nTaking chopstick %d", i );

}
void put_chops( int i ) {
}

int philosopher( void *data ) {
  int i, l, r;

  i = atoi ( (char *) data );
  l = i;   //left
  r = (i+1) % 5;
  while ( !quit ) {

    think( i );
    printf("\nPhilosopher %d ", i );
    SDL_SemWait ( chopLock[l] );
    take_chops ( l );
    //SDL_Delay ( rand() % 2000 );  //could lead to deadlock
    SDL_SemWait ( chopLock[r] );
    take_chops ( r );
    nEating++;
    eat ( i );
    nEating--;
    put_chops ( r );
    SDL_SemPost ( chopLock[r]  );
    put_chops ( l );
    SDL_SemPost ( chopLock[l] );

} }
```

```
void checkCount ( int sig ) {
  if ( sig == SIGINT )

    printf("\n%d philosophers eating\n", nEating );
  else if ( sig == SIGQUIT ) {

    quit = true;
    printf("\nQuitting, please wait....\n");
    for ( int i = 0; i < 5; i++ ) {   // break any deadlock

      printf("\nUnlocking %d ", i );
      SDL_SemPost ( chopLock[i]  );
      printf("\nUnlocking %d done", i );

} }

}

int main () {

  struct sigaction act, actq;

  act.sa_handler = checkCount;
  sigemptyset ( &act.sa_mask );
  sigaction ( SIGINT, &act, 0 );
  actq.sa_handler = checkCount;
  sigaction ( SIGQUIT, &actq, 0 );

  SDL_Thread *p[5];
  const char *names[] = { "0", "1", "2", "3", "4" };

//thread identifiers

  for ( int i = 0; i < 5; i++ )
    chopLock[i] = SDL_CreateSemaphore( 1 );

  for ( int i = 0; i < 5; i++ )
    p[i] = SDL_CreateThread ( philosopher, (char *) names[i] );

  for ( int i = 0; i < 5; i++ )
    SDL_WaitThread ( p[i], NULL );

  for ( int i = 0; i < 5; i++ )
    SDL_DestroySemaphore ( chopLock[i] );

return 0;

}
```

Dine 2-Output:

```
$ g++ -o dine2 dine2.cpp -lSDL
$ ./dine2
```

```
Philosopher 2
Taking chopstick 2
Taking chopstick 3
Philosopher 2 eating!

Philosopher 1
Taking chopstick 1
Philosopher 3
Taking chopstick 3
Taking chopstick 4
Philosopher 3 eating!

Taking chopstick 2
Philosopher 1 eating!

Philosopher 0
Taking chopstick 0
Taking chopstick 1
Philosopher 0 eating!

Philosopher 4
Taking chopstick 4
Taking chopstick 0
Philosopher 4 eating!

Philosopher 1
Taking chopstick 1
Taking chopstick 2
Philosopher 1 eating!

Philosopher 0
Taking chopstick 0
Philosopher 2
Philosopher 3
Taking chopstick 3
Taking chopstick 4
Philosopher 3 eating!
^C
2 philosophers eating

Philosopher 4
Taking chopstick 4
Taking chopstick 2
Taking chopstick 3

Philosopher 2 eating!

Taking chopstick 1
Philosopher 0 eating!

^\Philosopher 3

Quitting, please wait....

Unlocking 0
Unlocking 0 done
Unlocking 1
Unlocking 1 done
Unlocking 2
Unlocking 2 done
```

```
Unlocking 3
Taking chopstick 0
Philosopher 4 eating!

Unlocking 3 done
Unlocking 4
Unlocking 4 done
Taking chopstick 3
Taking chopstick 4
Philosopher 3 eating!

Philosopher 1
Taking chopstick 1
Taking chopstick 2
Philosopher 1 eating!
```

As we see on the output above we see that now unlike ***dine1.cpp*** 2 philosophers are allowed to eat simultaneously.

```
//Dine-3.cpp

//dine3.cpp
/*

 dine3.cpp
 Compile:  g++ -o  dine3 dine3.cpp -lSDL
 Execute:  ./dine3

*/

#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <signal.h>
#include <unistd.h>

#define LEFT (i - 1) % 5
#define RIGHT (i + 1) % 5
#define HUNGRY 0
#define EATING 1
#define THINKING 2

SDL_sem *s[5];
bool quit = false;
int nEating = 0;
SDL_mutex *mutex;
int state[5];

void test(int i)
{
    //one semaphore per philosopher to lock chopsticks
    // number of philosophers eating
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[i] = EATING;
        SDL_SemPost(s[i]);
    }
}
```

```
void think(int i)
{
    SDL_Delay(rand() % 2000);
}

void take_chops(int i)
{
    SDL_LockMutex(mutex);
    state[i] = HUNGRY;
    printf("\nTaking chopstick %d", i);
    test(i);
    SDL_UnlockMutex(mutex);
}
void eat(int i)
{
    printf("\nPhilosopher %d eating!\n", i);
    SDL_Delay(rand() % 2000);
}
void put_chops(int i)
{
    SDL_LockMutex(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    SDL_UnlockMutex(mutex);
}

void checkCount(int sig)
{
    if (sig == SIGINT)
        printf("\n%d philosophers eating\n", nEating);
    else if (sig == SIGQUIT)
    {
        quit = true;
        printf("\nQuitting, please wait....\n");
        for (int i = 0; i < 5; i++)
        { // break any deadlock
            printf("\nUnlocking %d ", i);
            SDL_SemPost(s[i]);
            printf("\nUnlocking %d done", i);
        }
    }
}

int philosopher(void *data)
{
    int i, l, r;
    i = atoi((char *)data);
    l = i; //left
    r = (i + 1) % 5;
    while (!quit)
    {
        think(i);
        printf("\nPhilosopher %d ", i);
        SDL_SemWait(s[l]);
        take_chops(l);
        SDL_Delay(rand() % 2000);
        SDL_SemWait(s[r]);
        take_chops(r);
        nEating++;
        eat(i);
        nEating--;
        put_chops(r);
        SDL_SemPost(s[r]);
```

```
        put_chops(l);
        SDL_SemPost(s[l]);
    }
}

int main()
{
    struct sigaction act, actq;
    act.sa_handler = checkCount;

    sigemptyset(&act.sa_mask);
    sigaction(SIGINT, &act, 0);
    actq.sa_handler = checkCount;
    sigaction(SIGQUIT, &actq, 0);

    SDL_Thread *p[5];

    const char *names[] = {"0", "1", "2", "3", "4"};
    for (int i = 0; i < 5; i++)
        s[i] = SDL_CreateSemaphore(1);
    for (int i = 0; i < 5; i++)
        p[i] = SDL_CreateThread(philosopher, (char *)names[i]);
    for (int i = 0; i < 5; i++)
        SDL_WaitThread(p[i], NULL);
    for (int i = 0; i < 5; i++)
        SDL_DestroySemaphore(s[i]);
    return 0;
}
```

Output:

```
$ g++ -o dine3 dine3.cpp -lSDL

$ ./dine3
Philosopher 2
Taking chopstick 2

Philosopher 1
Taking chopstick 1
Philosopher 3
Taking chopstick 3
Philosopher 0
Taking chopstick 0
Taking chopstick 4
Philosopher 3 eating!

Philosopher 4
Taking chopstick 2
Philosopher 1 eating!

Taking chopstick 3
Philosopher 2 eating!

Taking chopstick 1
Philosopher 0 eating!

Philosopher 1
Taking chopstick 1
Taking chopstick 4
Philosopher 3
Taking chopstick 3
Taking chopstick 0
```

```
Philosopher 4 eating!

Philosopher 0
Taking chopstick 0
Philosopher 2
Taking chopstick 2
Taking chopstick 2
Philosopher 1 eating!

//thread identifiers

Taking chopstick 3
Philosopher 2 eating!

Taking chopstick 4
Philosopher 3 eating!
^C
3 philosophers eating

Taking chopstick 1
Philosopher 0 eating!

Philosopher 4
Taking chopstick 4
Philosopher 2
Taking chopstick 2
Philosopher 1
Taking chopstick 1
Philosopher 0
Taking chopstick 0
Taking chopstick 1
Philosopher 0 eating!
^\
Quitting, please wait....

Unlocking 0
Unlocking 0 done
Unlocking 1
Unlocking 1 done
Unlocking 2
Unlocking 2 done
Unlocking 3
Unlocking 3 done
Unlocking 4
Unlocking 4 done
Taking chopstick 3
Philosopher 2 eating!

Philosopher 3
Taking chopstick 3
Taking chopstick 2
Philosopher 1 eating!

Taking chopstick 0
Philosopher 4 eating!

Taking chopstick 4
Philosopher 3 eating!
```

Based on the output above we see how deadlock did not occurred because all the philosophers got to eat. Based on the output 3 philosophers are able

```
to eat at the same time.
```

2-XV6 Process Priority

- Add *priority* to *struct proc* in *proc.h*:

```
struct proc {
  uint sz;
  . . .
  char name[16];               // Process name (debugging)
  int priority;
  . . .
```

- Assign default priority in **allocproc()** in *proc.c*:

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;
  . . .
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 10; // default priority
  . . .
```

- Modify **cps()** in *proc.c* discussed in the last lab to include the printout of the priority:

```
int
cps()
{
  struct proc *p;


  // Enable interrupts on this processor.
  sti();


  //int runningProcesses = 0;
```

```
  //int sleepingProcesses = 0;

  // Loop over process table looking for process with pid.
  acquire(&ptable.lock);
  cprintf("name \t pid \t state \t\t priority\n");
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if ( p->state == SLEEPING ) {
              //sleepingProcesses++;
              cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name, p->pid, p->priority);
        }
        else if ( p->state == RUNNING ) {
              //runningProcesses++;
              cprintf("%s \t %d \t RUNNING \t %d \n ", p->name, p->pid, p->priority);
        }
        else if ( p->state == RUNNABLE ) {
              cprintf("%s \t %d  \t RUNNABLE \t %d\n", p->name, p->pid, p->priority);
        }
  }
  release(&ptable.lock);
  return 22;
}
```

- Modify *foo.c* discussed in Lab 6 so that it loops for a much longer time before exit:

```
...

for ( z = 0; z < 8000000.0; z += 0.001 )
        x = x + 3.14 * 89.64;  // useless calculations to consume CPU time
  exit();


...
```

- Add the function **chpr()** (meaning *change priority*) in *proc.c*:

```
// change priority
int
chpr( int pid, int priority )
{
```

```
  struct proc *p;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->pid == pid) {
        p->priority = priority;
        break;
    }
  }
  release(&ptable.lock);

  return pid;
}
```

- Add **sys_chpr()** in *sysproc.c*:

```
int
sys_chpr ( void )
{
  int pid, pr;
  if (argint(0, &pid) < 0)
    return -1;
  if (argint(1, &pr) < 0)
    return -1;

  return chpr ( pid, pr );
}
```

- Add **chpr()** as a system call to xv6 as discussed in the last lab.

  - Adding *chpr* to *syscall.h:*

    ```
    ...
    #define SYS_close  21
    #define SYS_cps    22
    #define SYS_chpr   23
    ```

  - Adding *cphr* to *usys.S:*

```
...
SYSCALL(uptime)
SYSCALL(cps)
SYSCALL(chpr)
```

- Adding the function prototype for ***cphr*** to *syscall.c:*

```
...
extern int sys_uptime(void);
extern int sys_cps(void);
extern int sys_chpr(void);
...
```

```
...
[SYS_close]   sys_close,
[SYS_cps]     sys_cps,
[SYS_chpr]    sys_chpr,
...
```

- Create the user file *nice.c* with which calls **chpr**:

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

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

```
  if (argc < 3 ) {
     printf(2, "Usage: nice pid priority\n");
     exit();
  }
  pid = atoi ( argv[1] );
  priority = atoi ( argv[2] );
  if ( priority < 0 || priority > 20 ) {
     printf(2, "Invalid priority (0-20)!\n");
     exit();
  }
  chpr ( pid, priority );
  exit();
}
```

- Adding *nice* to the *Makefile*

  _wc

  _nice

  _foo

  _cp

  _ps

  ...wc.c nice.c cp.c ps.c foo.c zombie.c

- Testing the modified **cps()** function using *foo*:

qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive

file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512

xv6...

cpu1: starting 1

cpu0: starting 0

sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58

```
init: starting sh
$ foo 4 &
$ ps
name        pid     state       priority
init  1     SLEEPING            10
 sh   2     SLEEPING            10
 foo        5       RUNNING             10
 foo        4       RUNNABLE            10
foo   6     RUNNABLE            10
foo   7     RUNNABLE            10
foo   8     RUNNABLE            10
ps        9 RUNNING             10
```

- Changing the priority level of pid 4 from 10 to 18 using the *nice* command

```
$ nice 4 18
$ ps
name    pid     state     priority
init    1       SLEEPING  10
 sh     2       SLEEPING  10
 foo    5       RUNNABLE  10
foo     4       RUNNABLE  18
foo     6       RUNNING   10
 foo    7       RUNNABLE  10
foo     8       RUNNABLE  10
ps      11      RUNNING   10
```