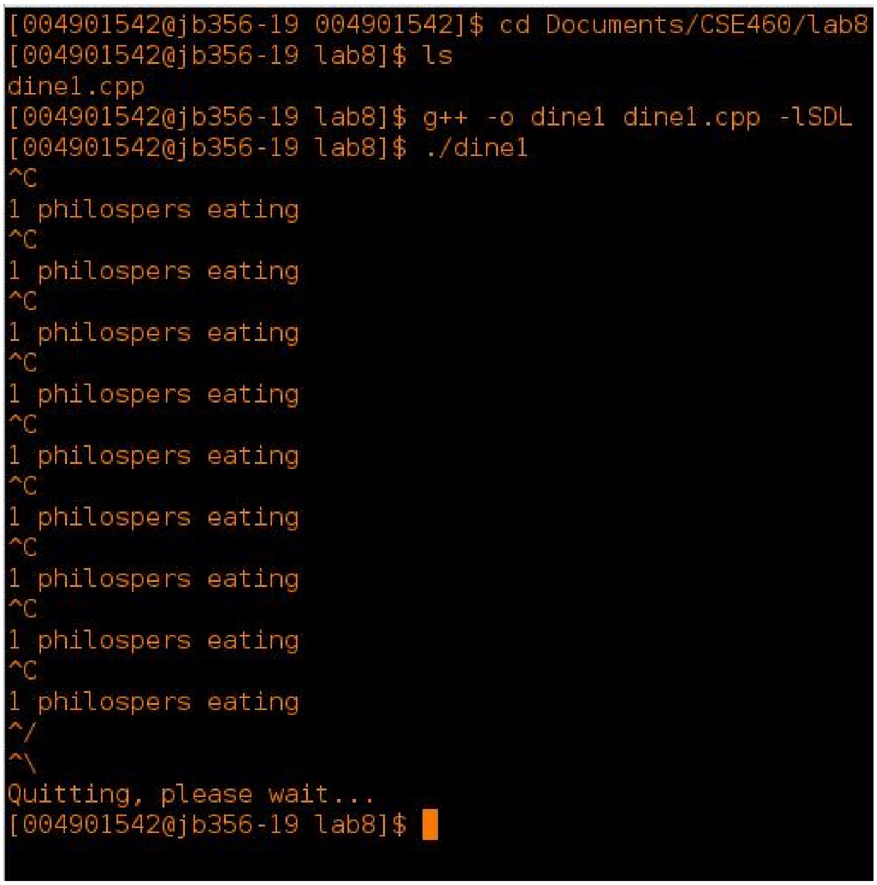
Angel Macias
CSE 460
Lab #8

Dining Philosophers and Deadlock

1. Try *dine1.cpp* above. Type ^C to check the number of philosopers eating. Run it for some time. What conclusion can you draw on the number of philosopers that can eat at one time? To quit the program, type ^\.



```
[004901542@jb356-19 004901542]$ cd Documents/CSE460/lab8
[004901542@jb356-19 lab8]$ ls
dine1.cpp
[004901542@jb356-19 lab8]$ g++ -o dine1 dine1.cpp -lSDL
[004901542@jb356-19 lab8]$ ./dine1
^C
1 philospers eating
^C
1 philospers eating
^C
1 philospers eating
^C
1 philospers eating
^C
1 philospers eating
^C
1 philospers eating
^C
1 philospers eating
^C
1 philospers eating
^C
1 philospers eating
^/
^\
Quitting, please wait...
[004901542@jb356-19 lab8]$
```

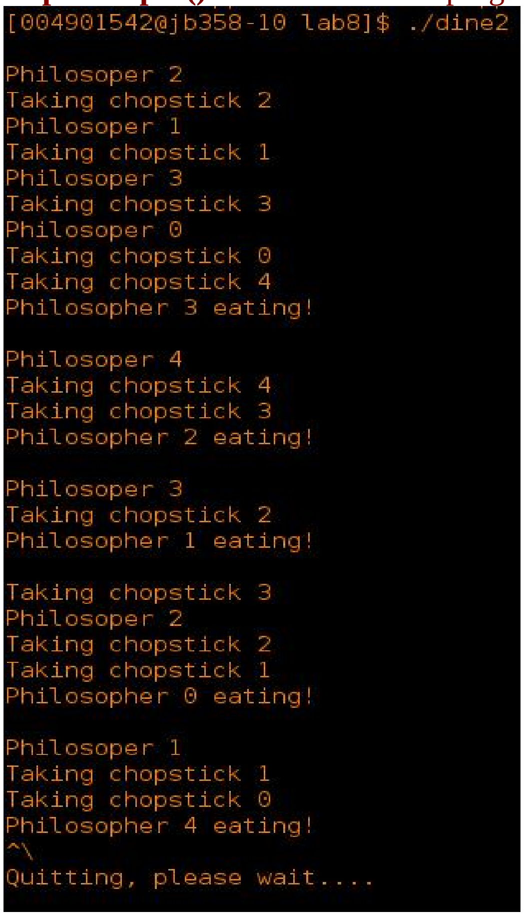
Only one philosopher is allowed to eat at one time.

2. a) Compile and run *dine2.cpp*, and repeat the experiment as above. What is the maximum number of philosopers who can eat simultaneously?

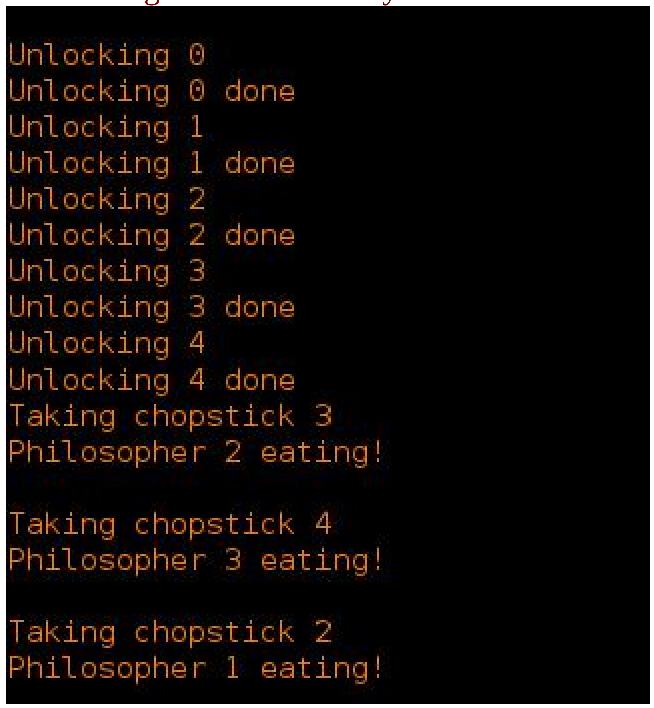
```
[004901542@jb358-10 lab8]$ ./dine2

Philosoper 2
Taking chopstick 2
Taking chopstick 3
Philosopher 2 eating!

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

Taking chopstick 2
Philosopher 1 eating!

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

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

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

Philosoper 0
Taking chopstick 0
Philosoper 2
Philosoper 3
Taking chopstick 3
Taking chopstick 4
Philosopher 3 eating!

Philosoper 4
Taking chopstick 4
Taking chopstick 2
Taking chopstick 3
Philosopher 2 eating!

Taking chopstick 1
```

The maximum number of philosophers that can eat at any time, simultaneously is 2.

b) Add a delay statement like *SDL_Delay (rand() % 2000);* right after the *take_chops(l)* statement in the **philosoper()** function. Run the program for a longer time. What do you observe?



```
[004901542@jb358-10 lab8]$ ./dine2

Philosoper 2
Taking chopstick 2
Philosoper 1
Taking chopstick 1
Philosoper 3
Taking chopstick 3
Philosoper 0
Taking chopstick 0
Taking chopstick 4
Philosopher 3 eating!

Philosoper 4
Taking chopstick 4
Taking chopstick 3
Philosopher 2 eating!

Philosoper 3
Taking chopstick 2
Philosopher 1 eating!

Taking chopstick 3
Philosoper 2
Taking chopstick 2
Taking chopstick 1
Philosopher 0 eating!

Philosoper 1
Taking chopstick 1
Taking chopstick 0
Philosopher 4 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!

Taking chopstick 4
Philosopher 3 eating!

Taking chopstick 2
Philosopher 1 eating!
```

Deadlock occurs because philosopher 2 and philosopher 3 are waiting for the other philosophers to be done eating and never get a chance to eat.

3) Implement this mechanism as discussed in class and call your program *dine3.cpp*. Repeat the above experiment to see whether deadlock occurs and what the maximum number of philosophers can dine simultaneously.

Code:

```
/*
  dine2.cpp :  mutexes lock chopsticks
  Compile:  g++ -o  dine2 dine2.cpp -lSDL
  Execute:  ./dine2
*/

#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 *chopLock[5];  //locks for chopsticks
bool quit = false;
int nEating = 0;   // number of philosopers eating
int state[5];
SDL_mutex *mutex;

void test ( int i )
{
   if ( state[i] == HUNGRY && state[LEFT] != EATING &&
                state[RIGHT] != EATING ) {
      state[i] = EATING;
      SDL_SemPost ( chopLock[i] );
    }
}

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

void eat( int i )
{
  printf("\nPhilosopher %d eating!\n", i );
  SDL_Delay ( rand() % 2000);
}

void take_chops( int i )
{
  SDL_LockMutex ( mutex );
  printf("\nTaking chopstick %d", i );
  state[i] = HUNGRY;
  test(i);
  SDL_SemPost ( chopLock[i] );
  SDL_UnlockMutex ( mutex );
}

void put_chops( int i )
{

  SDL_LockMutex ( mutex );
  state[i] = THINKING;
  test ( LEFT );
  test ( RIGHT );
  SDL_UnlockMutex ( mutex );
}

int philosopher( void *data )
{
  int i, l, r;
  i = atoi ( (char *) data );
  l = i;   //left
  r = (i+1) % 5;
  while ( !quit ) {
    think( i );
    printf("\nPhilosoper %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 philospers 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];                   //thread identifiers
  const char *names[] = { "0", "1", "2", "3", "4" };

  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;
}
```

```
[004901542@jb358-10 lab8]$ ./dine2

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

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

Philosoper 2
Taking chopstick 2
Philosoper 4
Taking chopstick 4
Taking chopstick 3
Philosopher 2 eating!

Taking chopstick 2
Philosopher 1 eating!

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

Taking chopstick 0
Philosopher 4 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 4
Philosopher 3 eating!

Philosoper 1
Taking chopstick 1
Taking chopstick 2
Philosopher 1 eating!
[004901542@jb358-10 lab8]$
```

XV6 Process Priority

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

```
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int priority;                // Process priority
  // add timestamps and others
  uint createTime;             // process creation time
  int  sleepTime;              // process sleeping time
  int  readyTime;              // process ready (RUNNABLE) time
  int  runTime;                // process running time
  int priority;                // process priority
  int tickcounter;
  char dum[8];
};
```

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

```
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority=10;       //default priority
  p->createTime = ticks;
  p->readyTime = 0;
  p->runTime = 0;
  p->sleepTime = 0;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
```

3. Modify **cps()** in *proc.c* discussed in the last lab to include the printout of the priority like the following

```
cprintf("name \t pid \t state \t priority \n");
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if ( p->state == SLEEPING ) {
      cprintf("%s \t %d  \t SLEEPING \t %d \n ", p->name, p->pid, p->priority );
      processCount++;
    }
    else if ( p->state == RUNNING ) {
      cprintf("%s \t %d  \t RUNNING \t %d \n ", p->name, p->pid, p->priority );
      processCount++;
    }
}
```

4. 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();
```

5. 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;
}
```

6. Add **sys_chpr()** in sysproc.c

```
int
sys_cps ( void )
{
  return cps ();
}

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 );
}
```

7. Test nice using foo

```
$ ps
name     pid     state          priority
init     1       SLEEPING       10
 sh      2       SLEEPING       10
 foo     6       RUNNING        10
 ps      16      RUNNING        10
 Total number of RUNNING and SLEEPING processes: 4
$ nice 6 12
$ ps
name     pid     state          priority
init     1       SLEEPING       10
 sh      2       SLEEPING       10
 foo     6       RUNNING        12
 ps      18      RUNNING        10
 Total number of RUNNING and SLEEPING processes: 4
$
```

Score(20/20): I believe I deserve full credit for this lab because I did all of the required steps and showed my outputs and code. I also got all the desired outputs that were required for the lab.