15-213 Introduction to Computer Systems

Final Exam

May 3, 2005

Name: Model Solution

Andrew User ID: fp

Recitation Section: _____

- This is an open-book exam.
- Notes and calculators are permitted, but not computers.
- Write your answer legibly in the space provided.
- You have 180 minutes for this exam.

Problem	Max	Score
1	**15**	
2	**23**	
3	**21**	
4	**26**	
5	**20**	
6	**15**	
7	**10**	
8	**20**	
Total	**150**	

1. Data Representation (15 points)

An engineer suspected an error in the floating point unit of her processor in the IA32-family. So she wrote her own software implementation of various functions on floating point numbers conforming to the IEEE standard. Below is her implementation of a function `fdouble`, to be applied to single-precision floating point numbers, manipulated as 32-bit integers.

```
typedef int float_t;

float_t fdouble (float_t x) {
  int sign = x & (1 << 31);        /* get sign bit */
  int exp = (x >> 23) & 0xFF;      /* get the biased exponent */
  int frac = x & ((1 << 23) - 1);  /* get the fractional part */

  /* check for NAN or infinity */
  if ____________________________  /* LINE 1 */
    return x;                      /* return x if NAN or infinity */

  if (exp == 0) {                  /* check if denormalized */
    /* double denormalized value */
    ____________________________   /* LINE 2 */
    /* check if overflow into a normalized number */
    if (frac >= (1 << 23)) {
      /* fix fractional part */
      frac = ____________________________  /* LINE 3 */
      exp = 1;                     /* change exponent */
    }
  }
  else { /* normalized */
    /* double normalized value */
    ____________________________       /* LINE 4 */
    /* check if infinity */
    if ____________________________    /* LINE 5 */
      /* fix result */
      ____________________________     /* LINE 6 */
  }

  return sign | (exp << 23) | frac;
}
```

1. (12 pts) Fill in the 6 missing lines to complete the implementation.

```
Line 1:  (exp == 0xFF)
Line 2:  frac = frac << 1
Line 3:  frac & ((1 << 23) - 1)
Line 4:  exp++
Line 5:  (exp == 0xFF)
Line 6:  frac = 0
```

2. (3 pts) Assume `float to_float(int x);` and `int to_int(float x);` let us interpret the bit pattern of an integer as a single-precision floating point number and vice versa. When the engineer tested `fdouble` with the following function

```
void test(float x) {
  float x2 = to_float(fdouble(to_int(x)));
  if (2*x != x2) {
    printf("Not equal!\n");
    exit(0);
  }
}
```

she found some discrepancies, even though her implementation of `fdouble` was correct. Explain these discrepancies in a sentence or two.

Floating point arithmetic in the IA32 architecture is performed in 80-bit extended precision format. On the other hand, `x2` is always computed to 32-bit single precision format. The truncation of the 80-bit answer should always agree with the 32-bit answer, but GCC may generate code that performs the comparison directly in the floating point unit, without first truncating `2*x`. An example would be when `x` is representable in single precision, but `2*x` would be too large, such as 3.0×10^{38}.

2. Assembly Language (23 points)

The following is the assembly (GAS) result of compiling a source program in C.

```
mystery:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %esi
        movl    12(%ebp), %ecx
        pushl   %ebx
        xorl    %ebx, %ebx
        movl    8(%ebp), %esi
        decl    %ecx
        cmpl    %ecx, %ebx
        jge     .L7
.L5:
        movl    (%esi,%ebx,4), %edx
        movl    (%esi,%ecx,4), %eax
        movl    %eax, (%esi,%ebx,4)
        incl    %ebx
        movl    %edx, (%esi,%ecx,4)
        decl    %ecx
        cmpl    %ecx, %ebx
        jl      .L5
.L7:
        popl    %ebx
        popl    %esi
        popl    %ebp
        ret
```

```
void mystery (mystery_t A[], int n) {
  int i = 0;
  int j = _n-1__________________;
  while (_i < j_______________) {
    int temp = _A[i]________________;
    _A[i] = A[j]________;
    _i++________________;
    _A[j] = temp________;
    _j--________________;
  }
}
```

1. (4 pts) Show the association between program variables and registers

C Variable	Register
A	**%esi**
i	**%ebx**
j	**%ecx**
temp	**%edx**

2. (14 pts) Fill in the gaps in the shown C source.

3. (5 pts) Which of the following type definitions are consistent with the assembly code? Circle yes or no.

   (a) Yes No **Yes**

   ```
   typedef int mystery_t;
   ```

   (b) Yes No **Yes**

   ```
   typedef float mystery_t;
   ```

   (c) Yes No **No**

   ```
   typedef double mystery_t;
   ```

   (d) Yes No **Yes**

   ```
   typedef struct {
     int x;
     int y;
   } *mystery_t;
   ```

   (e) Yes No **Yes**

   ```
   typedef union {
     int i;
     float f;
   } mystery_t;
   ```

3. Out-of-Order Execution (21 points)

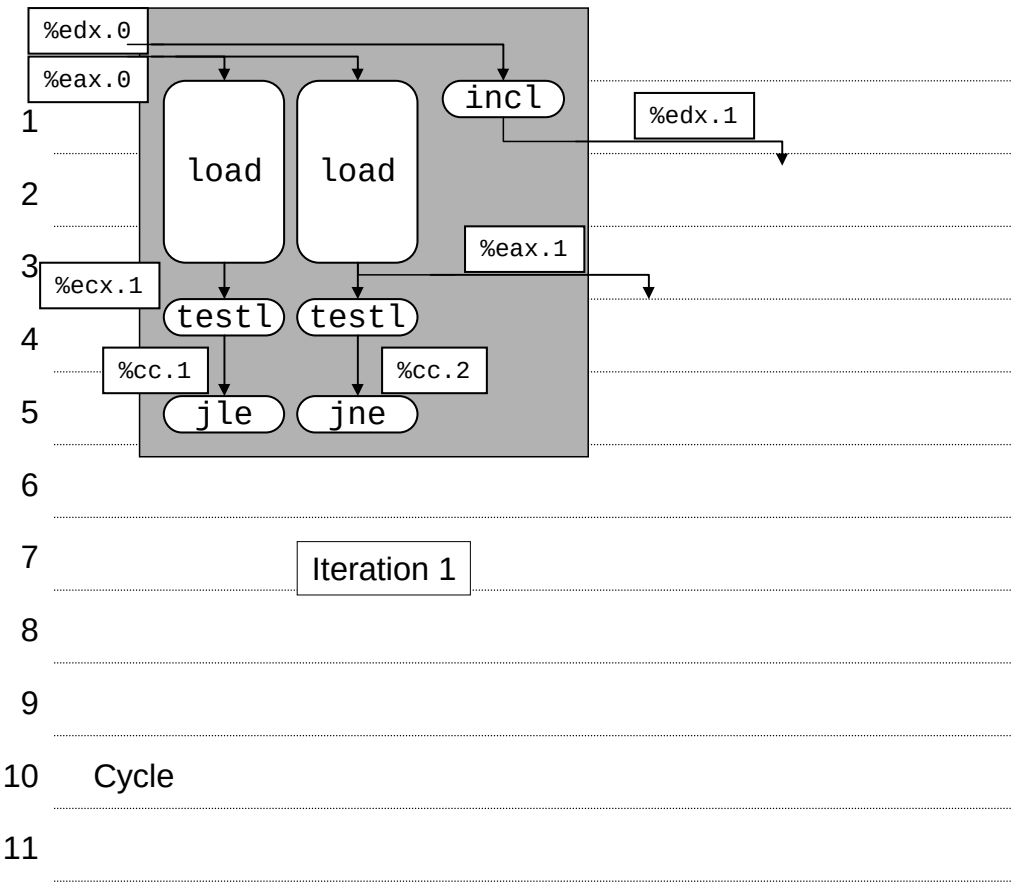
Consider the following program which counts the positive elements in a linked list.

```
void count_pos (List *p) {
  int i = 0;
  while (p) {
    if (p->data > 0)
      i++;
    p = p->next;
  }
  return i;
}
```

On the left we show the main loop of the program in assembly language, generated with `gcc -O2`. The corresponding executing unit operations are given on the right.

.L48:				
	movl 4(%eax), %ecx	load 4(%eax.0)	→	%ecx.1
	testl %ecx, %ecx	testl %ecx.1, %ecx.1	→	cc.1
	jle .L47	jle-not-taken cc.1		
	incl %edx	incl %edx.0	→	%edx.1
.L47:				
	movl (%eax), %eax	load (%eax.0)	→	%eax.1
	testl %eax, %eax	testl %eax.1, %eax.1	→	cc.2
	jne .L48	jne-taken cc.2		

Consider the data dependency diagram on the next page which only shows the first iteration. It is drawn assuming the inner branch is not taken and the outer branch is taken which is justified if we know that most list elements are positive. We assume an issue time of 1 and latency of 3 for a `load` operation that is a cache hit. The diagram ignores any processor resource limitations.



1. (7 pts) Label the empty boxes with the appropriately renamed registers from the execution unit.

2. (5 pts) What is the theoretically optimal CPE for this loop as drawn, assuming no resource limitations and perfect branch prediction?

3.0 CPE

3. (3 pts) The processor has to perform branch prediction for the two branch instructions. Which would easier to predict correctly? Explain your answer.

> The backward branch controlling the loop (`jle .L48`) will be taken almost every time, which will be easy for the processor to predict. The forward branch (`jle .L47`) is data dependent and much more difficult to predict.

4. (3 pts) Explain how a conditional move instruction might improve the efficiency of this code if the list does not contain predominantly positive or negative elements.

> A conditional move could be used to move 1 to a register, depending on the outcome of the comparison of `%ecx` with itself. This value could then be added to the accumulator `%edx`. This would avoid the need for the processor to do branch prediction for the inner branch, and would save the high cost of mispredicted branches in case positive and negative list elements are difficult to forecast.

5. (3 pts) Now assume that the processor has only one load unit, but you may still assume an unbounded number of functional units. What is the theoretically optimal CPE under these assumptions. Briefly explain your answer, using a diagram if you find it helpful.

The answer remains the same: 3.0 CPE. This is because we can postpone the load shown on the left for one time unit, overlapping it with the load shown on the right. The critical path then still takes 3 CPE.

4. Cache Memory (26 points)

Assume we have a 2-way set associative 1024-byte data cache with 16 byte blocks. We assume the machine uses 32-bit addresses and memory is byte-addressable.

1. (5 pts) Determine the following cache parameters.

   (a) The block offset takes __4__ bits.

   (b) The set index takes __5__ bits.

   (c) The tag takes __23__ bits.

   (d) There are __32__ total sets.

   (e) There are __64__ total cache lines.

Now Consider the following code, where `N` is a compile-time constant. This code sums up the first eight columns of the array.

```
int A[N][N];

int sum8col()
{
  int i,j;
  int sum = 0;
  for (j = 0; j < 8; j++)
    for (i = 0; i < N; i++)
      sum += A[i][j];
  return sum;
}
```

Assume we are on a machine where integers take up 4 bytes. You may also assume that `sum`, `i`, and `j` are held in a register, so that the only data cache accesses are to elements of the array `A`.

Assume that the array `A` starts at `0x800000`.

We consider the cache behavior for `N = 32` and `N = 16`, given an LRU eviction policy. Give your answers in hexadecimal form.

6. (6 pts) First, consider the case where `N = 32`.

   (a) The address of `A[1][0]` is __`0x800080`__.

   (b) With `i = 0` and `j = 0`, a block will be read into the cache containing array elements `A[0][0]`, `A[0][1]`, `A[0][2]`, `A[0][3]`. Will this block be evicted? If so, for what value of `i` and `j` will this block be first evicted?

   No

   Yes, with `i =` __`8`__ and `j =` __`0`__

[**Hint:** Remember that the cache is 2-way set associative, consider the order of the iterations, and keep in mind the LRU eviction policy.]

7. (6 pts) Second, we consider the case where `N = 16`.

   (a) The address of `A[1][0]` is `0x800040`.

   (b) With `i = 0` and `j = 0`, a block will be read into the cache containing array elements `A[0][0]`, `A[0][1]`, `A[0][2]`, `A[0][3]`. Will this block be evicted? If so, for what value of `i` and `j` will this block be first evicted?

   No **No**

   Yes, with `i =` _____ and `j =` _____

8. (3 pts) If we changed the order of iterations to

```
int A[N][N];

int sum8col()
{
  int i,j;
  int sum = 0;
  for (i = 0; i < N; i++)
    for (j = 0; j < 8; j++)
      sum += A[i][j];
  return sum;
}
```

   would the answers change for `N = 32`? Circle Yes or No. **No**

9. (3 pts) Would the answers change for `N = 16`? Circle Yes or No. **No**

10. (3 pts) In general, is the first (`j` outermost) or second (`i` outermost) order of iteration preferable? Explain your answer briefly.

In general, the second order of iteration (`i` outermost) is preferable because it has better locality given the manner that arrays are laid out in memory. For example, the first four array elements would be processed together and then never used again, independently of the number `N`.

5. Signals (20 points)

Consider the following code, which has been written with the assumption that an unpredictable number of `SIGINT` interrupts can arrive asynchronously.

```
 1  int i = 1;
 2
 3  void handler (int sig) {
 4    i = 0;
 5  }
 6
 7  int main() {
 8    int j;
 9    sigset_t s;
10    sigemptyset(&s);
11    sigaddset(&s, SIGINT);
12    signal(SIGINT, handler);
13    for (j = 0; j < 100; j++) {
14      i++;
15    }
16    printf("i = %d\n", i);
17    exit(0);
18  };
```

Now we consider the following values for `i` that may be printed at the `printf` command.

$$0, 1, 100, 101$$

For each question, indicate if and where the given calls to `sigprocmask` need to be inserted in order to obtain precisely the indicated set of possible outputs among 0, 1, 100, 101. Note that any given run prints just one value, and that the program may also print other values, but we are only interested in 0, 1, 100, and 101.

1. (5 pts) All of 0, 1, 100, 101. **(a)**

   (a) Neither needs to be inserted (leave next two questions blank).

   (b) Insert `sigprocmask(SIG_BLOCK, &s, 0);` right after line ____.
   Insert `sigprocmask(SIG_UNBLOCK, &s, 0);` right after line ____.

2. (5 pts) Just 0 and 101, but not 1, or 100. **(b), 11 and 15**

   (a) Neither needs to be inserted (leave next two questions blank).

   (b) Insert `sigprocmask(SIG_BLOCK, &s, 0);` right after line ____.
   Insert `sigprocmask(SIG_UNBLOCK, &s, 0);` right after line ____.

3. (5 pts) Just 101, but not 0, 1, or 100. **(b), 11 and 16**

   (a) Neither needs to be inserted (leave next two questions blank).

   (b) Insert `sigprocmask(SIG_BLOCK, &s, 0);` right after line ____.
   Insert `sigprocmask(SIG_UNBLOCK, &s, 0);` right after line ____.

4. (5 pts) Just 100 and 101, but not 0, 1. **(b), 12 and 16**

   (a) Neither needs to be inserted (leave next two questions blank).

   (b) Insert `sigprocmask(SIG_BLOCK, &s, 0);` right after line ____.
   Insert `sigprocmask(SIG_UNBLOCK, &s, 0);` right after line ____.

6. Network Programming (15 points)

In this problem we look at a simple client/server system for a spell-checking service. We first consider the client, then the worker thread on the server, then the main server program. For simplicity, we assume the service runs on 128.2.222.158, port 3340 and the strings being spell-checked never exceed `MAXBUF` in length. We forego some error checking for the sake of brevity.

In each case we first present a program that you should read carefully and then answer the questions. We begin with the client program.

```
#include "csapp.h"
#define DEFAULT_PORT 3340

int main(int argc, char** argv) {
  int fd, num;
  char buf[MAXBUF];
  struct sockaddr_in serveraddr;
  char *msg = argv[1];
  if (strlen(msg) >= MAXBUF-1) exit(1);
  if ((fd = socket(AF_INET, SOCK_STREAM, 0) < 0)) exit(1);
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_addr.s_addr = ntohl(0x8002de9e);
  serveraddr.sin_port = ntohs(DEFAULT_PORT);
  if (connect(fd, (struct sockaddr*)&serveraddr, sizeof(serveraddr)) < 0)
    exit(1);
  if ((num = write(fd, msg, strlen(msg)+1)) < 0) exit(1);
  if ((num = read(fd, buf, MAXBUF-1)) < 0) exit(1);
  buf[num] = 0;
  printf("%s", buf);
  exit(0);
}
```

1. (5 pts) Circle all that apply. **(a), (b)**

   (a) It would be preferable to use `rio_readn` and `rio_writen` instead of `read` and `write` because of possible short counts.

   (b) `ntohl` and `ntohs` should be changed to `htonl` and `htons`, respectively.

   (c) The last character of the server response will be lost.

   (d) We need to close the socket explicitly in order to free system resources.

   (e) We should never read and write on the same socket.

Next, the worker thread for the server. It should be designed so the main server can spawn a separate thread for each request. We assume that `spell_check(buf)` modifies `buf` so that it contains only the incorrectly spelled words in `buf` and returns the number of characters in the resulting string.

```
#include "csapp.h"
#define DEFAULT_PORT 3340

int bytes_served = 0;

void* spell_thread(void *arg) {
  int n;
  int clientfd = *(int *)arg;
  char buf[MAXBUF];

  if ((n = rio_readn(clientfd, buf, MAXBUF)) < 0)
    return NULL;
  bytes_served += n;
  n = spell_check(buf);
  rio_writen(clientfd, buf, n);
  close(clientfd);
  return NULL;
}
```

2. (5 pts) Circle all that apply. **(a), (c), (d)**

   (a) The address passed to `spell_thread` should not be allocated on the stack, because that could create a race condition.

   (b) The `rio_readn` should be contained in a loop rather than an `if` because of the possibility of short counts from a socket.

   (c) The thread may run detached or not.

   (d) There should be a mutex to protect `bytes_served`.

   (e) The `spell_check` function should protect `buf` with a mutex.

Finally, the main server function.

```
int main(int argc, char **argv) {
  int listenfd, clientlen, optval=1;
  struct sockaddr_in serveraddr, clientaddr;
  if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) exit(1);
  if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                 (const void *)&optval, sizeof(int)) < 0) exit(1);
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
  serveraddr.sin_port = htons(DEFAULT_PORT);
  if (bind(listenfd, (struct sockaddr*)&serveraddr, sizeof(serveraddr)) < 0)
    exit(1);
  if (listen(listenfd, 5) < 0) exit(0);
  clientlen = sizeof(clientaddr);
  while (1) {
    int clientfd;
    pthread_t thread;
    if ((clientfd = accept(listenfd, (struct sockaddr*)&clientaddr,
                           &clientlen)) < 0)
      exit(1);
    pthread_create(&thread, NULL, spell_thread, (void *)&clientfd);
    pthread_detach(thread);
  }
}
```

3. (5 pts) Circle all that apply. **(b)**

   (a) The calls to `bind` and `listen` are in the wrong order.

   (b) The call to `pthread_create` creates a race condition.

   (c) The calls to `htonl` and `htons` should be replaced by `ntohl` and `ntohs`, respectively.

   (d) We should call `pthread_join(thread)` instead of `pthread_detach(thread)` in order to avoid consuming unnecessary system resources.

   (e) It would make more sense to create processes with `fork` instead of threads with `pthread_create` because there are no shared resources anyway.

7. Memory Allocation (10 points)

In this problem we consider the interaction between `malloc/free` and multiple threads.

1. (5 pts) Circle all that apply. **(a), (b), (d)**

   (a) Multiple threads share the same heap, so we have to consider the possibility of race conditions for `malloc`.

   (b) When a correct program executes, at most one thread should call `free` on any given pointer.

   (c) We do not need to worry about race conditions for `free` because on a correct program, at most one thread will try to call `free` on any given pointer.

   (d) The internal data structures of `malloc` affect where best to place critical regions.

   (e) For a correct program, memory allocated in one thread must always be freed in the same thread.

2. (5 pts) Describe how you would make `malloc` and `free` thread-safe. State first what kind of basic implementation strategy (e.g., segregated free lists) is the starting point of your analysis, what kind of semaphore(s) (e.g., split binary semaphores) you would use, and where in the code you would add them. Your choices do not need to be efficient, just sound.

   Assuming an explicitly linked free list, the simplest solution would be to have a binary semaphore that functions as a mutex. This mutex guarantees that at most one thread is performing a `malloc` or `free` operation at a time. We wait for this semaphore upon entry to `malloc` and release it just before exit. The same applies for the `free` function. This solution could be quite inefficient if the operations take a long time and there are many calls to `malloc` and `free` from multiple threads, but one would want to perform some measurements to see if it is indeed a practical performance bottleneck. More fine-grained solutions are possible, but much trickier, and depend on more details of the implementation.

8. Semaphores (20 points)

In this problem we develop two symmetric solutions to the *Dining Philosophers* problem as introduced in lecture. In this problem, there is a round table with 5 plates. There are also 5 forks positioned between the plates. Philosophers think and occasionally get hungry. When they get hungry they sit down at their place and eat, for which they need the forks on both sides of their plate. When they are sated, they go back to thinking.

The following code describes a simulation of the dining philosophers for one day, assuming they eat three meals a day.

```
sem_t frk[5];

void *philosopher(void *vargp) {
  int i = (int)vargp;
  int j = 0;
  while (j < 3) {
    printf("%d thinking\n", i);
    P(&frk[i]);
    P(&frk[(i+1)%5]);
    printf("%d eating\n", i);
    j++;
    V(&frk[(i+1)%5]);
    V(&frk[i]);
  }
}

int main() {
  int i;
  pthread_t tid;
  for (i = 0; i < 5; i++)
    sem_init(&frk[i], 0, 1);
  for (i = 0; i < 5; i++)
    Pthread_create(&tid, NULL, philosopher, (void *)i);
  Pthread_exit(NULL);
}
```

1. (3 pts) Explain the protocol followed by each philosopher in terms of thinking, eating, picking up, or putting down a fork which is embodied in the code.

   For three iterations, each philosopher performs the following:

   (a) think

   (b) pick up fork to his right

   (c) pick up fork to his left

   (d) eat

   (e) put down fork to his left

   (f) put down fork to his right

   When a fork is not available, he will simply wait until it becomes available.

2. (3 pts) Explain briefly why this simulation can deadlock.

   The simulation can deadlock because all 5 philosophers may sit down and the table and pick up the fork to their right. At this point all philosophers will wait for the fork to their left to become available, which it never will. We have a deadlock.
   In terms of the code, semaphores `frk[0]` through `frk[4]` are all `0`, and all 5 threads are at the second `P` command, waiting for one of the semaphores to become `1`.

3. (7 pts) In order to avoid the potential deadlock, we adopt a protocol where we allow at most 4 philosophers to sit down at the table at a time. Add a semaphore to the code to implement this new protocol. Modify the code below by adding new lines to implement this protocol. Indicate clearly where the new code should go. You should not modify existing lines of code. Remember to declare and initialize your semaphore.

```
sem_t frk[5];
sem_t table; /* NEW LINE 1 */
void *philosopher(void *vargp) {
  int i = (int)vargp;
  int j = 0;
  while (j < 3) {

    printf("%d thinking\n", i);
    P(&table);  /* NEW LINE 2 */
    P(&frk[i]);

    P(&frk[(i+1)%5]);

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

    j++;

    V(&frk[(i+1)%5]);

    V(&frk[i]);
    V(&table);  /* NEW LINE 3 */
  }
}

int main() {
  int i;
  pthread_t tid;
  for (i = 0; i < 5; i++)
    sem_init(&frk[i], 0, 1);
  sem_init(&table, 0, 4);  /* NEW LINE 4 */
  for (i = 0; i < 5; i++)
    Pthread_create(&tid, NULL, philosopher, (void *)i);
  Pthread_exit(NULL);
}
```

4. (7 pts) Another way to avoid the deadlock is to force every philosopher to pick up the forks to his right and left simultaneously (that is, atomically). Modify the code below by adding new lines to implement this protocol. Indicate clearly where the new code should go. You should not modify existing lines of code.

```
sem_t frk[5];
sem_t pickup;  /* NEW LINE 1 */
void *philosopher(void *vargp) {
  int i = (int)vargp;
  int j = 0;
  while (j < 3) {

    printf("%d thinking\n", i);
    P(&pickup);  /* NEW LINE 2 */
    P(&frk[i]);

    P(&frk[(i+1)%5]);
    V(&pickup);  /* NEW LINE 3 */
    printf("%d eating\n", i);

    j++;

    V(&frk[(i+1)%5]);

    V(&frk[i]);

  }
}

int main() {
  int i;
  pthread_t tid;
  for (i = 0; i < 5; i++)
    sem_init(&frk[i], 0, 1);
  sem_init(&pickup, 0, 1);  /* NEW LINE 4 */
  for (i = 0; i < 5; i++)
    Pthread_create(&tid, NULL, philosopher, (void *)i);
  Pthread_exit(NULL);
}
```