CC3301 Examen

2.5 horas

Julio de 2010

Pregunta 1 (Conceptos)

- (C)

  Queremos hacer una función que retorne el valor del bit más significativo de un entero sin signo (0 o 1). Alumnos del curso proponen varias soluciones. Discuta si están buenas y porqué. Muestre cuales son portables a cualquier arquitectura (distintas representaciones de enteros y tamaños) y cuales no.

```
int b1(int i) {
  return (i < 0);
}
```

```
int b2(int i) {
  return(i >>31);
}
```

```
int b3(int i) {
  return(i&(1<<31);
}
```

```
int b4(int i) {
  unsigned char *p=&i;
  return *p >> 7;
}
```

- (señales)

  Cuando uso threads, el significado exacto de una señal es confuso: ¿interrumpe el thread en curso solamente? ¿detiene todos los threads en curso? Discuta qué cree ud que es lo que debiera hacer una señal en ese caso y justifique.

- (sistema de archivos)

  Explique por qué no se puede hacer un link duro a un directorio (carpeta).

- (procesos)

  Quiero escribir un servidor que implementa un servicio de copia de archivos. El cliente le envía un nombre de archivo y el servidor le envía su contenido por el socket. Los archivos pueden ser bastante grandes, pero no se espera más de unos 100 clientes simultáneos. Explique si usted usaría threads, procesos o select para implemetarlo y por qué.

- (threads)

  Explique qué problemas aparecen al usar threads y usar mecanismos de sincronización entre ellas.

- (perl)

  Explique cuando se justifica escribir programas en lenguajes como perl, en vez de lenguajes como C y Java.

Pregunta 2 (Unix)

Este es el código simplificado de un programa que hace un: `ls | more` y espera que termine.

```
main() {
    int lspid, morepid, pid;
    int status = 0;
    int fds[2];

    pipe(fds);
    lspid = fork();
    if(lspid == 0) { /* ls */
        /* Cerrar el extremo read del pipe que no voy a usar */
        close(fds[0]);
        /* Asigno: 1 (stdout) = fds[1] (lado de escritura del pipe) */
        close(1); dup(fds[1]);
        /* Cerrar la copia que me queda sobre el pipe o no tendre' EOF */
        close(fds[1]);
        execl("/bin/ls", "ls", "-l", 0);
        exit(-1);
    }

    morepid = fork();
    if(morepid == 0) { /* more */
        /* Cerrar el extremo write del pipe que no voy a usar */
        close(fds[1]);
        /* Asigno: 0 (stdin) = fds[0] (lado de lectura del pipe) */
        close(0); dup(fds[0]);
        /* Cerrar la copia que me queda sobre el pipe o no tendre' EOF */
        close(fds[0]);
        execl("/bin/more", "more", 0);
        exit(-1);
    }

   /* Como padre comun, cierro el pipe, ambos extremos (yo no lo uso) */
   close(fds[0]); close(fds[1]);

   /* como buen padre, espero la muerte de todos mis hijos */
   while((pid = wait(&status)) != -1);
}
```

Se le pide modificar este código para que todo lo que escribe ls en el pipe quede copiado en un archivo llamado "`out`". Para hacer eso, se debe crear otro proceso y otro pipe, para que lea desde el pipe conectado a ls, escriba en el archivo y escriba en el otro pipe conectado a more. Deben dejar todo bien hecho para que los procesos mueran al terminar los datos o morir alguno de los procesos. Suponga que no hay errores ni fallas de los comandos.

Pregunta 3 (Redes)

Este es el código de un cliente del servidor de eco:

```
#define BUF_SIZE 10
main() {
   int s;
   int cnt, n, size = BUF_SIZE;
   char buf[BUF_SIZE];

   s = j_socket();

   if(j_connect(s, "localhost", 1818) < 0) {
fprintf(stderr, "connection refused\n");
exit(1);
   }

   write(s, "hola", 5);
   n = 0;
   while((cnt=read(s, buf+n, size-n)) > 0) {
n += cnt;
if(n >= 5) break;
   }
   printf("%s\n", buf);
   close(s);
}
```

Modifíquelo, para hacer un cliente que monitorea varios servidores de eco revisando si responden correctamente una vez por minuto. Cuando un servidor no responde, o responde erróneamente, debe escribir en su salida de errores un mensaje avisando. Cada vez que esto ocurre, debe cerrar esa conexión y reintentar conectarse una vez (o sea, reintentará una vez por minuto). Suponga que conoce los datos a monitorear en una constante y dos variables, por ejemplo:

```
#define SERVERS 3
char *hostnames[SERVERS] = {"localhost", "localhost", "localhost"};
int ports[SERVERS] = {1818, 1819, 1820};
```

Debe conectarse a los SERVERS servidores y revisar que respondan.

Pregunta 4 (Sincronización)

Esta es la solución clásica y correcta de múltiples productores y consumidores usando condiciones:

```
void putbox(BOX *b, char c)
{
    pthread_mutex_lock(&b->mutex);
    while(b->nvacios == 0)
        pthread_cond_wait(&b->vacios, &b->mutex);
    b->buf[b->in] = c;
    b->in = (b->in+1)%NBUFS;
    b->nllenos++; b->nvacios--;
    pthread_cond_signal(&b->llenos);
    pthread_mutex_unlock(&b->mutex);
}

char getbox(BOX *b)
{
    char c;

    pthread_mutex_lock(&b->mutex);
    while(b->nllenos == 0)
        pthread_cond_wait(&b->llenos, &b->mutex);
    c = b->buf[b->out];
    b->out = (b->out+1)%NBUFS;
    b->nllenos--; b->nvacios++;
    pthread_cond_signal(&b->vacios);
    pthread_mutex_unlock(&b->mutex);
    return(c);
}
```

Modifique esta solución para incluir una tercera primitiva asociada a las cajas:

```
void closebox(BOX *b);
```

Esta primitiva debe actuar parecido al close() de un pipe: una vez que una caja está cerrada:

- los gets deben satisfacerse mientras haya datos en los buffers y luego entregar EOF a todos los que hagan get.
- los puts deben matar al thread llamador.

También debe despertar a todos los threads que están bloqueados esperando por la caja. Para matar un thread use `pthread_exit(NULL);`.

Revise que su solución soporte paralelismo entre productor y consumidor y luego entre múltiples productores y múltiples consumidores. Explique.