Sistemas Operativos - 1er Parcial

[redacted]
3 de Octubre de 2023

Ej 1) 13/25Modelaste otro problema, pero mas alla de eso
La falta de mutex te puede generar race conditions
Hace signals de mas que permite que mas de 10 clientes puedan entrar a la zona crítica

Ejercicio 1

```
#define N 10
semaphore molinete = sem(1);
semaphore barrera = sem(N);
atomic<int> esperando = 0;
atomic<int> comprando = 0;
bool congestionado = false;
bool liberarMolinete = false;

void usuario() {
    // El molinete controla los usuarios que quieren entrar al sistema para comprar.
    molinete.wait();
    molinete.signal();

    // La barrera solo permite que N=10 usuarios compren concurrentemente.
    // Con la variable atómica esperando contabilizamos cuántos usuarios esperan para comprar.
    esperando.getAndInc();
    barrera.wait();
    esperando.getAndDec();
```

deberia estar en un mutex

```
    // Si somos el usuario número N=10 que está comprando entonces el sistema está congestionado.
    if (comprando.getAndInc() == N-1) {
        congestionado = true;

        // Si el sistema estaba congestionado en la tanda anterior de N=10 usuarios,
        // cuando terminan de entrar los nuevos N=10 usuarios rehabilitamos el molinete
        // asi los nuevos usuarios pueden avanzar a la "sala de espera" (por la barrera).
        // De esta forma ningún usuario puede pasar el molinete antes de que pasen los
        // próximos N=10 usuarios que ya estaban esperando desde antes por la barrera.
        if (liberarMolinete) {
            molinete.signal();
            liberarMolinete = false;
        }
    }

    comprar_ticket();
```

si hay 10 usuarios deberia bloquearse y no permitir que entren mas hasta que se vacie pero esto no sucede aca

```
    if (congestionado) {
        // Como el sistema está congestionado, necesitamos esperar que terminen de
        // comprar los N=10 usuarios antes de habilitar el acceso al resto.
        if (comprando.getAndDec() == 1) {
            congestionado = false;

            // Si hay más de N=10 usuarios esperando, entonces activamos el mecanismo
            // para evitar que otros usuarios se colen.
            if (esperando.get() >= N) {
                // Pedimos acá el molinete y no lo liberamos aún, en efecto bloqueando a
                // todos los usuarios nuevos que llegan al sistema a partir de ahora.
                molinete.wait();
                liberarMolinete = true;
            }
```

```
            // Ahora sí podemos habilitar que pase la próxima tanda de N=10 usuarios.
            barrera.signal(N);
        }
    } else {
        // Si el sistema no está congestionado entonces habilitamos un usuario nuevo
        // para que pase la barrera y compre su ticket.
        comprando.getAndDec();
        barrera.signal();
    }
}
```

hace signals de mas

Ejercicio 2

12/25

Podemos utilizar una cola multi nivel donde los procesos tienen prioridades estáticas, no se mueven entre colas.

- Nivel 0 (prioridad alta): FCFS para procesos interactivos.
- Nivel 1 (prioridad baja): Round-robin para procesos real-time.

los RT siempre son los de mayor prioridad
son los que hacen cosas tras bambalinas podriamos decir
Hubieses invertido el orden

Priorizamos a los procesos interactivos para minimizar la latencia del juego que percibe el jugador. Pero más importante aún, porque asumimos que estos procesos tienen ráfagas muy cortas de CPU, pues solo revisan el teclado o mouse y luego actualizan las estructuras de datos internas del juego. Incluso podrían crear nuevos procesos real-time para las nuevas unidades que construye el jugador. Utilizamos una cola FCFS para responder a las interacciones del usuario en el orden que sucedieron, ya que eso es lo naturalmente esperado en un juego.

Los procesos real-time que controlan a las unidades del juego se procesan con una cola round-robin para intentar garantizar un uso justo del CPU. Podemos llegar a tener miles de unidades en juego, y cada una tiene que actualizar su estado interno constantemente (posición, vida, etc). Para optimizar el uso del CPU, los procesos real-time pueden ceder su quantum antes de tiempo si no están haciendo nada, o están muy lejos del combate fuera de la pantalla, o son unidades inmovilizadas (por ejemplo por algún poder u arma del enemigo).

También podríamos considerar utilizar una cola EDF (earliest deadline first). Cada proceso define su deadline en base a la acción que está realizando. Si la unidad se está moviendo o atacando debemos asegurarnos de obtener el CPU pronto, caso contrario podemos esperar más tiempo antes de obtener el CPU. Pero como pueden haber miles de unidades en juego, ordenar constantemente los procesos por su deadline podría resultar en peor performance, por eso optamos por round-robin.

Tambien te sirve el EDF , pero de vuelta deberia estar como mayor prioridad el manejo de los RT

Ejercicio 3

LRU

Least recently used mantiene una cola de desalojo ordenada por el último tiempo de acceso a la página. Cuando necesitamos desalojar una página, elegimos la que se usó hace más tiempo, con la suposición de que si no se accede hace mucho tiempo entonces no es una página muy usada por los procesos ready.

Página solicitada	F1	F2	F3	F4	Pafe fault	Cola de desalojo
Estado inicial	0	1	2	3	-	0 1 2 3
2	0	1	2	3	no	0 1 3 2
1	0	1	2	3	no	0 3 2 1
7	7	1	2	3	si	3 2 1 7
3	7	1	2	3	no	2 1 7 3
0	7	1	0	3	si	1 7 3 0
5	7	5	0	3	si	7 3 0 5
2	2	5	0	3	si	3 0 5 2
1	2	5	0	1	si	0 5 2 1
2	2	5	0	1	no	0 5 1 2
9	2	5	9	1	si	5 1 2 9
5	2	5	9	1	no	1 2 9 5

Página solicitada	F1	F2	F3	F4	Pafe fault	Cola de desalojo
7	2	5	9	7	si	2 9 5 7
3	3	5	9	7	si	9 5 7 3
8	3	5	8	7	si	5 7 3 8
5	3	5	8	7	no	7 3 8 5

Tasa promedio de page faults: `11 pages faults / 15 accesos = 0.73`

tuviste 9 page faults
la tasa de PF es 9/15

Second chance

Esta política utiliza una cola de desalojo FIFO con el agregado del second chance. Al cargar una página en memoria, se inicializa un bit de accedido en 0. Cada vez que se accede a una página se marca su bit accedido en 1 (en la tabla lo marcamos con un *). Cuando tenemos que desalojar una página, tomamos el tope de la cola FIFO, pero si está marcada, entonces le sacamos el bit de accedido y la reinsertamos al final de la cola. En efecto, le dimos a la página una segunda oportunidad. Repetimos este proceso hasta encontrar una página que no esté marcada.

Página solicitada	F1	F2	F3	F4	Pafe fault	Cola de desalojo
Estado inicial	0	1	2	3	-	0 1 2 3
2	0	1	2	3	no	0 1 2* 3
1	0	1	2	3	no	0 1* 2* 3
7	7	1	2	3	si	1* 2* 3 7
3	7	1	2	3	no	1* 2* 3* 7
0	0	1	2	3	si	2* 3* 7 1 -> 3* 7 1 2 -> 7 1 2 3 -> 1 2 3 0
5	0	5	2	3	si	2 3 0 5
2	0	5	2	3	no	2 3* 0 5
1	0	5	1	3	si	3* 0 5 1
2	2	5	1	3	si	0 5 1 3 -> 5 1 3 2
9	2	9	1	3	si	1 3 2 9
5	2	9	5	3	si	3 2 9 5
7	2	9	5	7	si	2 9 5 7
3	3	9	5	7	si	9 5 7 3
8	3	8	5	7	si	5 7 3 8
5	3	8	5	7	no	5* 7 3 8

la que tiene SC es la 2
luego arrastrás el error

Tasa promedio de page faults: `10 pages faults / 15 accesos = 0.66`

Ejercicio 4

a)

```c
#define READ 0
#define WRITE 1

void child(int m, int fdw) {
    // Pedimos el número de labo y lo escribimos en el pipe para que el proceso padre
    // pueda leerlo más tarde.
    int labo = dameLabo(getpid(), m);
    write(fdw, &labo, sizeof(labo));
    exit(EXIT_SUCCESS);
}

void main(int argc, char* argv[]) {
    if (argc != 3) {
```

```
    fprintf(STDERR_FILENO, "%s <n> <m>\n", argv[0]);
    exit(EXIT_FAILURE);
}

int n = atoi(argv[1]);
int m = atoi(argv[2]);

// Creamos n pipes para comunicarnos con nuestros hijos.
// Los pipes son unidireccionales, pero solo necesitamos 1 por cada hijo
// para la comunicación hijo -> padre cuando el hijo informa el labo asignado.
int pipes[n][2];
for (int i = 0; i < n; i++) {
    if (pipe(pipes[i]) < 0) {
        fprintf(STDERR_FILENO, "error creando pipe\n");
        exit(EXIT_FAILURE);
    }
}

// Creamos los n procesos hijos.
pid_t children[n];
for (int i = 0; i < n; i++) {
    children[i] = fork();
    if (children[i] < 0) {
        fprintf(STDERR_FILENO, "error creando proceso hijo\n");
        exit(EXIT_FAILURE);
    }
    if (children[i] == 0) {
        // Cerramos todos los pipes que no vamos a usar.
        // Esto no es realmente necesario porque en cada pipe vamos a escribir una
        // cantidad fija de bytes, entonces no necesitamos generar las condiciones
        // necesarias para que el SO nos retorne un EOF cuando hacemos read().
        for (int j = 0; j < n; j++) {
            close(pipes[j][READ]);
            if (j != i) close(pipes[j][WRITE]);
        }
        // Los procesos hijos heredan una copia de la memoria del padre, incluyendo
        // los file descriptors. En particular ya tenemos el valor de m en el proceso
        // hijo, no hace falta pasarlo por un pipe padre -> hijo.
        child(m, pipes[i][WRITE]);

        // Nunca volvemos a main() en los procesos hijos porque hacen un exit().
    }
}

// Ya creamos los n hijos que van a escribir sus labos en cada uno de sus pipes.
// Leemos los labos asignados y lo imprimimos junto al pid del hijo.
for (int i = 0; i < n; i++) {
    // Si bien leemos los pipes en orden (0, 1, ..., n-1), no tenemos garantías de que
    // los resultados esten disponibles en ese orden. No obstante, si el hijo aún no
    // escribió su labo en el pipe, el llamado a read() va a bloquear al padre hasta
    // que haya sizeof(int) bytes disponibles en el pipe. Esto siempre va a suceder por
    // construcción del programa (salvo errores fatales por parte del SO), por eso el
    // padre eventualmente va a terminar.
    int labo;
    read(pipes[i][READ], &labo, sizeof(labo));
    printf("%d: %d\n", children[i], labo);
```

```
        // Hacemos wait por alguno de los hijos que terminó para evitar que quede zombie.
        // No hay garantías que el pid retornado sea el del hijo i-ésimo, pero no nos
        // importa eso, sino simplemente queremos hacer en total n wait().
        wait(NULL);
    }

    // Podríamos acá cerrar los pipes, pero como el proceso padre ya termina, el SO se
    // va a encargar de hacer eso automáticamente.

    exit(EXIT_SUCCESS);
}
```

b)

Solo se modifica la última parte de `main()`. Omití los comentarios repetidos para ahorrar espacio.

```
#define READ 0
#define WRITE 1

void child(int m, int fdw) {
    int labo = dameLabo(getpid(), m);
    write(fdw, &labo, sizeof(labo));
    exit(EXIT_SUCCESS);
}

void main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(STDERR_FILENO, "%s <n> <m>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int n = atoi(argv[1]);
    int m = atoi(argv[2]);

    int pipes[n][2];
    for (int i = 0; i < n; i++) {
        if (pipe(pipes[i]) < 0) {
            fprintf(STDERR_FILENO, "error creando pipe\n");
            exit(EXIT_FAILURE);
        }
    }

    pid_t children[n];
    for (int i = 0; i < n; i++) {
        children[i] = fork();
        if (children[i] < 0) {
            fprintf(STDERR_FILENO, "error creando proceso hijo\n");
            exit(EXIT_FAILURE);
        }
        if (children[i] == 0) {
            for (int j = 0; j < n; j++) {
                close(pipes[j][READ]);
                if (j != i) close(pipes[j][WRITE]);
            }
            child(m, pipes[i][WRITE]);
        }
```

```
}

// Abrimos el archivo para escritura y reconfiguramos nuestro file descriptor 1 (stdout)
// para que apunte al archivo abierto. De esta forma, cuando nuevaSalida escribe en
// stdout, en realidad va a estar escribiendo en el archivo. Recordemos que los fd
// son heredados por los hijos.
int fd_out = open("resultados.out", O_WRONLY);
dup2(fd_out, STDOUT_FILENO);

for (int i = 0; i < n; i++) {
    // Leemos el labo asignado al hijo i-ésimo.
    int labo;
    read(pipes[i][READ], &labo, sizeof(labo));
    wait(NULL);

    // Creamos un pipe para stdin.
    int fd_in[2];
    if (pipe(fd_in) < 0) {
        fprintf(STDERR_FILENO, "error creando pipe\n");
        exit(EXIT_FAILURE);
    }

    // Creamos el proceso hijo para ejecutar el programa nuevaSalida.
    pid_t nuevaSalida = fork();
    if (nuevaSalida < 0) {
        fprintf(STDERR_FILENO, "error creando proceso hijo\n");
        exit(EXIT_FAILURE);
    }

    if (nuevaSalida == 0) {
        // De forma similar al stdout, reconfiguramos el file descriptor 0 (stdin)
        // para que apunte a este pipe compartido entre padre e hijo. De esta forma
        // podemos desde el padre escribir al pipe y que el hijo lo lea por stdin.
        dup2(fd_in[READ], STDIN_FILENO);

        // Desde el hijo no vamos a escribir en nuestro propio stdin. Pero cerramos
        // el pipe principalmente para que nuevaSalida pueda recibir un EOF cuando
        // cerramos la punta de escritura del pipe desde el padre.
        close(fd_in[WRITE]);

        // Reemplazamos el proceso hijo por la ejecución del programa nuevaSalida.
        // Se mantienen los file descriptors prevamiente configurados.
        char* argv_nuevaSalida = {"nuevaSalida", NULL};
        execvp(argv_nuevaSalida[0], argv_nuevaSalida);

        // Nunca llegamos acá si execvp funcionó correctamente.
    } else {
        // Somos el padre, escribimos en el pipe de stdin el pid del proceso hijo
        // (el que representa un estudiante) junto a su labo asignado.
        fprintf(fd_in[WRITE], "%d %d\n", children[i], labo);

        // Cerramos el pipe.
        close(fd_in[READ]);
        close(fd_in[WRITE]); // Este es el que importa para disparar EOF al hijo.

        // Esperamos que termine nuevaSalida antes de continuar.
```

```
            waitpid(nuevaSalida, NULL, 0);
        }
    }

    exit(EXIT_SUCCESS);
}
```