Manual Técnico

Simulador de Gestor de Procesos para Sistemas Operativos

Universidad Autónoma de Tamaulipas
Materia: Sistemas Operativos
Profesor: Dante Adolfo Muñoz Quintero
Semestre: 7º

Autores:

- Solbes Davalos Rodrigo
- Izaguirre Cortes Emanuel
- Rosales Pereles Denisse Ariadna
- Morales Urrutia Javier Antonio
- Reyes Alejo Emiliano

Arquitectura del Sistema

Visión General

El simulador está implementado como una aplicación monolítica en C que simula un sistema operativo simplificado. La arquitectura sigue un modelo de máquina de estados donde el tiempo avanza discretamente mediante pasos de simulación.

Flujo de Ejecución

1. **Inicialización:**
    - init_system() inicializa todas las estructuras
    - Selección del algoritmo de planificación
    - Configuración de parámetros (quantum para RR)
2. **Bucle Principal:**
    - Interfaz de usuario (menú CLI)

- Procesamiento de comandos
- Ejecución de operaciones

3. **Simulación Paso a Paso:**
    - execute_step() avanza el tiempo
    - Actualización de estados
    - Selección de siguiente proceso
    - Registro de eventos

Estructuras de Datos

1. PCB (Process Control Block)

```
typedef struct {
   int pid;                 // Process ID único
   ProcessState state;         // Estado actual del proceso
   int priority;              // Prioridad (mayor = más prioritario)
   Resources allocated;        // Recursos actualmente asignados
   Resources needed;           // Recursos necesarios
   int burst_time;            // Tiempo total de ejecución
   int remaining_time;        // Tiempo restante
   int arrival_time;          // Tiempo de llegada al sistema
   int completion_time;        // Tiempo de finalización
   int waiting_time;          // Tiempo acumulado en espera
   int turnaround_time;        // Tiempo de retorno
   int quantum_remaining;      // Quantum restante (RR)
   TerminationCause termination_cause;  // Causa de terminación
} PCB;
```

Descripción de Campos:

- **pid:** Identificador único asignado secuencialmente
- **state:** Uno de {READY, RUNNING, WAITING, TERMINATED}
- **priority:** Usado para desempate y extensiones futuras
- **allocated/needed:** Control de recursos (CPU y memoria)
- **burst_time:** Total de unidades de tiempo de ejecución
- **remaining_time:** Contador descendente hasta 0
- **arrival_time:** Timestamp de creación
- **completion_time:** Timestamp de terminación
- **waiting_time:** Acumulador de tiempo en READY/WAITING
- **turnaround_time:** completion_time - arrival_time
- **quantum_remaining:** Solo para Round Robin
- **termination_cause:** Razón de finalización

2. Resources

```
typedef struct {
  int cpu;          // 0 o 1 (sistema monoprocesador)
  int memory_blocks;   // Bloques de 1GB (máx 4)
} Resources;
```

Uso:

- system.available: Recursos libres en el sistema
- system.total: Capacidad total del sistema
- process.allocated: Recursos asignados al proceso
- process.needed: Recursos que requiere el proceso

3. Message

```
typedef struct {
   int sender_pid;      // PID del emisor
   int receiver_pid;    // PID del receptor
   char content[256];   // Contenido del mensaje
   bool delivered;      // Estado de entrega
} Message;
```

Características:

- Comunicación asíncrona
- Almacenamiento persistente hasta entrega
- FIFO para mensajes al mismo receptor

4. TLight

```
typedef struct {
   int id;                  // Identificador del semáforo
   int value;               // Valor actual
   int waiting_pids[MAX_PROCESSES]; // Cola de espera
   int waiting_count;       // Número de procesos esperando
} Semaphore;
```

Propiedades:

- Contador con operaciones atómicas simuladas
- Cola FIFO para procesos bloqueados
- Valor puede ser negativo (indica procesos en espera)

5. SharedBuffer

```
typedef struct {
   int items[BUFFER_SIZE];  // Elementos almacenados
```

```
  int count;            // Número de elementos
  int in;               // Índice de inserción
  int out;              // Índice de extracción
} SharedBuffer;
```

Implementación:

- Buffer circular de tamaño fijo
- Usado en problema productor-consumidor
- Sincronizado mediante semáforos

6. LogEntry

```
typedef struct {
  int time;         // Timestamp del evento
  int pid;          // PID relacionado (-1 para eventos del sistema)
  char event[256];  // Descripción del evento
} LogEntry;
```

Propósito:

- Auditoría de operaciones
- Debugging
- Análisis post-ejecución

7. System (Estructura Global)

```
typedef struct {
  // Gestión de Procesos
  PCB processes[MAX_PROCESSES];
  int process_count;
  int next_pid;
```

```
// Gestión de Recursos
Resources available;
Resources total;

// Comunicación
Message messages[MAX_MESSAGES];
int message_count;

// Sincronización
Semaphore semaphores[MAX_SEMAPHORES];
int semaphore_count;
SharedBuffer buffer;
int mutex, empty, full;  // IDs de semáforos para prod-cons

// Planificación
SchedulingAlgorithm algorithm;
int quantum;
int current_time;
int running_pid;

// Logs y Estadísticas
LogEntry logs[MAX_LOG_ENTRIES];
int log_count;
int total_processes_completed;
int total_waiting_time;
```

```
    int total_turnaround_time;
    int cpu_busy_time;
} System;
```

Variable Global:

```
System sys;  // Instancia única del sistema
```

Módulos y Funciones

Módulo: Inicialización

void init_system()

Propósito: Inicializa todas las estructuras del sistema.

Pseudocódigo:

```
function init_system():
    sys.process_count = 0
    sys.next_pid = 1
    sys.available.cpu = MAX_CPU (1)
    sys.available.memory_blocks = MAX_MEMORY_BLOCKS (4)
    sys.total = sys.available
    sys.message_count = 0
    sys.semaphore_count = 0
    sys.current_time = 0
    sys.running_pid = -1
    sys.log_count = 0
    inicializar estadísticas a 0
    sys.quantum = 2  // valor por defecto
    inicializar buffer compartido
    agregar log "Sistema inicializado"
```

Complejidad: O(1)

Módulo: Gestión de Procesos

int create_process(int burst_time, int priority, int memory_blocks)

Propósito: Crea un nuevo proceso en el sistema.

Parámetros:

- burst_time: Tiempo total de ejecución (>0)
- priority: Nivel de prioridad (típicamente 1-10)
- memory_blocks: Bloques de memoria necesarios (1-4)

Retorno:

- PID del proceso creado
- -1 en caso de error

Algoritmo:

```
function create_process(burst_time, priority, memory_blocks):
  if sys.process_count >= MAX_PROCESSES:
    return error

  if memory_blocks > MAX_MEMORY_BLOCKS:
    return error

  idx = sys.process_count
  p = &sys.processes[idx]

  // Inicializar PCB
  p.pid = sys.next_pid++
  p.state = READY
```

```
    p.priority = priority
    p.allocated = {0, 0}
    p.needed = {1, memory_blocks}
    p.burst_time = burst_time
    p.remaining_time = burst_time
    p.arrival_time = sys.current_time
    p.quantum_remaining = sys.quantum
    inicializar tiempos a 0

    sys.process_count++
    agregar log
    return p.pid
```

Complejidad: O(1)

Efectos Secundarios:

- Incrementa process_count
- Incrementa next_pid
- Añade entrada al log

void terminate_process(int pid, TerminationCause cause)

Propósito: Finaliza un proceso y libera sus recursos.

Algoritmo:

```
function terminate_process(pid, cause):
    idx = find_process_by_pid(pid)
    if idx == -1:
        return error
```

```
p = &sys.processes[idx]

if p.state == TERMINATED:
    return error

if p.state == RUNNING:
    sys.running_pid = -1

release_resources(pid)

p.state = TERMINATED
p.termination_cause = cause
p.completion_time = sys.current_time
p.turnaround_time = p.completion_time - p.arrival_time

// Actualizar estadísticas globales
sys.total_processes_completed++
sys.total_waiting_time += p.waiting_time
sys.total_turnaround_time += p.turnaround_time

agregar log
```

Complejidad: O(1)

void suspend_process(int pid)

Propósito: Suspende temporalmente un proceso.

Efecto: Cambia el estado a WAITING. Los recursos permanecen asignados.

Complejidad: O(n) por búsqueda de PID

void resume_process(int pid)

Propósito: Reanuda un proceso suspendido.

Precondición: El proceso debe estar en estado WAITING.

Efecto: Cambia el estado a READY.

Módulo: Gestión de Recursos

bool check_deadlock_prevention(int pid, Resources req)

Propósito: Implementa prevención de deadlock mediante verificación de recursos disponibles.

Algoritmo:

```
function check_deadlock_prevention(pid, req):
  if req.cpu > sys.available.cpu:
    return false
  if req.memory_blocks > sys.available.memory_blocks:
    return false
  return true
```

Política: No permitir solicitudes que excedan recursos disponibles (prevención simple).

Complejidad: O(1)

bool request_resources(int pid, Resources req)

Propósito: Asigna recursos a un proceso.

Algoritmo:

```
function request_resources(pid, req):
  idx = find_process_by_pid(pid)
```

```
if idx == -1:
   return false

p = &sys.processes[idx]

// Verificar prevención de deadlock
if not check_deadlock_prevention(pid, req):
   p.state = WAITING
   agregar log "solicitud denegada"
   return false

// Verificar disponibilidad
if req.cpu <= sys.available.cpu and
  req.memory_blocks <= sys.available.memory_blocks:

   // Asignar recursos
   sys.available.cpu -= req.cpu
   sys.available.memory_blocks -= req.memory_blocks
   p.allocated.cpu += req.cpu
   p.allocated.memory_blocks += req.memory_blocks

   agregar log "recursos asignados"
   return true

// No hay suficientes recursos
p.state = WAITING
```

```
  agregar log "en espera de recursos"
  return false
```

Complejidad: O(n) por búsqueda de PID

void release_resources(int pid)

Propósito: Libera todos los recursos asignados a un proceso.

Algoritmo:

```
function release_resources(pid):
  idx = find_process_by_pid(pid)
  if idx == -1:
    return

  p = &sys.processes[idx]

  sys.available.cpu += p.allocated.cpu
  sys.available.memory_blocks += p.allocated.memory_blocks

  agregar log "recursos liberados"

  p.allocated.cpu = 0
  p.allocated.memory_blocks = 0
```

Complejidad: O(n)

Módulo: Planificación

void select_next_process()

Propósito: Selecciona el siguiente proceso a ejecutar según el algoritmo.

Algoritmo (FCFS):

```
function select_next_process_FCFS():
  selected_idx = -1

  for i from 0 to sys.process_count - 1:
    if sys.processes[i].state == READY:
      if selected_idx == -1 or
        sys.processes[i].arrival_time < sys.processes[selected_idx].arrival_time:
        selected_idx = i

  if selected_idx != -1:
    asignar_proceso(selected_idx)
```

Complejidad: O(n)

Algoritmo (Round Robin):

```
function select_next_process_RR():
  selected_idx = -1

  // Buscar proceso READY con menor PID (simula cola circular)
  for i from 0 to sys.process_count - 1:
    if sys.processes[i].state == READY:
      if selected_idx == -1:
        selected_idx = i
      else if sys.processes[i].pid < sys.processes[selected_idx].pid:
        selected_idx = i

  if selected_idx != -1:
```

```
    sys.processes[selected_idx].quantum_remaining = sys.quantum
    asignar_proceso(selected_idx)
```

Complejidad: O(n)

Nota: La selección por menor PID en RR es una simplificación. Una implementación más realista usaría una cola circular explícita.

void execute_step()

Propósito: Ejecuta un paso de simulación (avanza el tiempo en 1 unidad).

Algoritmo:

```
function execute_step():
  sys.current_time++

  // Ejecutar proceso actual
  if sys.running_pid != -1:
    idx = find_process_by_pid(sys.running_pid)
    if idx != -1:
      p = &sys.processes[idx]

      if p.state == RUNNING:
        p.remaining_time--
        sys.cpu_busy_time++

        if sys.algorithm == ROUND_ROBIN:
          p.quantum_remaining--

        agregar log "ejecutando"
```

```
        // Verificar terminación
        if p.remaining_time <= 0:
            terminate_process(p.pid, NORMAL)
            sys.running_pid = -1

        // Verificar quantum agotado (RR)
        else if sys.algorithm == ROUND_ROBIN and p.quantum_remaining <= 0:
            p.state = READY
            sys.running_pid = -1
            agregar log "quantum agotado"

    // Actualizar tiempos de espera
    for i from 0 to sys.process_count - 1:
        if sys.processes[i].state in {READY, WAITING}:
            sys.processes[i].waiting_time++

    // Seleccionar siguiente proceso si CPU está libre
    if sys.running_pid == -1:
        select_next_process()

    mostrar estado actual
```

Complejidad: O(n)

Módulo: Comunicación

void send_message(int sender_pid, int receiver_pid, const char *content)

Propósito: Envía un mensaje de un proceso a otro.

Algoritmo:

```
function send_message(sender_pid, receiver_pid, content):
  if sys.message_count >= MAX_MESSAGES:
    return error

  msg = &sys.messages[sys.message_count++]
  msg.sender_pid = sender_pid
  msg.receiver_pid = receiver_pid
  copiar content a msg.content (máx 256 chars)
  msg.delivered = false

  agregar log
```

Complejidad: O(1)

Características:

- No verifica que sender/receiver existan (simplificación)
- Mensajes persisten hasta ser leídos
- Sin timeout

void receive_message(int receiver_pid)

Propósito: Recupera todos los mensajes pendientes para un receptor.

Algoritmo:

```
function receive_message(receiver_pid):
  found = false

  for i from 0 to sys.message_count - 1:
```

```
    if sys.messages[i].receiver_pid == receiver_pid and
      not sys.messages[i].delivered:

        mostrar mensaje
        sys.messages[i].delivered = true
        found = true
        agregar log

  if not found:
    mostrar "no hay mensajes"
```

Complejidad: O(m) donde m = número de mensajes

Módulo: Sincronización

int create_semaphore(int initial_value)

Propósito: Crea un nuevo semáforo.

Retorno: ID del semáforo o -1 si hay error.

Algoritmo:

```
function create_semaphore(initial_value):
  if sys.semaphore_count >= MAX_SEMAPHORES:
    return -1

  id = sys.semaphore_count
  sem = &sys.semaphores[id]
  sem.id = id
  sem.value = initial_value
  sem.waiting_count = 0
```

```
  sys.semaphore_count++
  agregar log
  return id
```

Complejidad: O(1)

void wait_semaphore(int sem_id, int pid)

Propósito: Operación P (wait) sobre un semáforo.

Algoritmo:

```
function wait_semaphore(sem_id, pid):
  if sem_id inválido:
    return error

  sem = &sys.semaphores[sem_id]
  sem.value--

  agregar log "wait ejecutado"

  if sem.value < 0:
    // Bloquear proceso
    idx = find_process_by_pid(pid)
    if idx != -1:
      sys.processes[idx].state = WAITING
      sem.waiting_pids[sem.waiting_count++] = pid
      agregar log "proceso bloqueado"
```

Complejidad: O(n)

Semántica:

- Valor >= 0: Continúa ejecución
- Valor < 0: Proceso bloqueado, |valor| = procesos esperando

void signal_semaphore(int sem_id)

Propósito: Operación V (signal) sobre un semáforo.

Algoritmo:

```
function signal_semaphore(sem_id):
  if sem_id inválido:
    return error

  sem = &sys.semaphores[sem_id]
  sem.value++

  agregar log "signal ejecutado"

  if sem.value <= 0 and sem.waiting_count > 0:
    // Desbloquear primer proceso en cola
    pid = sem.waiting_pids[0]

    // Shift de la cola
    for i from 0 to sem.waiting_count - 2:
      sem.waiting_pids[i] = sem.waiting_pids[i + 1]
    sem.waiting_count--

    // Cambiar estado a READY
```

```
idx = find_process_by_pid(pid)
if idx != -1:
  sys.processes[idx].state = READY
  agregar log "proceso desbloqueado"
```

Complejidad: O(n)

Módulo: Estadísticas

void show_statistics()

Propósito: Calcula y muestra métricas de rendimiento.

Métricas Calculadas:

1. **Tiempo Promedio de Espera:**
2. avg_waiting = total_waiting_time / total_processes_completed
3. **Tiempo Promedio de Retorno:**
4. avg_turnaround = total_turnaround_time / total_processes_completed
5. **Utilización de CPU:**
6. cpu_utilization = (cpu_busy_time / current_time) × 100
7. **Throughput:**
8. throughput = total_processes_completed / current_time

Complejidad: O(1)

Algoritmos Implementados

1. FCFS (First Come First Served)

Descripción: Planificación no preventiva que ejecuta procesos en orden de llegada.

Características:

- Simple y predecible
- No hay cambios de contexto hasta que el proceso termina

- Puede sufrir de efecto convoy (procesos cortos esperan a largos)
- Sin starvation

Pseudocódigo:

Cola = []

Para cada proceso en orden de arrival_time:

  Agregar a Cola

Mientras Cola no vacía:

  P = Cola.pop_front()

  Ejecutar P hasta completar

Ventajas:

- Implementación sencilla
- Bajo overhead
- Fairness en orden temporal

Desventajas:

- Tiempo de espera puede ser alto
- No apropiado para sistemas interactivos

2. Round Robin

Descripción: Planificación preventiva con quantum de tiempo fijo.

Características:

- Cada proceso recibe una porción de tiempo (quantum)
- Si no termina, vuelve al final de la cola
- Apropiado para time-sharing
- Fairness entre procesos

Pseudocódigo:

```
Cola = cola_circular(procesos READY)

Mientras Cola no vacía:
  P = Cola.pop_front()
  Ejecutar P por quantum unidades o hasta terminar

  Si P no ha terminado:
    P.state = READY
    Cola.push_back(P)
```

Análisis de Quantum:

Quantum	**Efecto**
Muy pequeño	Muchos cambios de contexto, overhead alto
Muy grande	Se aproxima a FCFS, pierde interactividad
Óptimo	Balance entre responsividad y overhead

Regla General: Quantum ≈ 80% de los procesos deberían completarse en un quantum.

3. Prevención de Deadlock

Estrategia: Hold and Wait negado.

Implementación:

```
function puede_asignar(proceso, recursos_solicitados):
  return recursos_solicitados <= recursos_disponibles
```

Limitaciones:

- No detecta deadlocks ya existentes
- Puede causar starvation si un proceso requiere muchos recursos
- No es óptima (podría asignar en casos seguros que rechaza)

Alternativas Teóricas:

- Algoritmo del Banquero (más complejo, verifica seguridad)
- Detección y recuperación
- Prevención mediante ordenamiento de recursos

Gestión de Recursos

Modelo de Recursos

El sistema maneja dos tipos de recursos:

1. **CPU (Monoprocesador):**
    - Total: 1 unidad
    - Asignación: Exclusiva al proceso RUNNING
    - Liberación: Al terminar quantum o proceso
2. **Memoria (4 GB):**
    - Total: 4 bloques de 1024 MB cada uno
    - Asignación: En bloques completos
    - Liberación: Al terminar proceso

Políticas de Asignación

1. **Inmediata:** Si hay recursos, se asignan al solicitar
2. **Espera:** Si no hay recursos, proceso pasa a WAITING
3. **Prevención:** No se asigna si podría causar deadlock

Invariantes del Sistema

∀ tiempo t:

available.cpu + Σ(processes[i].allocated.cpu) = total.cpu

available.memory + Σ(processes[i].allocated.memory) = total.memory

Sincronización y Comunicación

Semáforos

Implementación

```
// Operación Wait (P)
void wait(Semaphore *s, int pid) {
  s->value--;
  if (s->value < 0) {
    block(pid);
    add_to_queue(s, pid);
  }
}

// Operación Signal (V)
void signal(Semaphore *s) {
  s->value++;
  if (s->value <= 0) {
    pid = remove_from_queue(s);
    wakeup(pid);
  }
}
```

Propiedades

1. **Atomicidad:** Las operaciones son indivisibles (simulada)
2. **FIFO:** Procesos se desbloquean en orden de bloqueo
3. **No hay busy waiting:** Procesos bloqueados no consumen CPU

Problema Productor-Consumidor

Configuración

```
Semaphore mutex = 1;        // Exclusión mutua
Semaphore empty = BUFFER_SIZE; // Espacios vacíos
Semaphore full = 0;         // Espacios llenos
Buffer buffer;
```

Algoritmo Productor

```
function producer():
  while true:
    item = produce_item()

    wait(empty)   // Esperar espacio
    wait(mutex)   // Entrar a región crítica

    buffer.add(item)

    signal(mutex)  // Salir de región crítica
    signal(full)   // Señalar nuevo item
```

Algoritmo Consumidor

```
function consumer():
  while true:
```

```
    wait(full)     // Esperar item
    wait(mutex)    // Entrar a región crítica

    item = buffer.remove()

    signal(mutex)  // Salir de región crítica
    signal(empty)  // Señalar espacio libre

    consume_item(item)
```

Análisis de Corrección

Invariantes:

1. $0 \leq$ buffer.count $\leq$ BUFFER_SIZE
2. mutex.value $\in \{0, 1\}$ (binario)
3. empty.value + full.value + buffer.count = BUFFER_SIZE

Propiedades de Seguridad:

- Exclusión mutua en acceso al buffer
- No overflow del buffer
- No underflow del buffer

Propiedades de Vivacidad:

- Sin deadlock (orden correcto de wait/signal)
- Progreso garantizado (si hay productores/consumidores activos)

Sistema de Logs

Estructura de Log

```
typedef struct {
```

```
    int time;        // Timestamp
    int pid;         // -1 para eventos del sistema
    char event[256];   // Descripción
} LogEntry;
```

Tipos de Eventos Registrados

Tipo	**Ejemplo**
Sistema	"Sistema inicializado"
Proceso	"Proceso PID 1 creado"
Recursos	"Recursos asignados a PID 1"
Planificación	"PID 1 ahora en ejecución"
Comunicación	"Mensaje enviado de PID 1 a PID 2"
Sincronización	"PID 1 bloqueado esperando semáforo 0"

Función de Log

```
void add_log(int pid, const char *event) {
    if (sys.log_count < MAX_LOG_ENTRIES) {
        sys.logs[sys.log_count].time = sys.current_time;
        sys.logs[sys.log_count].pid = pid;
        strncpy(sys.logs[sys.log_count].event, event, 255);
        sys.logs[sys.log_count].event[255] = '\0';
        sys.log_count++;
    }
}
```

Estrategia de Buffer Circular

Cuando se alcanza MAX_LOG_ENTRIES, el sistema deja de registrar. Una mejora sería implementar un buffer circular que sobrescriba entradas antiguas.

Métricas y Estadísticas

Fórmulas Implementadas

1. Tiempo de Espera (Waiting Time)

WT_i = Σ(tiempo en estado READY o WAITING)

Actualización:

```
// En cada execute_step()
if (process.state == READY || process.state == WAITING) {
   process.waiting_time++;
}
```

2. Tiempo de Retorno (Turnaround Time)

TAT_i = completion_time_i - arrival_time_i

TAT_i = WT_i + burst_time_i

Cálculo:

```
// Al terminar proceso
process.turnaround_time = process.completion_time - process.arrival_time;
```

3. Utilización de CPU

CPU_utilization = (tiempo_ocupado / tiempo_total) × 100

Seguimiento:

```
// En cada execute_step() cuando hay proceso ejecutando

sys.cpu_busy_time++;
```

4. **Throughput**

Throughput = procesos_completados / tiempo_total

Análisis de Complejidad

Operación	**Complejidad Temporal**	**Complejidad Espacial**
create_process	O(1)	O(1)
find_process	O(n)	O(1)
select_next_process	O(n)	O(1)
execute_step	O(n)	O(1)
request_resources	O(n)	O(1)
send_message	O(1)	O(1)
receive_message	O(m)	O(1)
wait_semaphore	O(n)	O(1)
signal_semaphore	O(n)	O(1)

Donde:

- n = número de procesos
- m = número de mensajes

Limitaciones y Restricciones

Límites del Sistema

Recurso	**Límite**	**Constante**
Procesos simultáneos	50	MAX_PROCESSES
CPU	1	MAX_CPU
Memoria	4 bloques (4096 MB)	MAX_MEMORY_BLOCKS
Mensajes	100	MAX_MESSAGES

Semáforos	10	MAX_SEMAPHORES
Buffer productor-consumidor	5	BUFFER_SIZE
Entradas de log	1000	MAX_LOG_ENTRIES

Restricciones de Diseño

1. **Monoprocesador:** Solo un proceso puede ejecutar a la vez
2. **Sin hilos:** Los procesos no tienen hilos internos
3. **Sin paginación:** Memoria en bloques completos
4. **Sin interrupciones:** Simulación cooperativa
5. **Sin prioridades dinámicas:** Prioridad fija al crear

Simplificaciones

1. **Tiempo discreto:** No hay fracciones de unidad de tiempo
2. **Operaciones instantáneas:** Cambios de contexto sin overhead
3. **Recursos simples:** Solo CPU y memoria
4. **Sin I/O:** No hay dispositivos de entrada/salida
5. **Sin sistema de archivos:** No hay persistencia de datos

Guía de Desarrollo

Compilación

Flags Recomendados

gcc -Wall -Wextra -std=c99 -g simulador_procesos.c -o simulador_procesos

Explicación:

- -Wall -Wextra: Habilita todas las advertencias
- -std=c99: Estándar C99

- -g: Información de depuración

Optimización

gcc -O2 -std=c99 simulador_procesos.c -o simulador_procesos

Niveles de optimización:

- -O0: Sin optimización (por defecto)
- -O1: Optimizaciones básicas
- -O2: Optimizaciones recomendadas
- -O3: Optimizaciones agresivas

Depuración

GDB (GNU Debugger)

Compilar con símbolos de depuración

gcc -g simulador_procesos.c -o simulador_procesos

Iniciar GDB

gdb ./simulador_procesos

Comandos útiles en GDB

(gdb) break create_process # Punto de interrupción

(gdb) run # Ejecutar

(gdb) print sys.process_count # Inspeccionar variable

(gdb) step # Paso a paso

(gdb) continue # Continuar

Valgrind (Detección de memoria)

```
valgrind --leak-check=full ./simulador_procesos
```

Extensiones Sugeridas

1. Algoritmo de Planificación por Prioridad

```
void select_next_process_priority() {
  int selected_idx = -1;
  int max_priority = -1;

  for (int i = 0; i < sys.process_count; i++) {
    if (sys.processes[i].state == READY &&
      sys.processes[i].priority > max_priority) {
      max_priority = sys.processes[i].priority;
      selected_idx = i;
    }
  }

  if (selected_idx != -1) {
    assign_process(selected_idx);
  }
}
```

2. Envejecimiento para Prevenir Starvation

```
void age_processes() {
  for (int i = 0; i < sys.process_count; i++) {
    if (sys.processes[i].state == READY) {
```

```
        sys.processes[i].waiting_time++;

        // Incrementar prioridad cada N unidades
        if (sys.processes[i].waiting_time % 10 == 0) {
          sys.processes[i].priority++;
        }
      }
    }
}
```

3. Múltiples CPUs

```
#define MAX_CPU 4

typedef struct {
  int cpu_id;
  int running_pid;
} CPU;

CPU cpus[MAX_CPU];

void select_next_processes() {
  for (int i = 0; i < MAX_CPU; i++) {
    if (cpus[i].running_pid == -1) {
      int pid = find_next_ready_process();
      if (pid != -1) {
        assign_to_cpu(pid, i);
```

```
        }
      }
    }
}
```

4. Sistema de Archivos Simulado

```
typedef struct {
    char name[64];
    int size;
    int owner_pid;
} File;

File files[MAX_FILES];

int create_file(const char *name, int pid);
int read_file(int file_id, int pid);
int write_file(int file_id, int pid, const char *data);
int delete_file(int file_id, int pid);
```

Estructura de Pruebas

Prueba Unitaria: Creación de Proceso

```
void test_create_process() {
    init_system();

    int pid = create_process(10, 5, 2);
    assert(pid == 1);
```

```
    assert(sys.process_count == 1);
    assert(sys.processes[0].state == READY);
    assert(sys.processes[0].burst_time == 10);
    assert(sys.processes[0].priority == 5);

    printf("✓ test_create_process pasó\n");
}
```

Prueba de Integración: FCFS

```
void test_fcfs_scheduling() {
    init_system();
    sys.algorithm = FCFS;

    create_process(5, 3, 1);  // PID 1
    create_process(3, 5, 1);  // PID 2
    create_process(7, 4, 1);  // PID 3

    // Simular hasta que todos terminen
    while (sys.total_processes_completed < 3) {
        execute_step();
    }

    // Verificar orden de ejecución en logs
    assert(/* verificaciones */);

    printf("✓ test_fcfs_scheduling pasó\n");
```

}

Mejores Prácticas

1. **Manejo de Errores:**
    - Siempre verificar límites de arrays
    - Validar PIDs antes de usar
    - Verificar estados de procesos
2. **Documentación:**
    - Comentar funciones complejas
    - Documentar invariantes
    - Explicar decisiones de diseño
3. **Modularidad:**
    - Funciones pequeñas y enfocadas
    - Separar lógica de presentación
    - Usar constantes en lugar de números mágicos
4. **Performance:**
    - Evitar búsquedas lineales innecesarias
    - Considerar usar hash tables para PIDs
    - Optimizar bucles críticos

Referencias

Libros

1. **Operating System Concepts** (Silberschatz, Galvin, Gagne)
    - Capítulos 3-6: Procesos, Hilos, Planificación, Sincronización
2. **Modern Operating Systems** (Andrew S. Tanenbaum)
    - Capítulos 2-3: Procesos e Hilos, Deadlocks

Estándares

- **C99 Standard (ISO/IEC 9899:1999)**
- **POSIX.1-2017** (para conceptos de procesos)

Herramientas

- **GCC:** https://gcc.gnu.org/
- **GDB:** https://www.gnu.org/software/gdb/
- **Valgrind:** https://valgrind.org/
- **Make:** https://www.gnu.org/software/make/

Apéndices

Apéndice A: Códigos de Error

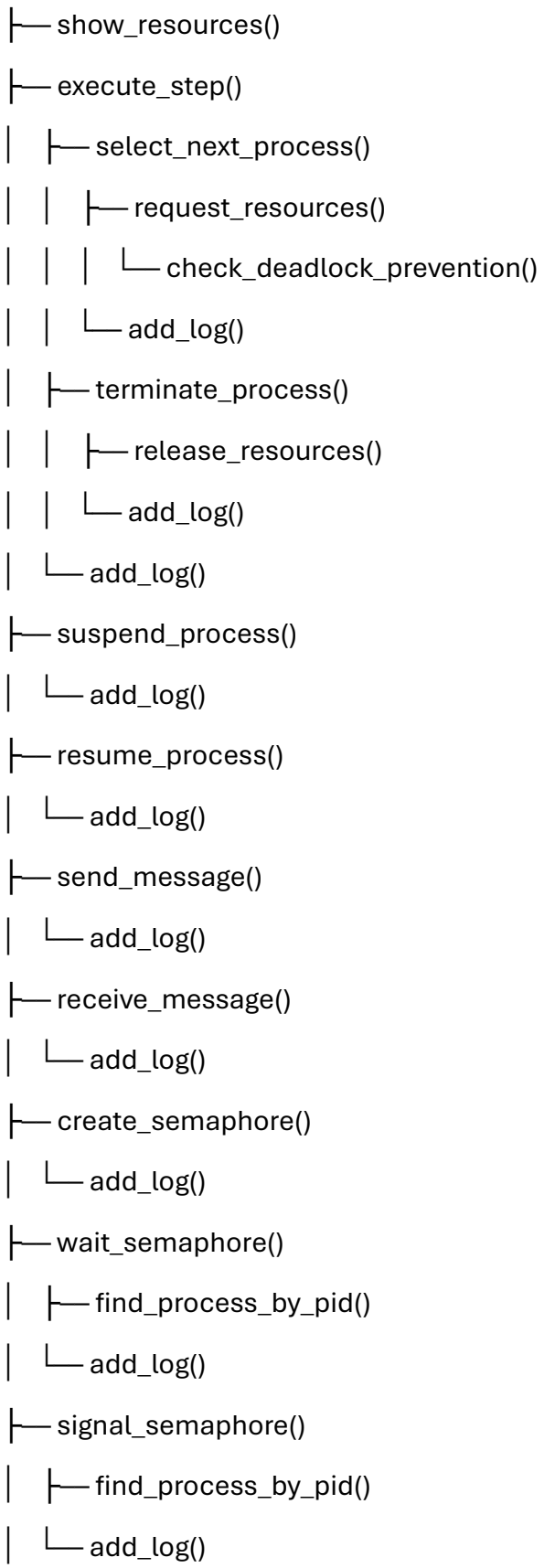
Código	Descripción
-1	Operación fallida (genérico)
-2	Proceso no encontrado
-3	Recursos insuficientes
-4	Límite alcanzado

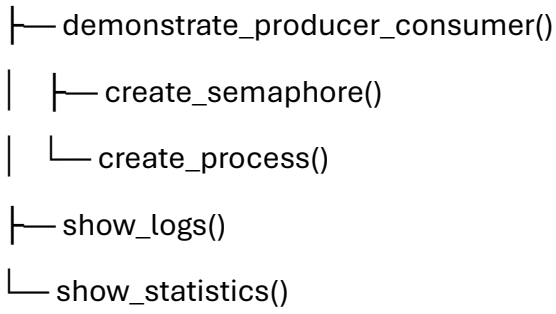
Apéndice B: Constantes del Sistema

```
#define MAX_PROCESSES 50
#define MAX_CPU 1
#define MAX_MEMORY_BLOCKS 4
#define MEMORY_BLOCK_SIZE 1024  // MB
#define MAX_MESSAGES 100
#define MAX_SEMAPHORES 10
#define BUFFER_SIZE 5
#define MAX_LOG_ENTRIES 1000
```

Apéndice C: Diagrama de Dependencias

```
main()
  ├── init_system()
  ├── select_algorithm()
  └── [bucle de menú]
      ├── create_process()
      │   └── add_log()
      ├── list_processes()
```


show_resources()
execute_step()
select_next_process()
request_resources()
check_deadlock_prevention()
add_log()
terminate_process()
release_resources()
add_log()
add_log()
suspend_process()
add_log()
resume_process()
add_log()
send_message()
add_log()
receive_message()
add_log()
create_semaphore()
add_log()
wait_semaphore()
find_process_by_pid()
add_log()
signal_semaphore()
find_process_by_pid()
add_log()


├── demonstrate_producer_consumer()
│ ├── create_semaphore()
│ └── create_process()
├── show_logs()
└── show_statistics()