6장. 동기화 도구들 (2)

운영체제

목차

학습목표

- 임계구역 문제를 설명하고, 경쟁조건을 설명
- 메모리장벽, compare-and-swap 연산자 및 원자적 변수를 사용하여 임계구역 문제에 대한 하드웨어 해결책 설명
- Mutex 락, 세마포, 모니터 및 조건 변수를 사용하여 임계구역 문제를 해결하는 방법
- 적은, 중간, 심한 경쟁 시나리오에서 임계구역 문제를 해결하는 도구를 평가

6.5 MUTEX LOCKS

6. Synchronization Tools

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock

Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

            critical section

        release lock

remainder section
}
```

- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.

Mutex Locks (cont.)

- Definition of the acquire() operation

```
acquire() {
    while (!available)
       ; // busy wait
    available = false;
}
```

- Definition of the release() operation

```
release() {
    available = true;
}
```

- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

6.6 SEMAPHORES

6. Synchronization Tools

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - wait() and signal()
    - Originally called P() and V()
    - 네델란드 Edger Dijkstra

Semaphore (cont.)

- Definition of the wait() operation

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the signal() operation

```
signal(S) {
    S++;
}
```

6.6.1 Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphore Usage (Cont.)

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

Semaphore Usage (Cont.)

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

6.6.2 Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

Waiting queue

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

6.7 MONITORS

6. Synchronization Tools

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal(mutex) .... wait(mutex)
  - wait(mutex) ... wait(mutex)
  - Omitting of wait (mutex) and/or signal (mutex)

- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

6.7.1 Usage

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

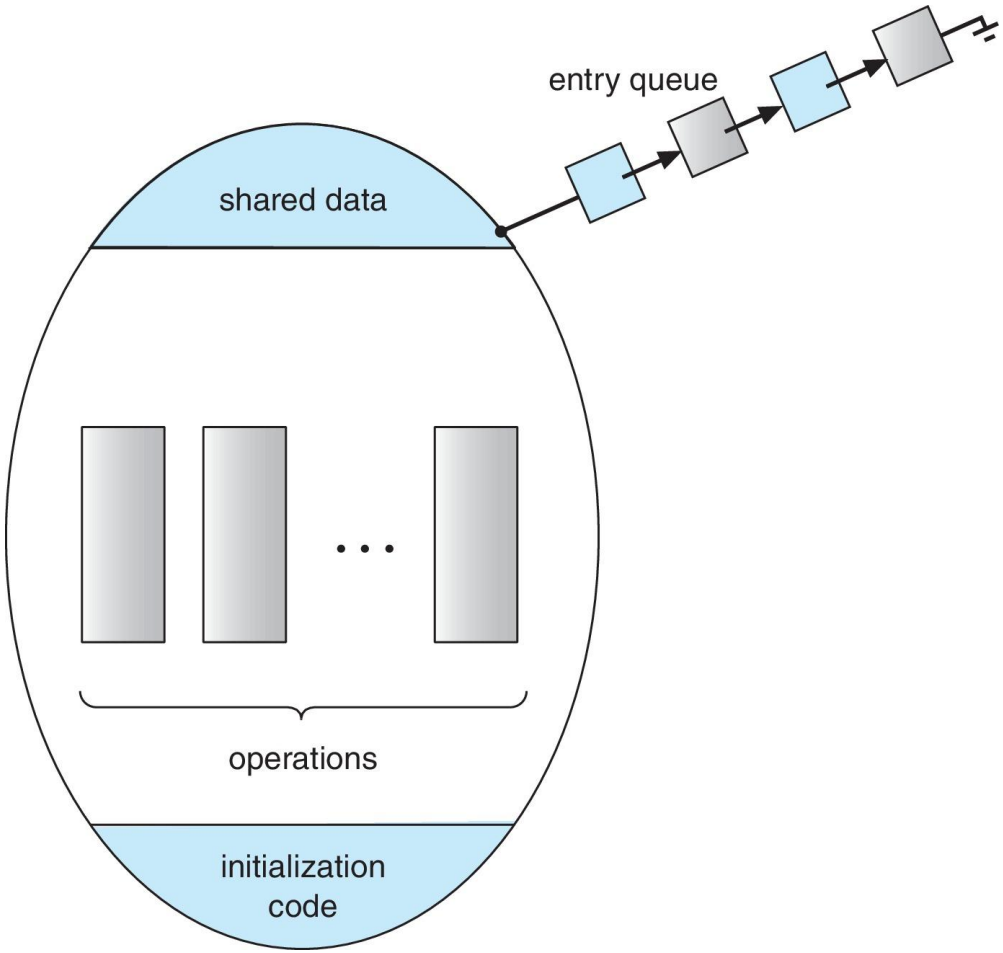
```
monitor monitor-name
{
   // shared variable declarations
   function P1 (…) { …. }

   function P2 (…) { …. }

   function Pn (…) {……}

  initialization code (…) { … }
}
```

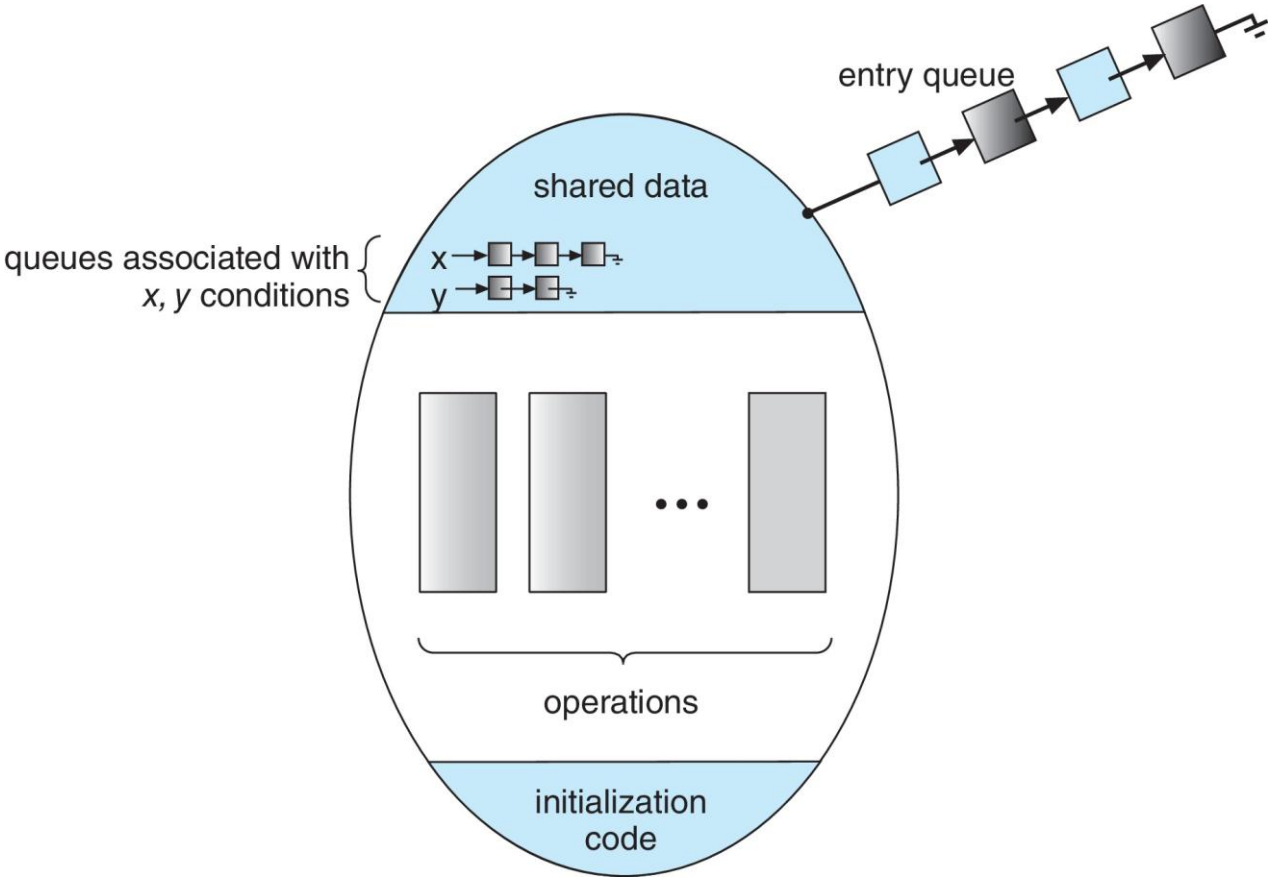
Schematic view of a Monitor



Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no x.wait() on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

6.7.2 Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting
                    inside the monitor
```

- Each function F will be replaced by

```
wait(mutex);
  …
body of F;
  …
if (next_count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementation – Condition Variables

- For each condition variable x, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation x.wait() can be implemented as:

```
x_count++;
if (next_count > 0)
        signal(next);
else
      signal(mutex);
wait(x_sem);
x_count--;
```

Implementation (Cont.)

- The operation x.signal() can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

6.7.3 Resuming Processes within a Monitor

- If several processes queued on condition variable **`x`**, and **`x.signal()`** is executed, which process should be resumed?
- FCFS frequently not adequate
- conditional-wait construct of the form **`x.wait(c)`**
  - Where c is priority number
  - Process with lowest number (highest priority) is scheduled next

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
    ...
access the resource;
    ...
R.release;
```

- Where R is an instance of type ResourceAllocator

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
        boolean busy;
        condition x;

        void acquire(int time) {
                if (busy)
                        x.wait(time);
                busy = true;
        }
        void release() {
                busy = FALSE;
                x.signal();
        }
        initialization code() {
         busy = false;
        }
}
```

Single Resource Monitor (Cont.)

- Usage:

```
acquire
  ...
release
```

- Incorrect use of monitor operations
  - **`release()`** … **`acquire()`**
  - **`acquire()`** … **`acquire()`**
  - Omitting of **`acquire()`** and/or **`release()`**

6.8 LIVENESS

6. Synchronization Tools

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- Liveness refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

6.8.1 Deadlocks

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
   ...                 ...
signal(S);          signal(Q);
signal(Q);          signal(S);
```

- Consider if P0 executes wait(S) and P1 wait(Q).
  - When P0 executes wait(Q), it must wait until P1 executes signal(Q)
  - However, P1 is waiting until P0 execute signal(S).
  - Since these signal() operations will never be executed, P0 and P1 are deadlocked.

6.8.2 Priority Inversion

- Other forms of deadlock:
- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via priority-inheritance protocol

Priority Inheritance Protocol

- Consider the scenario with three processes **P1, P2**, and **P3**.
  - **P1** has the highest priority, **P2** the next highest, and **P3** the lowest.
- Assume a resource **P3** is assigned a resource **R** that **P1** wants.
  - Thus, **P1** must wait for **P3** to finish using the resource.
  - However, **P2** becomes runnable and preempts **P3**.
  - What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
- To prevent this from occurring, a **priority inheritance protocol** is used.
  - simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource.

6.9 EVALUATION

6. Synchronization Tools

특정 동기화 도구 선택

- 최근 동기화 도구의 성능에 대한 관심
- 특정 동기화 도구를 사용할 시기를 결정하는 간단한 전략 제시
- 하드웨어 솔루션
  - 매우 낮은 수준
    - mutex 락과 같은 동기화 도구를 구성하기 위한 기초로 사용
  - 최근 lock-free 알고리즘 구현을 위해 CAS 명령 사용에 중점
    - 개발 및 테스트가 어려움
  - CAS 기반 접근 방식
    - 낙관적 접근법
    - 갱신 후 충돌 감지 사용
  - 상호배제 락킹
    - 비관적 접근법
    - 락 획득 후 갱신

특정 동기화 도구 선택 (계속)

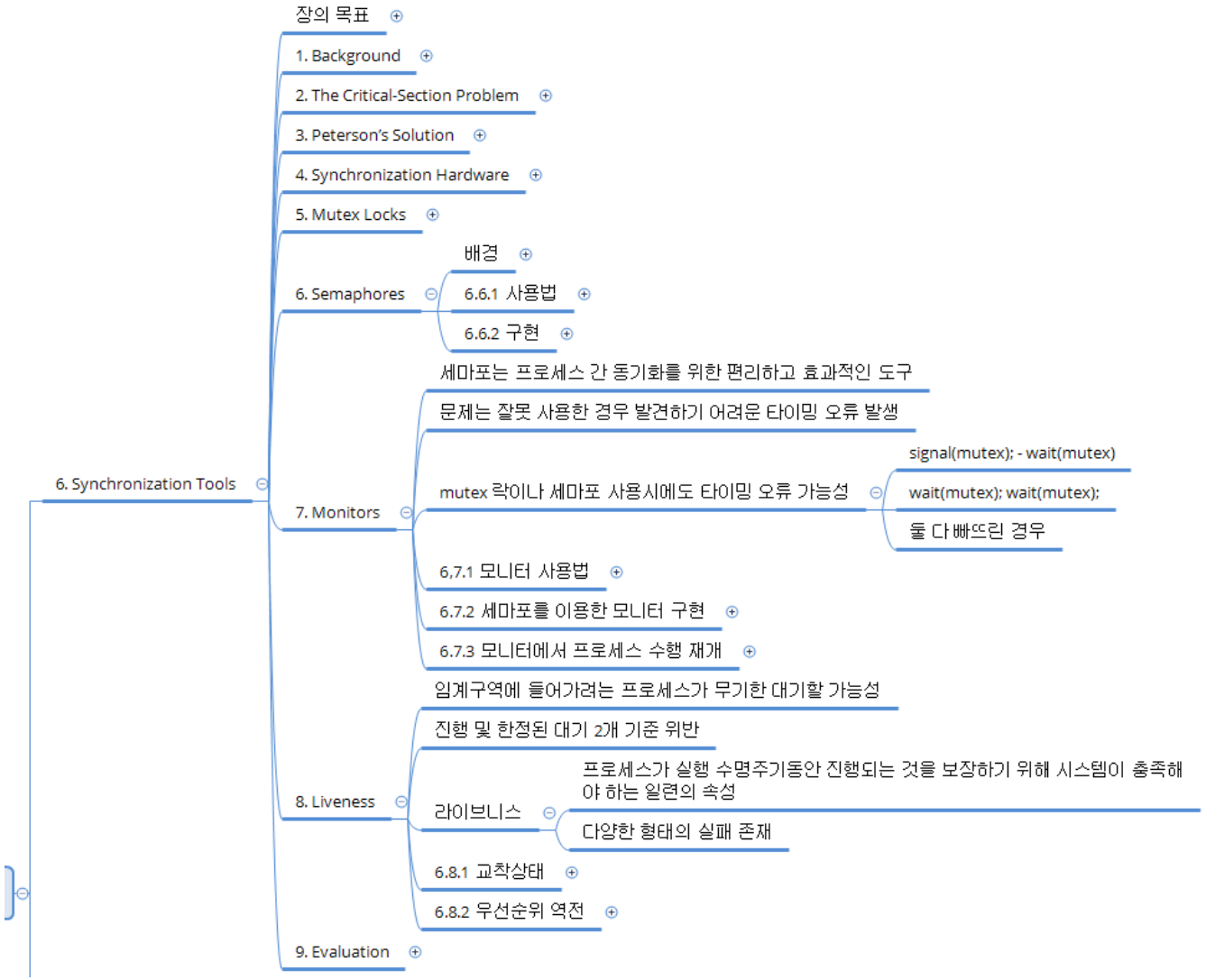
- 경합 부하에 따른 CAS 동기화와 기존 동기화 비교
  - 경합 없음
    - 둘 다 빠르지만, CAS가 다소 빠름
  - 적당한 경합
    - CAS가 훨씬 빠름
  - 심한 경합
    - 기존 동기화가 빠름
- 기법의 선택은 시스템 성능에 큰 영향
  - 원자적 정수
    - 기존 락보다 가벼워서 공유 변수에 대한 단일 업데이트에 적합
  - mutex 락과 세마포
    - mutex 락이 더 가벼움
  - 한정된 수의 자원에 접근
    - counting semaphore
  - reader-writer lock이 mutex lock보다 선호

특정 동기화 도구 선택 (계속)

- 모니터와 조건변수의 사용
  - 단순성, 사용편의성 때문에 선호
  - 오버헤드가 큼
  - 확장성이 좋지 않은 경우도 있음

- 새로운 연구
  - 더 효율적인 코드를 생성하는 컴파일러 설계
  - 병행 프로그래밍 지원 언어 개발
  - 기존 라이브러리와 API 성능 향상

학습한 내용


6. Synchronization Tools
장의 목표
1. Background
2. The Critical-Section Problem
3. Peterson's Solution
4. Synchronization Hardware
5. Mutex Locks
6. Semaphores
배경
6.6.1 사용법
6.6.2 구현
7. Monitors
세마포는 프로세스 간 동기화를 위한 편리하고 효과적인 도구
문제는 잘못 사용한 경우 발견하기 어려운 타이밍 오류 발생
mutex 락이나 세마포 사용시에도 타이밍 오류 가능성
signal(mutex); - wait(mutex)
wait(mutex); wait(mutex);
둘 다 빠뜨린 경우
6.7.1 모니터 사용법
6.7.2 세마포를 이용한 모니터 구현
6.7.3 모니터에서 프로세스 수행 재개
8. Liveness
임계구역에 들어가려는 프로세스가 무기한 대기할 가능성
진행 및 한정된 대기 2개 기준 위반
라이브니스
프로세스가 실행 수명주기동안 진행되는 것을 보장하기 위해 시스템이 충족해야 하는 일련의 속성
다양한 형태의 실패 존재
6.8.1 교착상태
6.8.2 우선순위 역전
9. Evaluation