IN1011

Operating Systems

Lecture 05

(part 1): Mutual Exclusion

(part 2): Semaphores

IN1011

Operating Systems

Lecture 05 (part 1): Mutual Exclusion, Dekker's Algorithm, Peterson's Algorithm

Challenges with Concurrent Execution

- The following challenges arise with concurrently executing processes:
  - **Race condition** – where the order in which processes access and alter a shared resource could significantly affect the remainder, and outcome, of their execution.
  - In such situations there may be a need for
    - **mutual exclusion** (i.e. the processes/threads are made to access the resource one-at-a-time)
    - enforcing a **precedence** (i.e. processes/threads must execute parts of their code in a certain order);
  - **Resource starvation** – a process waits an unacceptably long time to gain access to a shared resource, due to other processes having access to the resource;
  - **Deadlock** – processes wait indefinitely for each other, unable to proceed with their respective executions;

Awareness Between Processes

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Table 5.2
Process Interaction

Mutual Exclusion

```
          PROCESS 1 */                    /* PROCESS 2 */                    /* PROCESS n */

void P1                             void P2                             void Pn
{                                   {                                   {
   while (true) {                      while (true) {                      while (true) {
      /* preceding code */;               /* preceding code */;               /* preceding code */;
      entercritical (Ra);                 entercritical (Ra);                 entercritical (Ra);
      /* critical section */;     ...     /* critical section */;     ...     /* critical section */;
      exitcritical (Ra);                  exitcritical (Ra);                  exitcritical (Ra);
      /* following code */;               /* following code */;               /* following code */;
   }                                   }                                   }
}                                   }                                   }
```

Figure 5.4 Illustration of Mutual Exclusion

This lecture contains a number of “psuedocode” examples like these (i.e. the code will not compile). This code looks like it has been written in the C/C++ family of languages.

Mutual Exclusion: Software Approaches

- In 1965, the famous computer scientist **Edsger Dijkstra** reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician **Theodorus Dekker**
- By developing Dekker's algorithm in stages, we can illustrate a number of the bugs encountered in developing concurrent programs
- In 1981, **Gary Peterson** formulated an alternative mutual exclusion solution similar to Dekker's

Dekker's Algorithm

/* PROCESS 0 /*	/* PROCESS 1 */

•
•
while (turn != 0)
   /* do nothing */ ;
/* critical section*/;
turn = 1;
•
	•
	•
	while (turn != 1)
	   /* do nothing */;
	/* critical section*/;
	turn = 0;
	•

(a) First attempt

Dekker's Algorithm

/* PROCESS 0 */	/* PROCESS 1 */

•
•
while (flag[1])
   /* do nothing */;
flag[0] = true;
/*critical section*/;
flag[0] = false;
•
	•
	•
	while (flag[0])
	   /* do nothing */;
	flag[1] = true;
	/* critical section*/;
	flag[1] = false;
	•

(b) Second attempt

Dekker's Algorithm

/* PROCESS 0 */	/* PROCESS 1 */

•
•
`flag[0] = true;`
`while` `(flag[1])`
   `/* do nothing */;`
`/* critical section*/;`
`flag[0] = false;`
•
	•
	•
	`flag[1] = true;`
	`while` `(flag[0])`
	   `/* do nothing */;`
	`/* critical section*/;`
	`flag[1] = false;`
	•

(c) Third attempt

Dekker's Algorithm

/* PROCESS 0 */	/* PROCESS 1 */
•	•
•	•
flag[0] = true;	flag[1] = true;
while (flag[1]) {	**while** (flag[0]) {
flag[0] = false;	flag[1] = false;
/*delay */;	/*delay */;
flag[0] = true;	flag[1] = true;
}	}
/*critical section*/;	/* critical section*/;
flag[0] = false;	flag[1] = false;
•	•

(d) Fourth attempt

Dekker's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          while (flag [1]) {
               if (turn == 1) {
                    flag [0] = false;
                    while (turn == 1) /* do nothing
*/;
                    flag [0] = true;
               }
          }
          /* critical section  */;
          turn = 1;
          flag [0] = false;
          /* remainder   */;
     }
}
void P1( )
{
     while (true) {
          flag [1] = true;
          while (flag [0]) {
               if (turn == 0) {
                    flag [1] = false;
                    while (turn == 0) /* do nothing
*/;
                    flag [1] = true;
               }
          }
          /* critical section   */;
          turn = 0;
          flag [1] = false;
          /* remainder   */;
     }
}
void main ()
{
     flag [0] = false;
     flag [1] = false;
     turn = 1;
     parbegin (P0, P1);
}
```

Peterson's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1) /* do nothing */;
          /* critical section */;
          flag [0] = false;
          /* remainder */;
     }
}
void P1()
{
     while (true) {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0) /* do nothing */;
          /* critical section */;
          flag [1] = false;
          /* remainder */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

Mutual Exclusion Requirements

- Only one process at a time is allowed access to a critical section;
- A process that halts in its non-critical section must not prevent other processes from accessing their critical sections;
- A process that requires access to its critical section must not be delayed indefinitely –- so no <u>deadlock</u> or <u>starvation</u> is allowed;
- When no process is requesting access to a critical section, **any** subsequent process that requests access should be granted access without delay;
- No assumptions about relative process speeds, or number of processes, are made;
- A process remains inside it’s critical section only for a finite amount of time.

IN1011

Operating Systems

Lecture 05 (part 2): Semaphores

Some Problems Solved with Semaphores

- Mutual Exclusion
- Producer/Consumer – infinite and finite buffer versions
- Reader/Writer

Semaphore

- A **Semaphore** is an integer value used for signaling among processes, together with 3 atomic operations: **initialise**, **decrement** and **increment**.
- These operations are the only way to inspect or manipulate semaphores
- Semaphores allow two or more processes to cooperate by means of simple signals. This is done by calling **semSignal( )** or **semWait( )** methods, related to the increment and decrement semaphore operations:
  - Semaphore **"s"** is initialised to a nonnegative integer value;
  - When called, the **semWait(s)** operation decrements **"s"** by 1. If **"s"** becomes negative, the process executing semWait is blocked. Otherwise, the process continues execution;
  - When called, the **semSignal(s)** operation increments **"s"** by 1. If the resulting value of **"s"** is less than or equal to zero, then a process blocked by a semWait operation is unblocked.

Semaphore Primitive

semWait is how a process requests access to a shared resource. If access is unavailable, the variable s is decremented (to indicate a process has made a request) and the process joins a “blocked” queue to wait for when the resource becomes free.

semSignal is how a process announces to other waiting processes that it is releasing a shared resource (s is incremented). Upon doing this, another process leaves the blocked queue and enters the ready queue (in anticipation of gaining access to the shared resource).

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Figure 5.6 A Definition of Semaphore Primitives

semWait and semSignal are assumed atomic – i.e. cannot be interrupted and must execute to completion

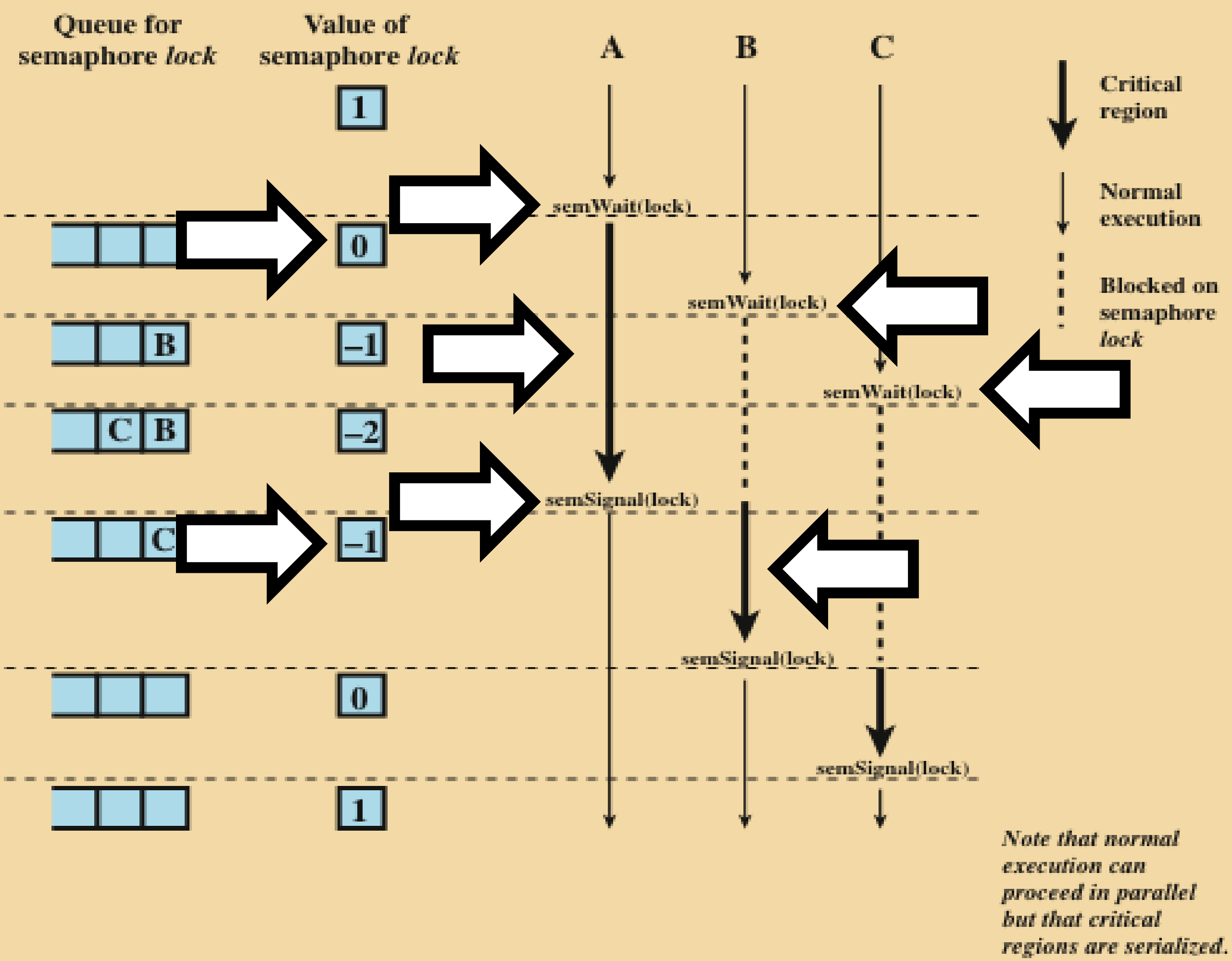
Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.9 Mutual Exclusion Using Semaphores

In each process, a semWait(s) is executed just before its critical section. If s becomes negative, the process is blocked. If s is 1, then its decremented to 0 and the process immediately enters its critical section. Because s is no longer positive, no other process will be able to enter their critical sections.

Mutual Exclusion Using Semaphores (contd.)



Producer/Consumer Problem

General Statement:	One or more producers are generating data and placing these in a buffer
	A single consumer is taking items out of the buffer one at a time
	Only one producer or consumer may access the buffer at any one time

The challenge: ensure that the producer won't try to add data into the buffer if the buffer is full. And, that the consumer won't try to read from an empty buffer

Producer/Consumer : Infinite Buffer

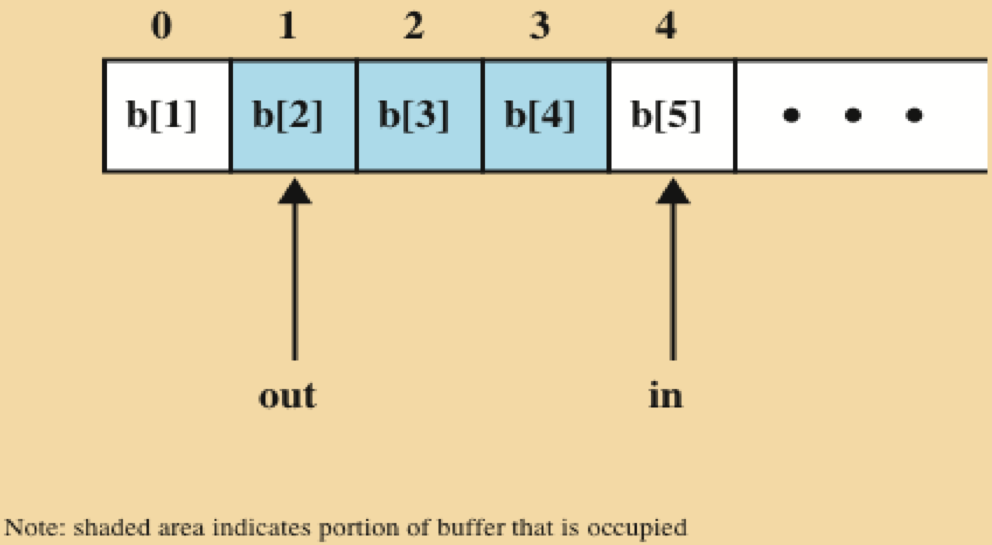
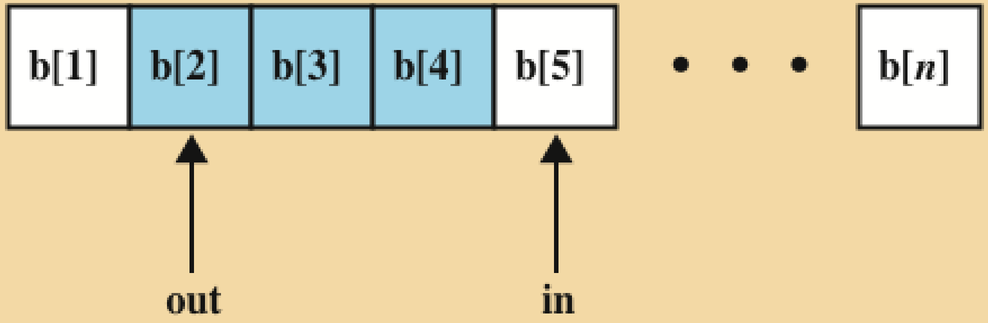
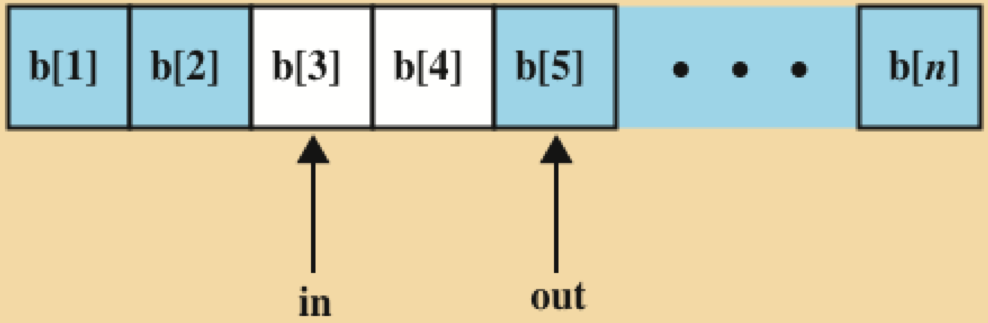


Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Producer/Consumer : Finite Circular Buffer





```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```