

Lecture 9

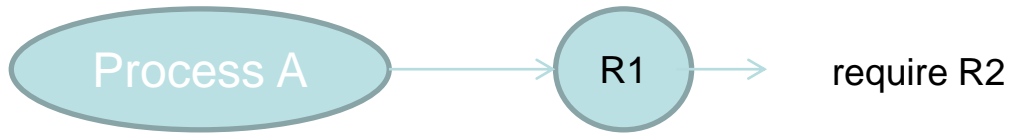
Processes concurrency - mutual
exclusion

Race condition

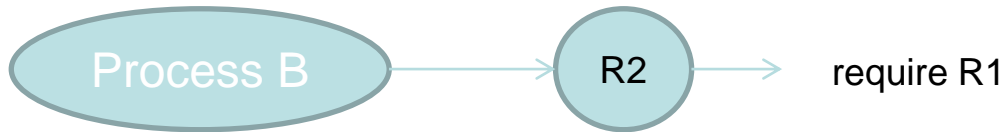
- In concurrent/parallel systems, a *race condition* can occur when multiple processes or threads read and write data so that the final result depends on the order of execution of instructions in the multiple processes/threads.
- Example: processes P1 and P2 share global variables a and b with initial values $a = 2$ and $b = 4$.
 - P1 has the instruction $a = a + b$
 - P2 has the instruction $b = a + b$
- Due to concurrency, the final values of a and b depend on the order of execution of the two instructions:
 $a = 6, b = 10$, or $b = 6$ and $a = 8$

Process interaction

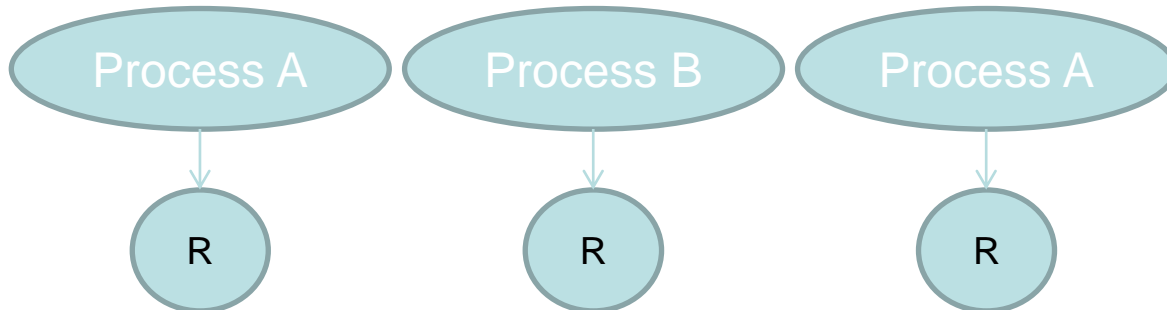
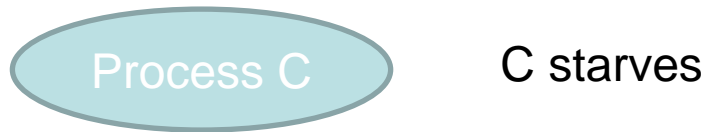
- In a computer system,
 1. processes run independently of each other, but they compete for resources;
 2. processes share resources, e.g., L2 cache, queues;
 3. processes cooperate, one process' results depending of other processes' execution data.
- *Mutual exclusion*: two or more processes compete for the same resource (i.e. printer); one process gets the control, the other(s) wait its(their) turn. The resource is called *critical resource*, and the section of the program that uses it is a *critical section*.
- When two processes block each other's access to a resource each already acquired, they are *deadlocked*.
- When three or more processes compete for a resource, it is possible to have the control granted only to some of them (i.e. they are high priority), and one or several processes can't get the CPU – they experience *starvation*.



Deadlock!



-----> time



Rules for mutual exclusion

1. Mutual exclusion should be applied: only one process is allowed to enter its critical section for one (shared) resource at a time.
 2. When no process is in a critical section, any process that requests entry to its critical section must be permitted without delay.
 3. A process that halts outside its critical section must do so without interfering with other processes.
 4. It must not be possible for a process requiring access to a critical section to be delayed indefinitely – no deadlock or starvation.
 5. A process stays inside its critical section for a finite time only.
- No assumptions are made about relative process speeds or number of cores.

Semaphores

- *Semaphore*: a variable that has an integer value upon which three operations can be executed:
 - initialised to a nonnegative integer value.
 - *semWait* operation decrements the semaphore value. If the value becomes negative, the process executing *semWait* is **blocked**. Otherwise, the process continues execution.
 - *semSignal* increments the semaphore value. If the resulting value is ≤ 0 , then a process blocked by a *semWait* operation, if any, is **unblocked**.
- If the semaphore value is positive, that value equals the number of processes that can issue a wait and immediately continue to execute.
- If the value is 0, the next process to call wait is blocked and the semaphore value goes negative.

Comments

- In general, there is no way to know before a process decrements a semaphore whether it will block or not.
- After a process increments a semaphore and another process gets woken up, both processes proceed running concurrently.
- When a semaphore is signalled, there might be a process (or more) waiting or none. The number of unblocked processes can be one or zero.
- The semWait and semSignal primitives are *atomic* – they can't be interrupted.

```

struct semaphore {
    int count;
    queueType queue;
};

void semWait (semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* the process is placed in s.queue */;
        /* block this process */;
    }
}

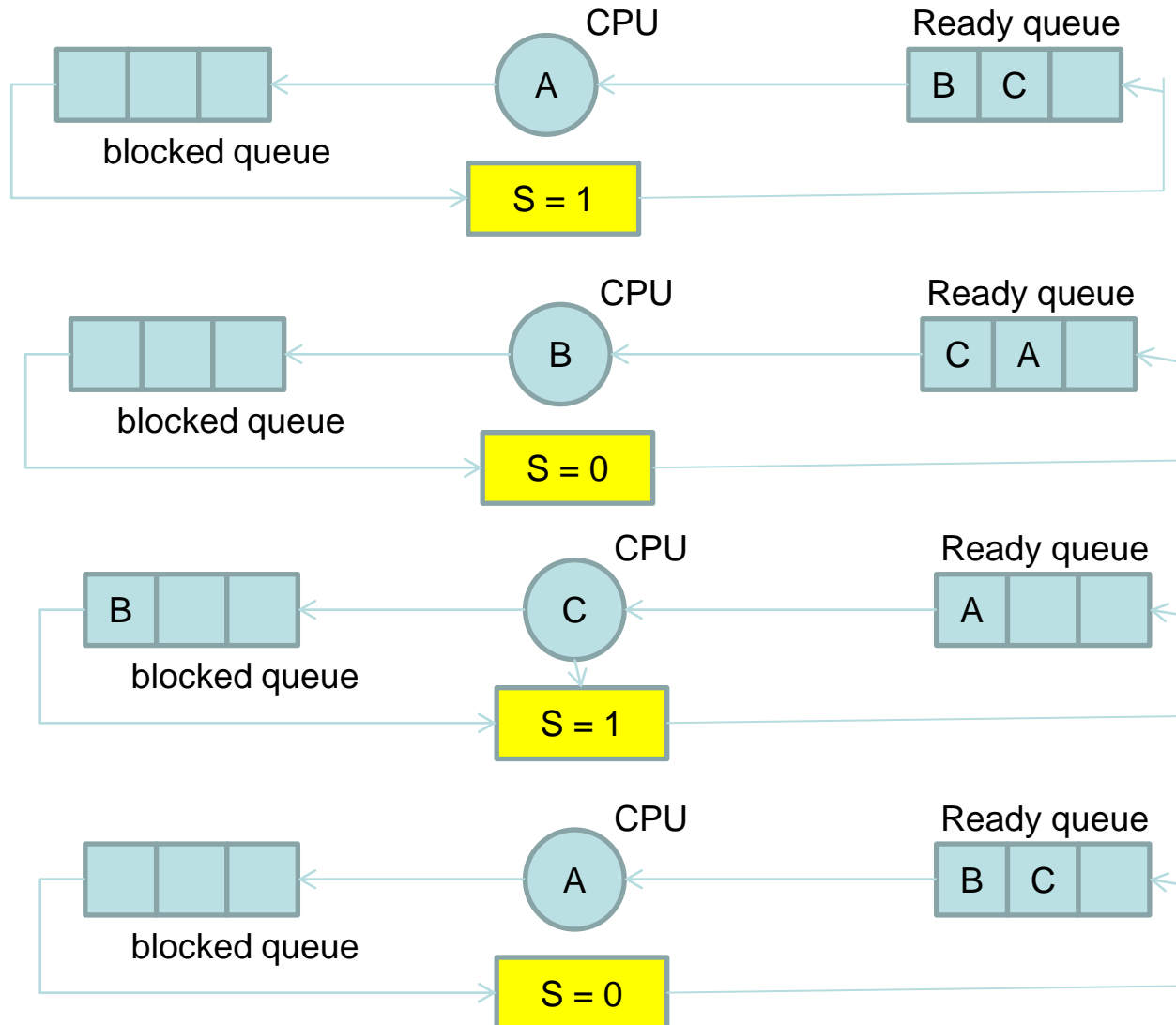
void semSignal (semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process from s.queue */;
        /* place the process in the ready queue */;
    }
}

```


Binary semaphore

- The binary semaphore may only take the values of zero and one.
- Operations:
 - Initialisation: 0 or 1;
 - `semWaitB` checks the semaphore value. If the value is 0, the process executing `semWaitB` is blocked, else if value is 1, decrement it and process continues execution.
 - `semSignalB` checks to see if there is any process blocked. If yes, a process is unblocked, else the value of the semaphore is set to 1.
- Mutex: a binary semaphore for which the same process locks and unlocks the semaphore.
- The queue associated with the semaphore is usually *FIFO - strong semaphore*.

- Example: processes A and B depend on results produced by process C. S is a binary semaphore.

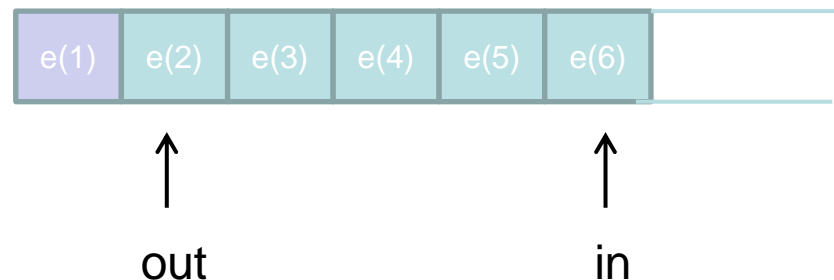


Mutual exclusion with semaphore

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

Application: Producer/Consumer

- One or more producers place data they create in a buffer. One consumer takes items out of the buffer, one at a time.
- Only one producer or the consumer can access the buffer at a time.
- The problem is to make sure that producers don't try to add data into a full buffer, and that the consumer doesn't try to take data from an empty buffer.
- We assume the buffer is infinite and consists of a linear array of elements.



Producer:

```
while (true) {  
    /* produce data d */;  
    e[in] = d;  
    in++;  
}
```

Consumer:

```
while (true) {  
    while (in <= out)  
        /* do nothing */  
    t = d[out];  
    out++;  
    /* consume t */;  
}
```

```
/* program producer/consumer */  
int n;  
binary_sem s = 1, delay = 0;  
  
void producer()  
{  
    while (true) {  
        produce();  
        semWaitB(s);  
        append();  
        n++;  
        if (n == 1) semSignalB(delay);  
        semSignalB(s);  
    }  
}
```

```

void consumer()
{
    int m;    /*local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m ==0) semWaitB(delay);

    }
}

void main()
{
    n = 0;
    parbegin (producer, consumer);
}

```

Questions and problems

- When does a race condition occur?
- List the requirements for mutual exclusion.
- What operations can be performed on a semaphore.
- What is a binary semaphore?
- Adapt the producer/consumer application for a finite buffer.
- What changes are required if there are two or more producers and two or more consumers?

References

- A. S. Tanenbaum and H. Bos: Modern Operating Systems, Pearson, 4th edition, 2014.
- W. Stallings: Operating Systems. Internals and design principles. Pearson, 2012