NTNU | Norwegian University of Science and Technology

Operating Systems

Theoretical Exercise 3: Solutions and Discussion

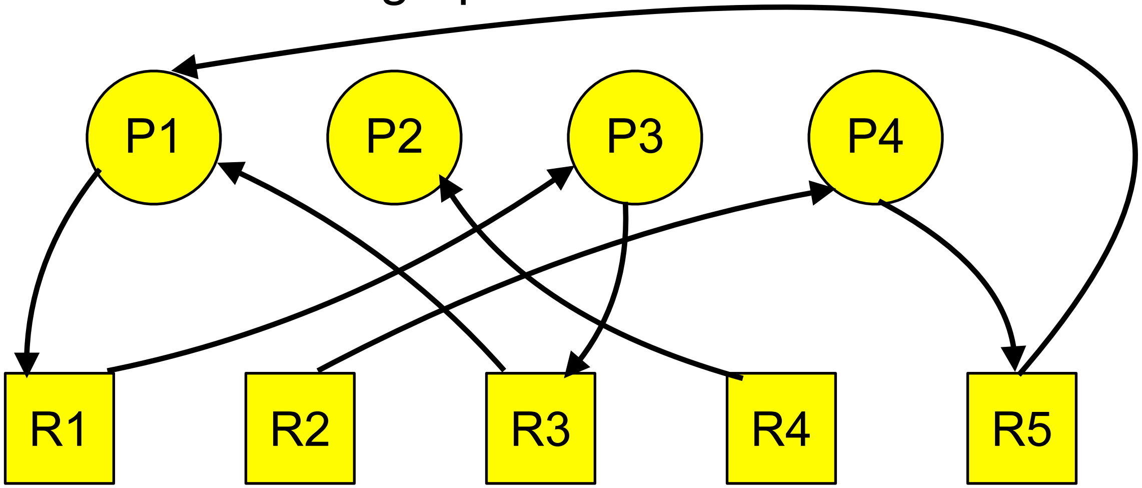
Michael Engel

3.1 Resource allocation graphs

Consider a system with four processes P1...P4 which want to access five exclusive, non preemptible resources R1...R5.

The atomic requests for the resources are arriving in the following order: P1 → R3, P3 → R1, P4 → R2, P1 → R5, P3 → R3, P4 → R5, P2 → R4 and finally P1 → R1.

a. Draw the resource allocation graph

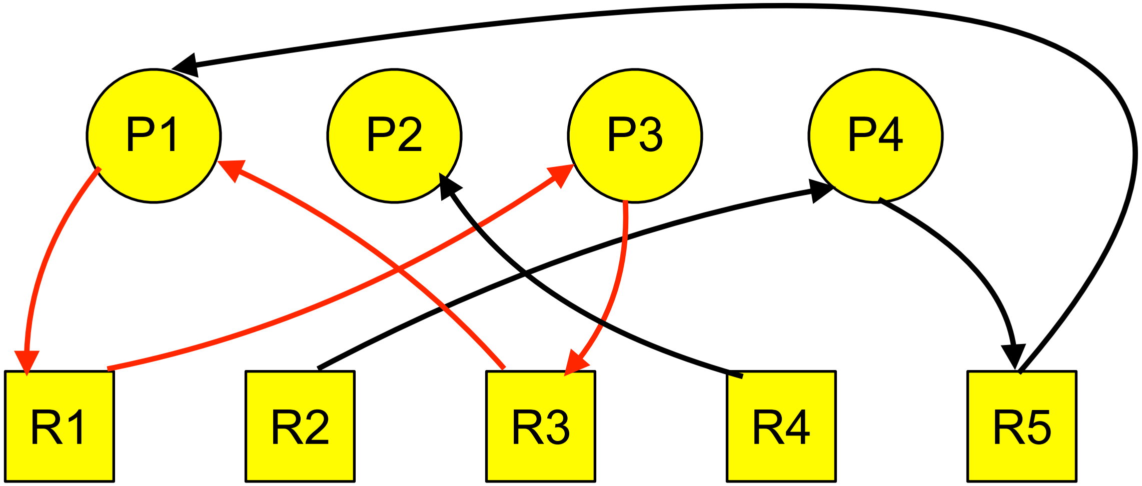


3.1 Resource allocation graphs

The atomic requests for the resources are arriving in the following order: P1 → R3, P3 → R1, P4 → R2, P1 → R5, P3 → R3, P4 → R5, P2 → R4 and finally P1 → R1.

b. Which condition has to be fulfilled for a deadlock to occur?

A circle in the resource allocation graph



3.1 Resource allocation graphs

The atomic requests for the resources are arriving in the following order: P1 → R3, P3 → R1, P4 → R2, P1 → R5, P3 → R3, P4 → R5, P2 → R4 and finally P1 → R1.

c. Is there a deadlock present in the system described above?

Yes, since there is a circle in the resource allocation graph:

P1 → R1 → P3 → R3 → P1

P1 waits for R1, which is already allocated to P3
while
P3 waits for R3, which is already allocated to P1

3.2 Resource allocation graphs

Three programs **Pa**, **Pb** and **Pc** with functions printing their own name:

- **Pa**: a1(), a2() and a3() / **Pb**: b1() and b2() / **Pc**: c1(), c2() and c3()

```
void a1() {
  <possibly block here using wait()>
  printf("a1 ");
  <possibly signal here using signal()>
}
```

```
int main() {
  a1(); a2(); a3();
}
```

Three semaphores *Sa*, *Sb* and *Sc*

Desired output: a1 b1 a2 c1 c2 b2 a3 c3

a. To which initial values do you have to set semaphores *Sa*, *Sb* and *Sc*?

- *Sa* = 1, *Sb* = 0, *Sc* = 0

  (alternative: *Sa* = 0, *Sb* = 0, *Sc* = 0 – requires different initialization)

3.2 Resource allocation graphs

Three programs **Pa**, **Pb** and **Pc** with functions printing their own name:

- **Pa**: a1(), a2() and a3() / **Pb**: b1() and b2() / **Pc**: c1(), c2() and c3()

```
void a1() {
  <possibly block here using wait()>
  printf("a1 ");
  <possibly signal here using signal()>
}
```

```
int main() {
  a1(); a2(); a3();
}
```

Three semaphores
Sa, *Sb* and *Sc*

Desired output: a1 b1 a2 c1 c2 b2 a3 c3

b. Fill in a table that indicates the required calls to the semaphore functions wait() and signal() in the respective functions of Pa, Pb and Pc

	a1	a2	a3	b1	b2	c1	c2	c3
wait(...)	*Sa**	*Sa*	*Sa*	*Sb*	*Sb*	*Sc*	*Sc/–*	*Sc*
signal(...)	*Sb*	*Sc*	*Sc*	*Sa*	*Sa*	*Sc/–*	*Sb*	–

** for the alternative from slide 5: "–"*

3.3 Even more semaphores

How many times does the following short C program print the letter X? Assume that the semaphore *sem* is initialized to the value 4.

```
int main(void) {
  for ( ; ; ) {
    printf("X\n");
    wait(&sem);
  }
  printf("X\n");
  return 0;
}
```

- **5 times**:
  - The **first** "X" is printed *inside* the `for` loop
  - sem is decremented: 4 → 3
  - The **second** "X" is printed *inside* the `for` loop
  - sem is decremented: 3 → 2
  - The **third** "X" is printed *inside* the `for` loop
  - sem is decremented: 2 → 1
  - The **fourth** "X" is printed *inside* the `for` loop
  - sem is decremented: 1 → 0
  - The **fifth** "X" is printed *inside* the `for` loop
  - wait ***tries to decrement*** sem, it is ***already 0*** → `wait` ***blocks!***
  - no process `signal()`s sem → no further `printf` is executed!

3.4 Synchronization using interrupts

On x86 CPUs, interrupts can be disabled and reenabled using the machine instructions cli and sti. Why is this a significant problem (and, as a consequence, not allowed to be performed by regular user programs)?

- Disabling interrupts affects ***all processes!***
  - The sti/cli instructions are "all or nothing": disable or enable ***all possible interrupts*** of the CPU
  - …not only for the processes that want to synchronize
- In addition, the OS can be affected itself, since it needs interrupts for its own operation
  - e.g. timer, device interrupts
  - forgetting to re-enable interrupts hangs the whole system!

3.4 Synchronization using interrupts

Example: ***timer interrupts***

- The timer interrupt ("tick") handler is triggered every millisecond on x86:

```
<asm/param.h>:
#define HZ 1000  /* internal kernel time frequency */
```

- The interrupt handler increments the internal "`_jiffies`" variable*
  (Implemented in `kernel/sys_call.s`)

```
_timer_interrupt:
   …                    // save registers
  incl _jiffies
  …                    // restore registers
  jmp ret_from_sys_call
```

*** e.g. in Linux 0.12 – see https://github.com/Original-Linux/Linux-0.12**

- The hardware only has one bit per interrupt to indicate that there was a request.
- If multiple interrupts occur between cli() and sti(), the handler is executed only once! ➛ `_jiffies` is incremented only once instead of multiple times!
  ➛ the system clock ***"loses time"***