Operating systems

ELTE IK.

Dr. Illés Zoltán

zoltan.illes@elte.hu

Financed from the financial support ELTE won from the Higher Education Restructuring Fund of the Hungarian Government

We have talked earlier…

- Progress of Operating systems
  - Computer – Op.system generations
- Notions of Operating system, structures
  - Client-server model, …
  - System calls
- Files,directories, filesystems
  - Hardware architecture
  - Logical architecture
  - FAT, UNIX, NTFS,…

What comes today…

- Processes
  - Creating, ending
  - States of processes
- Process communication
  - Race situation
  - semaphores, mutexes, monitors
- Classical IPC problems

The model of processes

- Program – process differences
- A process is a running program in the memory (code+I/O data+state)
- How many processes are working at the same time?
  - Single Task – Multi Task
  - Real Multi Task?
- Sequential model
- Switching between processes: multiprogramming
- At the same time there is only 1 active process.
  - In case of more cores there are more active process

System model

- 1 processor + 1 system memory + 1 I/O device = 1 task execution
- In Interactive (window) systems, there are more than one process in the memory
  - Environment switching: Only the program located in foreground is running.
  - Cooperative system: the current process periodically offers the cpu control to other processes. Ask a question: who wants the CPU? (Win 3.1)
  - Preemptive system: the kernel takes back the cpu after a predefined time, and gives it to the next process.
  - Real-time systems

Creating processes

- Recently there are used preemptive systems (Linux, Windows)
- There are more than one active, living process.
- Reasons of process creation:
  - System initialization
  - A process creator system call
    - Copy of origin process(fork)
    - Exchange of the origin(execve)
  - A user command (command&)
  - Etc.
- There are foreground processes
- Background processes (daemons)

Process relations

- Parent – child relation
- Process tree:
  - Every process has a parent
  - A process can have more than one child
  - Process subtree
  - E.g.: Init, /etc/rc script execution
    - The id of the init is 1.
  - Fork command…Be careful using it
- Reincarnation server
  - Some services has such a feature.
  - In case of death of a process, the control service (reincarnation server) creates a new one instead of death process.

End of processes

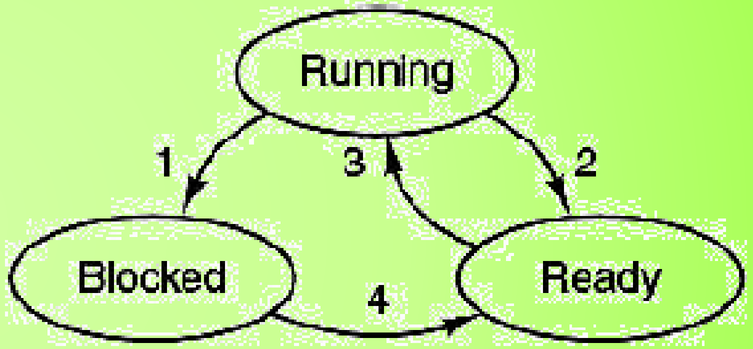
- After starting, the process finishes its task within a given time frame.
- Reasons of a process end:
- Voluntary end
  - Normal process end (exit, return commands.)
  - On error, the program ends his run(also execute a return command)
- Unintended end
  - Illegal command causes a process end, (divide by 0, illegal memory usage, etc)
  - Outside „partner” helps in finishing. Other process sends a signal.

State of processes

- A process: a program in the memory with instruction pointer, stack, etc.
- The processes usually are not independent.
  - Some process uses other results.
- A process can be one of the next 3 states:
  - Runing
  - Ready to run, temporary it waits for a CPU time frame.
  - Blocked , it can not continue his execution, e.g. the process needs an other process result to run! (cat Fradi.txt|grep Fradi|sort, grep and sort are blocked in the beginning…)

State transitions

1. Running -> Blocked
   - Has to wait for something
2. Running ->Ready to Run
3. Ready to Run -> Running
   - Scheduler makes decision, the processes do not know about it.
4. Blocked->Ready to Run
   Waited data is arrived



Implementation of processes

- A CPU executes only the actual command, pointed by CS:IP
- Only one process is running at one time.
- The CPU does not know about processes:
  - During process change (switch) what do we have to store to be able to properly continue the original process?
  - Mostly everything,….execution counter, registers, memory state, file descriptors, etc.
  - These informations are stored in the process table(process control block, CPU helps to manage this, State segment, etc)
- I/O interrupt table

When and how to change a process to another?

- When: During timer interrupt, special system call etc.
- Scheduler saves process information into process table.
- Loads the next process state, and runs it! (Set CPU registers back, etc).
- Can not save chache store
  - Often changed – needs more resources
  - The perfect changing time interval is not an exact value (not obvious).

Process Control Block (PCB)

- It is built during system initialization
  - Contains process informations
- This is like an array (based on PID) – each array element a composed structure containing process parameters.
- Major parameters of a process:
  - Process IDentifier (PID), name (name og program)
  - Owner, Group ID
  - Memory occupation, register data
  - etc.

Threads

- General situation: One process – one instruction set – one thread
- Sometimes we need to create inside a process more threads.
  - Thread: It is a standalone instruction set (typically a function) inside a process.
  - Often called: „lightweight process”
- Threads: Independent instruction sets.
  - Inside a process can be only one
  - Inside a process can be more-If one of them is blocked, the complete process will be blocked!
  - Thread table – for keeping a record of threads
- The thread has the same memory space as his parent!

Process–Thread properties

- Only a process has:
  - Address space
  - Global variables
  - File descriptors
  - Child processes
  - Signal handling, wake up listeners
  - …
- A thread also has:
  - Process counters (instruction pointer, etc.)
  - Registers copy, own stack

Thread problems

- Be careful using fork– If a parent process has more threads, the child also will have!
- File management– What to do if a thread closes a file while an another thread uses it?
- Error management – errno is a global variable
- Memory management–…
- Relevant: A system call can handles threads (thread safe call)

Communication of processes

- IPC – Inter Process Communication
- We have three major problem during process collaboration:
  - A process cannot be interrupted during execution a „critical, sensitive instruction block”.
  - A serial execution of subtasks (joining). (We can print only after collecting printing information!)
  - How can a process send any information to an other one?
- All three problems fits to threads too, but the third one is not causes any problem because of same memory space!

Parallel systems

- Scheduler swithes processes quickly, so it gives a „parallel” execution feeling.
- Multiprocessor systems
  - 2 or more processors
  - Higher computing power
  - Do not increase reliability
  - Clusters – Hardware parallelism
  - It is to increase reliability!
- Key question: How can we use common resources?

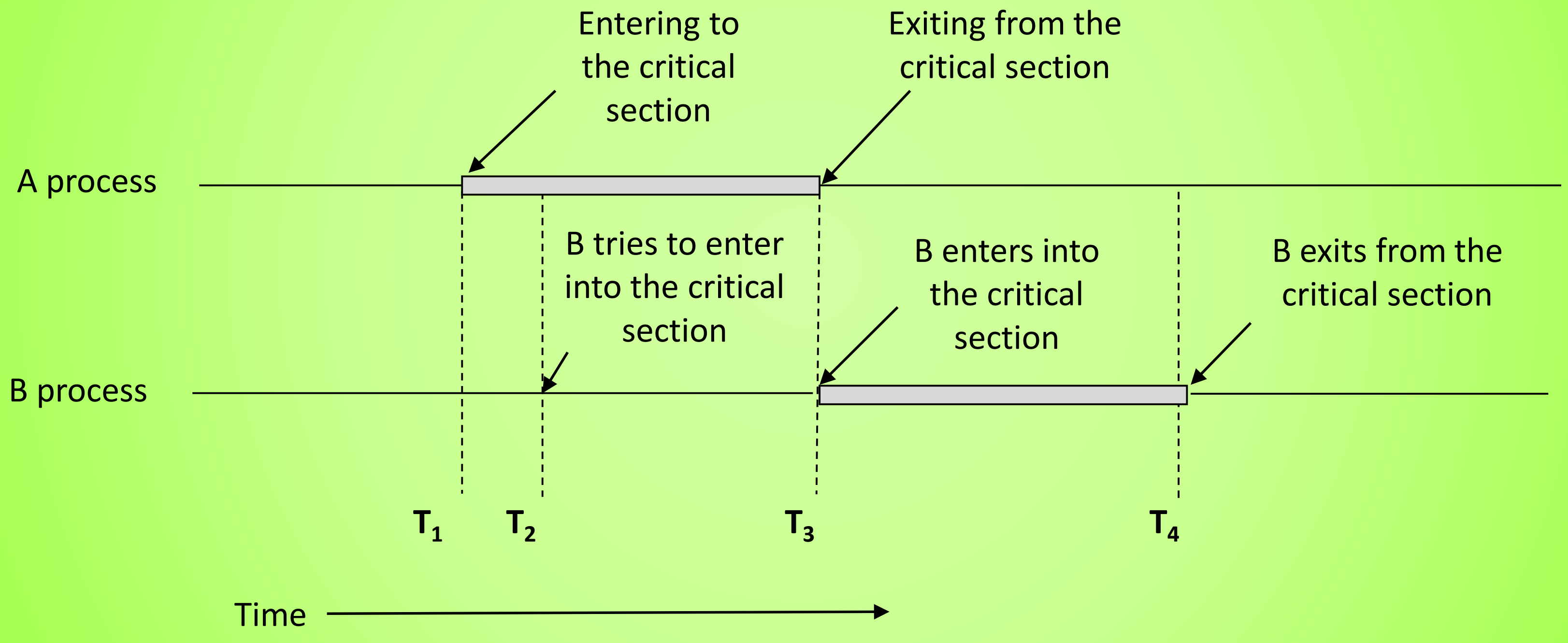
Common resources

- When 2 or processes want to use the same memory…
  - E.g.: 2 printing task, common printing queue
- Race condition– place: 2 or more processes wants to read/write common memory space, the result often is undetermined, it’s related to the time moment!
  - This error is hard to discover often!
- Solution: We need a method which guarantees the common resource access only by 1 process at the same time!

Mutual exclusion

- Critical section: That code part which works with a resource(memory) and can be accessed by several processes.
- A good mutual exclusion suits the following requirements:
  - No two processes in the critical section at the same time.
  - It is no related to speed, CPU os other parameter.
  - Any process, outside the critical section, can not block an another process.
  - A process does not wait forever to enter to the critical section.

The required behavior of the mutual exclusion


Entering to the critical section
Exiting from the critical section
A process
B tries to enter into the critical section
B enters into the critical section
B exits from the critical section
B process
T_1
T_2
T_3
T_4
Time

Mutual exclusion implementations I.

- Disabling interrupts (all)
  - After entering crit. section interrupts will be disabled (STI- set interrupt flag to 1)
  - On exit: Enable interrupts (CLI- Clear interrupt)
  - It is not perfect because a user level program can not disable interrupts.
- Common 2 state(enable-disable) variable
  - 0 (free) and 1 (reserved) critical section
  - 2 process might be in critical section!
    - 1 process enters, and before setting this variable to 1, kernel changes the running process!!.

Mutual exclusion implementationsII.

- Strict changing:
  - We can implement it for more then 2 processes.
  - It does not grant the mutual exclusion requirement no.3.(Outside process cannot block an another process)! If ha process 1 is slow, in non critical section, and process0 quickly enters into critical section and leaves that, and finish its non critical section too, then process0 will be blocked by process1!
  - Process0 Process1

```
while(1)
 {
   while(next!=0) ;
   critical_section();
   next=1;
   not_critical_section();
 }
```

```
while(1)
 {
   while(next!=1) ;
   critical_section();
   next=0;
   not_critical_section();
 }
```

G.L.Peterson’s solution

- 1981, an improvement of strict changing
- Before critical section every process must call an enter function and after it call an exiting function.

```
#define N  2
int next;
int wants[N];
/* the modified process*/
while(1)
  {
    enter_func(process);
    critical_section();
    exit_func(process);
    not_critical_section();
  }
```

```
void enter_func (int proc)
{
  int other;
 other =1-proc; //N=2 so other proc is
  // other =(proc+1) % N;
  wants [proc]=1; //proc 1 wants to run
 next =proc;
  while(next ==proc &&
             wants [other]);
}
void exit_func (int proc)
{
  wants [proc]=0;    //outside from crit.
}
```

Little Peterson „modification”– big mistake

- Lets proc=0!
- At the signed line the schedules changes, so proc=1 now!
- The wants[0] is 0, therefore process 1enters into critical section!
- While process 1 is working in critical section, lets the schedules changes running process again, now back to process 0! In this case the variable „next” value is 1, proc is 0 so the next==proc will fail therefore the process 0 also enters into critical section!

```
void enter_proc(int proc)
{
  int next;
  next=1-proc; //so N=2…
  // next=(proc+1) % N;
 next=proc; //2 lines change
//* here changes the scheduler
 wants[proc]=1;//proc wants  run
  while(  next==proc &&
             wants[masik]);
}
void exit_func(int proc)
{
  wants[proc]=0;   //false
}
```

Busy waiting, machine code

- TSL instruction – Test and Set Lock
  - Atomic instruction (it can not be interrupted)

```
enter:
        TSL register, LOCK      ; LOCK value stored in the
                                ; register
                                ; and LOCK=1
                                ; during TSL,the CPU locks
                                ; memory address bus
        cmp register,0
        jne enter               ; if no 0, jump
        ret
;
exit:
        mov LOCK,0
        ret
```

Busy waiting : summary

- Both (Peterson and the TSL) solution works well, only they are waiting in a cycle.
- These solutions uses an „empty waiting cycle” – this method is called: Busy waiting!
- It wastes CPU time…
- It would be better to use a process blocking method insted of busy waiting! During process block, it do not use the CPU!!

Sleep – wakeup

- Busy waiting solution is not so effective!
- New solution: Using a block(sleep) method instead of busy waiting, and use a wake up method if needed!
  - sleep –wakeup, down-up keywords
  - These functions may have parameters too!
  - Tipical problem: Producer – Consumer problem

Producer-Consumer problem

- Well known problem as a confine (ranged) variable problem too.
- E.g: Baker-Bakery-Consumer „triangle”.
  - Baker is baking bread, until in bakery there is empty shelves(space).
  - A customer, consumer can buy a bread if bakery is not empty.(There is minimum 1 bread…)
  - If the bakery is full, than baker goes to sleep (rest) .
  - If the bakery is empty, than the customer will wait(sleep) for bread.

A Solution of the producer–consumer problem

- Baker(producer)   Customer (consumer)

```
#define N 100
int place=0;
void baker()
{
  int bread;
  while(1)
   {
            bred=new_bread()
            if (place==N) sleep();
            to_shelf(bread);
            place++;
            if (place==1)
                wake_up(customer);
   }
}
```

```
 void customer()
{
  int bread;
  while(1)
   {
            if (place==0) sleep();
            bread=ask_bread();
            place--;
            if (place==N-1)
                wake_up(baker);
            eat(bread);
   }
}
```

Baker-Customer problem

- The access of variable „place" is not restricted, this will cause a race situation.
  - The customer cheks place, and the content is 0, after this checking the schedules changes the running process. The baker increases the variable place, and the content will be 1. In this case bakes sends a wakeup signal to customer. This signal will be lost, because the customer is not sleeping!
  - The customer gets back runing state, and will sleep because he thinks the variable place is 0.
  - The baker will bake all N bread and he also will sleep!
  - So now both processes are sleeping!(deadlock)
- We can use a wakeup bit too, but the problem will remain!

Semaphores I.

- Suggested by E.W. Dijkstra (1965), this is a special „variable" type.
- It is an integer variable.
- A semaphore stops the process, if its value is 0.
  - The process will wait,sleep, will not use CPU, it stops before semaphore.
- If the semaphore >0, the process can enter into critical section.
- The semaphore has 2 operation:
  - Down: Entering into critical section the value will be decreased.
  - Up: Exiting from the critical section the value will be increased!
  - These operations was called as P and V by Dijkstra!

Semaphores II.

- Atomic instruction: a semaphore value checking, modification, it can not be interrupted!
- This guarantees to avoide race situation.
- If a semaphore has only 2 value, the value 1 means the process (train) can continue his work, the 0 means the process (train) will stop!
  - Binary semaphore
  - This often called as MUTEX (Mutual Exclusion).

semaphore implementations

- Up, Down atomic instructions.
  - It can not be interrupted!
- How it is supported?
  - With kernel level system call, from user level can not.
  - At the begining of system call all interrupt will be disabled.
- semaphore instructions are in kernel level!
- It is accessed from development environment, supported by operating systems!

A Solution of the producer-consumer problem with semaphores I.

- Producer (baker) function

```
typedef int semaphore;
semaphore free=1;  /*Binary semaphore,1 may go on,  free sign*/
semaphore empty=N, tele=0;  /* empty the shelf, this is a free sign*/
void baker()                  /* N value „the size of bread-shelf" */
{
  int bread;
  while (1)
   {
     bread=baker_bake();
     down(&empty);                  /* empty decrease, if before >0, go on*/
     down(&free);             /* May we change the shelf of bakery? */
     bread_to_shelf(bred);  /* Yes, put on a bread*/
     up(&free);               /* Release the shelf of the bakery. */
     up(&full);               /* Sign to the customer, there is a bread. */
   }
}
```

A Solution of the producer–consumer problem with semaphors II.

- Consumer (customer) function.

```
void customer()                /* customer semaphore is full */
{
  int bread;
  while (1)
   {
     down(&full);              /*full is decreased, if before >0, go on */
     down(&free);  /*May we modify the shelf of the bakery? */
     bread=bread_polcról();  /* Yes, take away a bread. */
     up(&free);                /* Release the shelf of bakery. */
     up(&empty);               /* Sign to the baker, there is empty place,
you may bake! */
     bread_eating(bread);
   }
}
```

Summary of semaphore sample

- free: it safes the bread shelves, so it is accessed only by 1 process(the baker or the customer)
  - Mutual exclusion
  - Atomic instructions(up, down)
- full, empty semaphore: syncronisation semaphors, the producer stops if the shelves are full, or the consumer stops is the shop is empty.

semaphore functions,sample

- Unix environment:
  - semget: creating semaphor(System V)
  - semctl: semaphore control, read, write value
  - semop: semaphore operation (up,down)
  - sembuf structure
  - On practice, you'll see samples!
  - sem_open,sem_wait,sem_post,sem_unlink (Posix)
- C# sample.
  - VS 2008.
  - The implementation of baker-customer sample.

Thanks for your attention!

zoltan.illes@elte.hu