12-1-15

p.1

OPERATING SYSTEMS

Mondays + Thursdays 12:00 noon to 1:25 pm

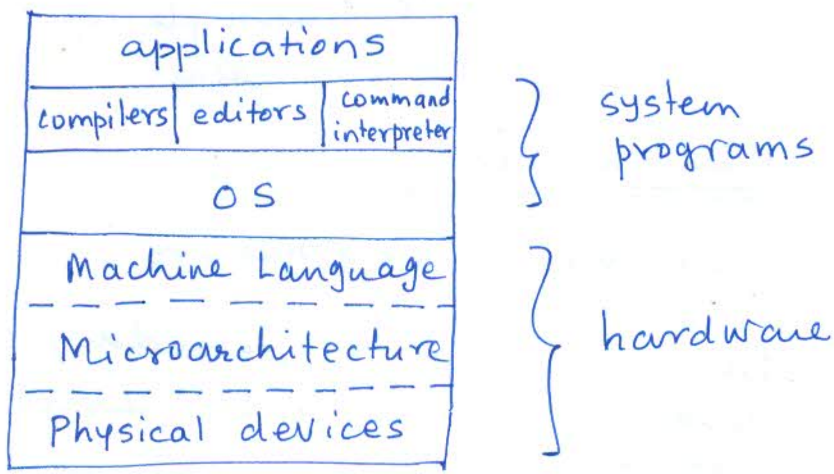
Text book -

Modern Operating Systems 2nd ed.

by A. S. Tanenbaum

PHI

INTRODUCTION



1.1.1 The Operating System as an Extended Machine

1.1.2 . . . . as a Resource Manager

brief history -

technology → mainframes / minis ....

CPU speed vs. I/O speed

multiprogramming

timesharing

UNIX – POSIX – LINUX ....

Windows / GUI

Network OS vs Distributed OS

→ problems with multiple copies of data

Hardware review

CPU ↔ cache ↔ memory

fetch
decode
execute

{ secondary storage
{ I/O devices

{ DMA
{ programmed I/O

interrupt mechanism

interrupt handling

buses

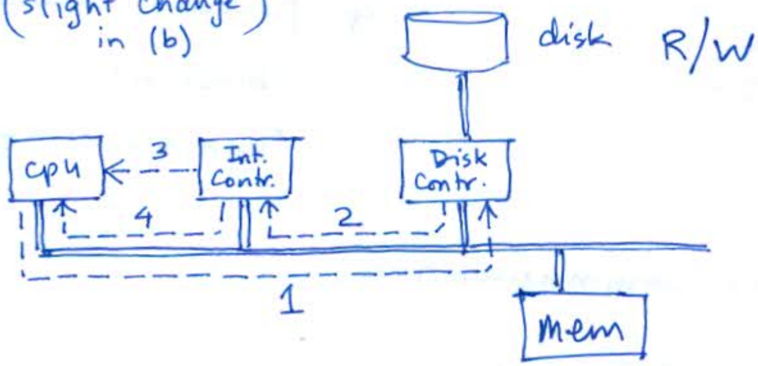
controllers

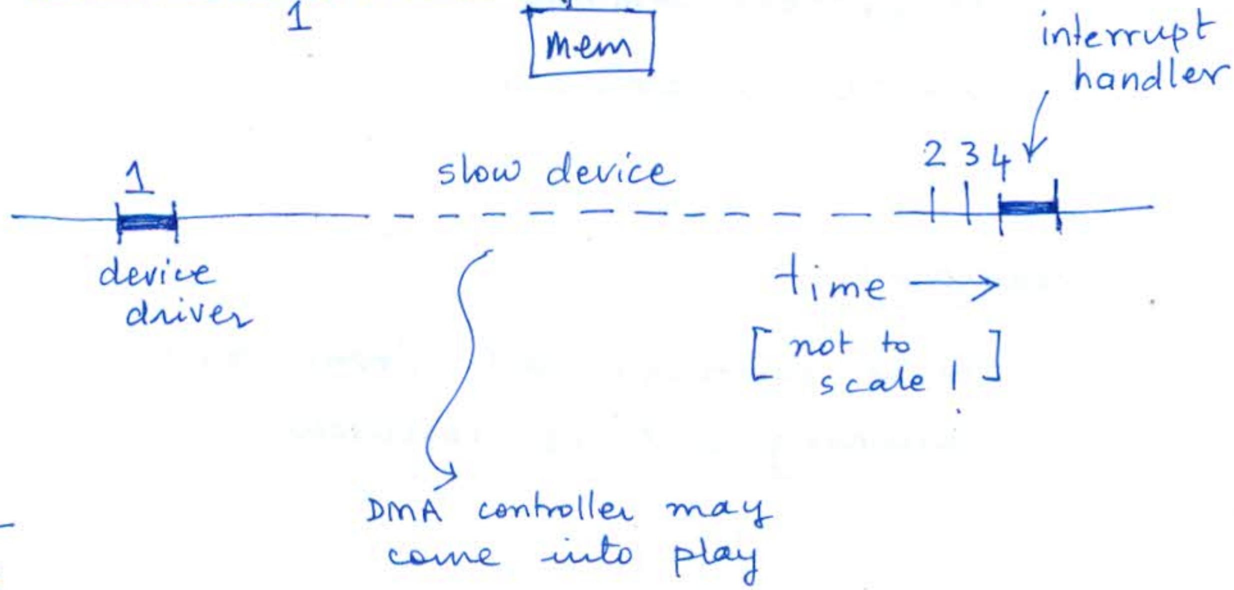
bootstrapping

8-1-15
p.1

interrupts - why?

* Fig 1-10
(slight change
in (b))





15-1-15

1.5

OPERATING SYSTEM CONCEPTS

process program in execution

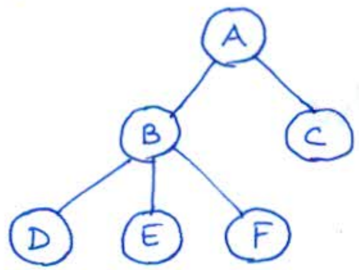
address space of process ["core image"]

process table

creation / termination

parent process - child process

p.2 process tree Fig 1-12



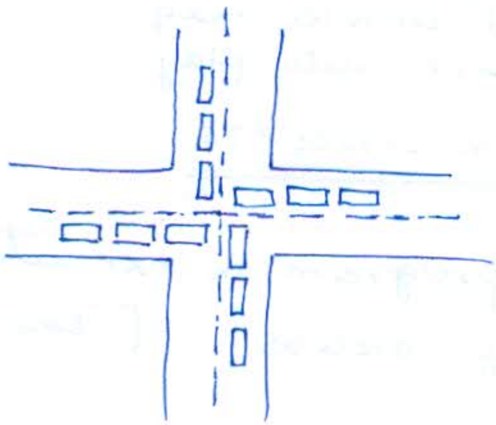
⇒ need two flavours of process creation
"blocking"
"non-blocking"

* interprocess communication

user ID vs. process ID

Deadlocks

– to be distinguished from simply "running out of resources"



Memory management

SIMULATION

Secondary storage
|
File system

19-1-15 (p.1)

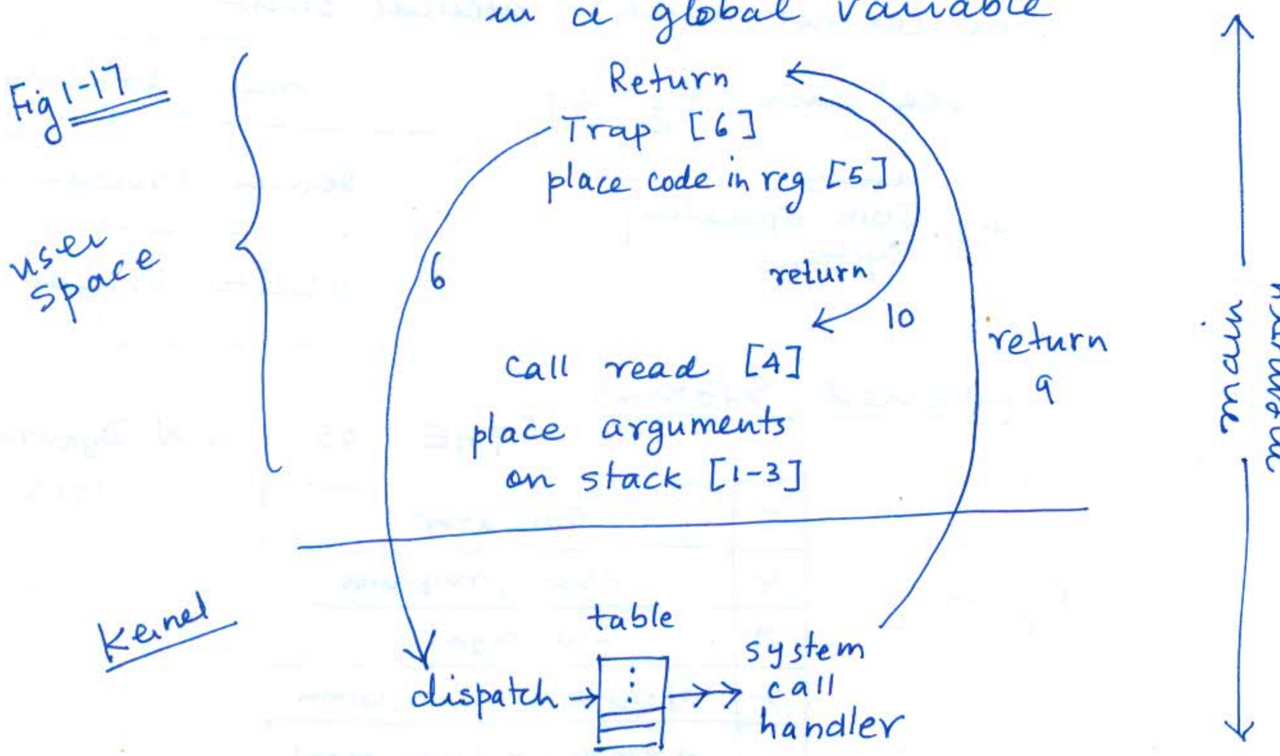
1.6 SYSTEM CALLS

* interface between user programs & OS
* difference between usual function calls (a.k.a. procedure calls) and system calls → the latter "trap" to OS kernel

e.g. count = read (fd, buffer, nbytes).

if EOF, then count < nbytes
if error, count = −1 and error code in a global variable

Fig 1-17



p.2

* POSIX - IEEE standard

Fig. 1-18 - Self study

Fig. 1-23 - Win32 API (subset)

Recall - OS as "Extended Machine"

1.7 Operating System Structure

Monolithic Systems

- collection of procedures
- one big object file
- no (or not much) internal structure

see also Fig 1-24

main procedure

service procedures

utility procedures

e.g early IBM operating systems

Layered Systems

"THE" OS - E.W. Dijkstra 1968

Fig 1-25

5	Operator
4	user programs
3	I/o mgmt
2	operator-process comm.
1	memory + drum mgmt
0	processor allocation ← - - multiprogramming

Virtual Machines

"identical copies" of hardware

[trap to h/w as needed]

Fig 1-26 shows VM/370 with CMS

extension of this concept

Exokernels

each defined VM has specified subset of hardware resources [not a copy of the "whole mc"]

Client-Server Model

- on one system [with "microkernel"]
- on a distributed system

* recall difference between Network OS and Distributed OS

22-1-15 (P.1)

Chap. 2 PROCESSES

"most central concept"

- resources must be fully utilized
- multiple activities in progress at any one time
- "pseudo parallelism" as opposed to (h/w) multiprocessing
- process – conceptual model (sequential) which helps deal with parallelism

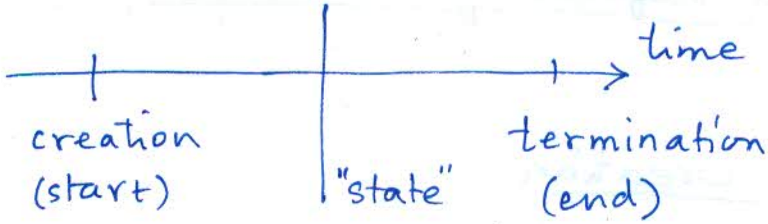
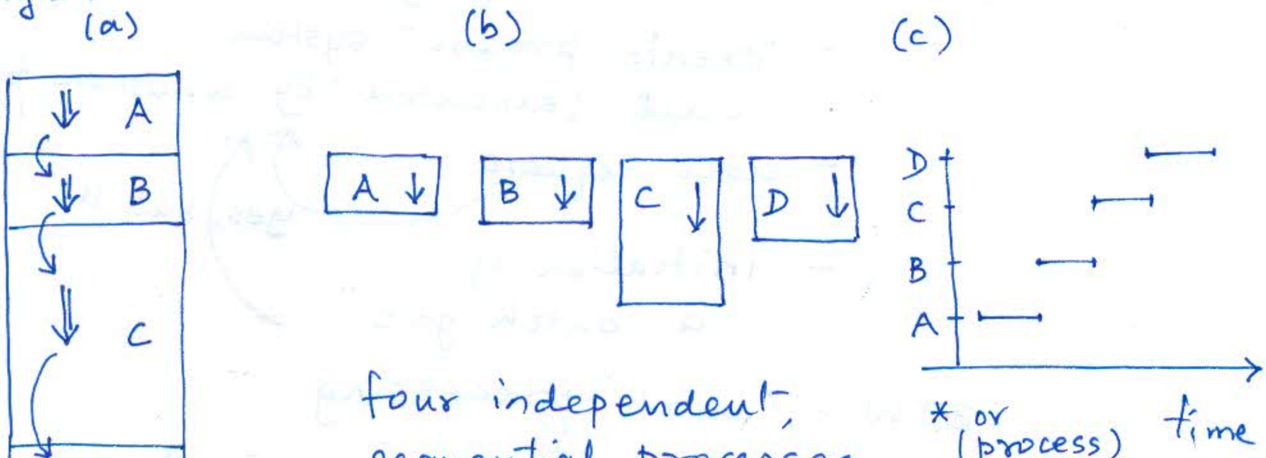


Fig 2-1

one PC



(a) four programs in memory

(b) four independent, sequential processes each with "own PC"

(c) only one program* executing at any time

* or (process)

p.2

- CPU switches back and forth between processes
- rate of progress of any process depends on total system load
- but RESULTS DO NOT !

* [In certain special systems, real-time constraints are important.]

  need Real-Time Operating System

Q: What defines process "state" ?

Process creation

At:
- system initialization
- "create process" system call (executed by another process)
- user request
- initiation of a "batch job"

yes, but !!!

BTW - "batch" processing vs "continuous" processing

background vs. foreground processes

"daemons"

22-1-15 (p.3)

UNIX - "fork" system call
- creates copy (clone) of calling process
[usually followed by "exec"]
- each process has its own address space
[code can be shared] * → "pure" or "reentrant" code

note: resources needed for process creation is must be available for the system call to succeed.

Process termination

"exit()" → Normal exit
error exit } "voluntary"
fatal error
killed by another process } "involuntary"

Process hierarchy

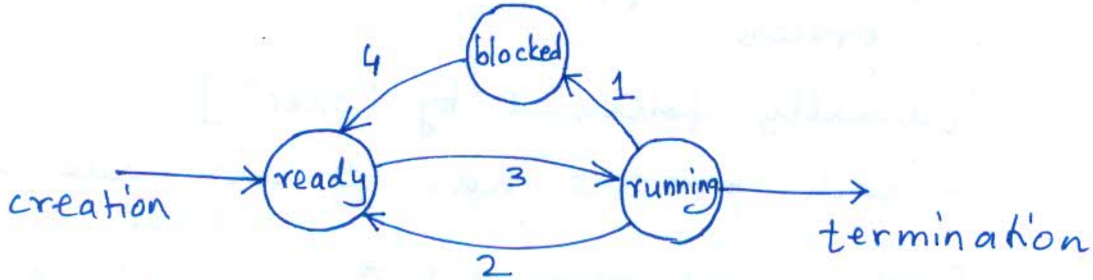
- easily seen in UNIX / LINUX ...
- not in WINDOWS

i.e. not an indispensable feature of an OS, but a neat and convenient one

can prove useful in managing processes

p.4

Process states



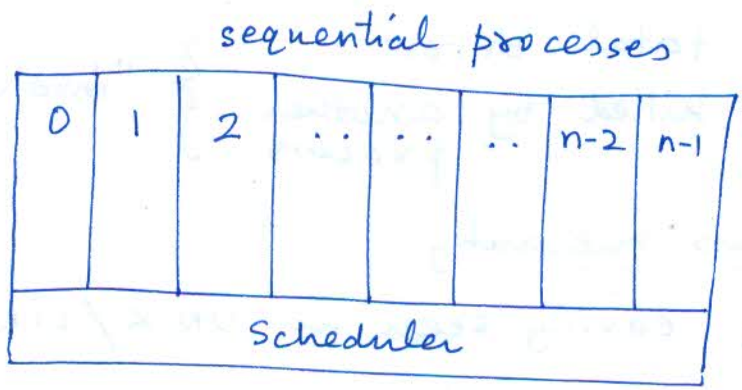
1: process blocks for I/o or another event

2: scheduler picks another process

3: " " this "

4: I/o completes or event occurs

Fig 2-3



← lowest-layer of OS handles interrupts & scheduling

Such a system would be described as a "process-structured" operating system.

29-1-15

(P.1) Implementation of Processes

Process Table - data structure e.g. array of structures ["process control blocks"]

Fig 2-4 Typical Process Table entry

Process Mgmt - registers, PC, PSW, SP ...
process state, priority
scheduling parameters
ID, parent ID, process group
signals
Time started, CPU time used ......

Memory Mgmt - Pointers to text, data and stack segments
[UNIX terminology]*

File Mgmt - Root directory
Working "
File descriptors
User ID, Group ID

note: various fields listed above help with* resource mgmt
or required for

Fig 2-5 (simplified) what an OS does upon an interrupt

1. Hardware stacks PC
2. " loads new PC from interrupt vector
3. A.L. procedure saves registers
4 " " " sets up new stack
5 ISR runs [can be in C]
6 Scheduler decides which process is to run next
7-8 A.L. procedure starts up (resumes) selected process

interrupted process and selected process (step 6) may be different

* hardware dependent code is written in Assembly language [small percentage of OS code]

29-1-15 (p.3)

2.3
THREADS

process provides (so far) [(a) resource grouping; single thread of execution (b)]

often it is useful - from appn point of view - to provide (a) with multiple threads (of execution)

for execution, each thread must have its own PC, registers, stack & SP, state

threads share all other resources
- address space, open files, accounting, access rights .....

- sometimes known as "light-weight processes"
- capability known as "multithreading"

* self study - Fig 2-6, Fig 2-7

- at the time of creation, a process (usually) starts off as a single thread

p.4 as needed, "system calls" such as thread-create
thread-exit are invoked
thread-wait
thread-yield

"careful thought and design are needed to make multithreaded programs work correctly"

Fig 2-9 a word-processor with 3 threads

- user interaction
- reformatting (b/g)
- periodic saving of file to disk (b/g)

Figs 2-10, 2-11

dispatcher thread:

```
while (TRUE) {
        get_next_request(&buf);
        handoff_work(&buf);
}
```

9-2-15
(P.1)

2.3 INTERPROCESS COMMUNICATION

need
- explicit
- implicit ← sharing of system resources

three types of communication / coordination

(i) one process passes information to another

(ii) processes should not "get into each other's way" during critical activities

(iii) logical dependencies

Issues are applicable to threads as well, except that:

(a) they have common address space

(b) they are part of a common* application (and presumably correctly programmed as such)

*[multithreaded]

P.2 2.3.1

Race conditions

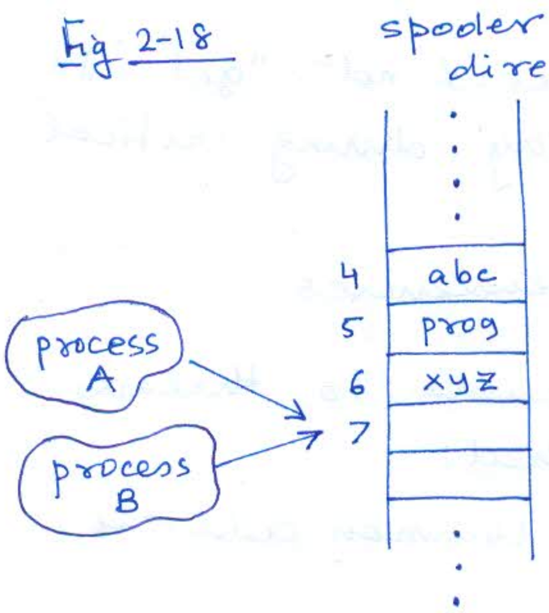
→ when two or more processes access shared storage (main memory or on a file)

Example –

Print spooler

spooler directory – printer daemon

Fig 2-18



out = 4
in = 7 } two shared variables

two or more processes are reading and writing shared data, but the final result* depends on which process runs when

* correctness of the final result

[recall that interrupts and the scheduler are running independently]

9.2-15

(p. 3)

2.3.2 Critical regions

→ two (or more) processes should not read & write shared data "at the same time" (i.e. in any possible sequence of reads & writes)

→ i.e. if one process is changing the value of a shared variable, other processes should be excluded from doing the same thing until the first one has "completed its work on the shared variable"

→ concept of critical region* helps us to formulate a solution

[or critical section]

→ no two processes should be in
(1) their respective critical region at the same time

Also – (2) no assumptions about speeds or number of processors

"good solution" (3) no process outside its CR may block other processes

(4) no process should have to "wait forever" to enter its CR

p.4

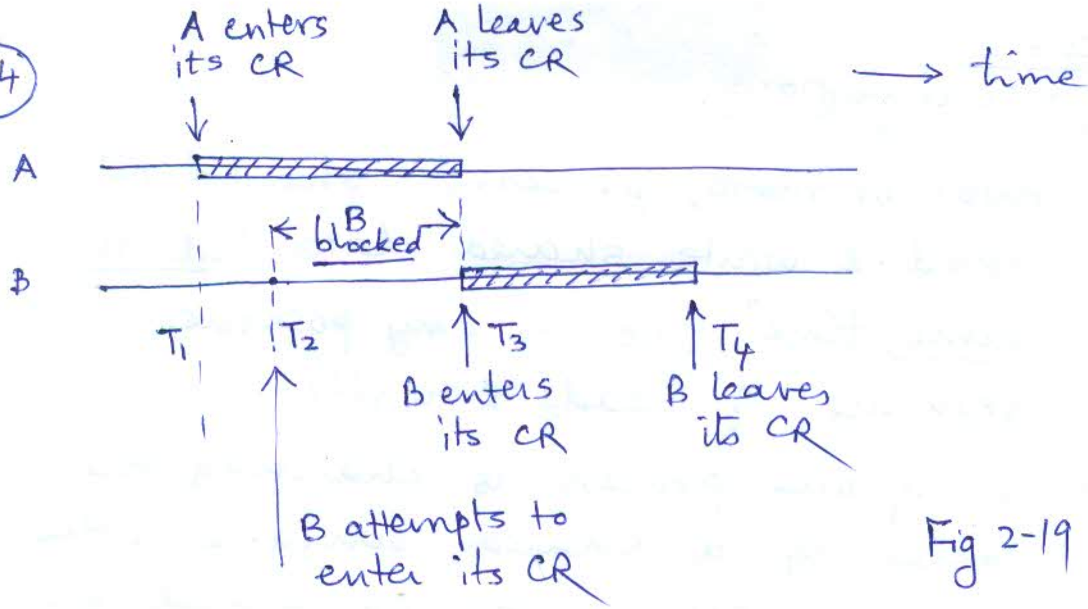


Fig 2-19

2.3.3

Mutual exclusion with "busy waiting"

* disabling interrupts
* lock variables
* strict alternation — violates condition 3 seen above
* Petersen's solution

"busy waiting" is not really acceptable in a good solution

→ must look for a better solution

12-2-15

p.1

TSL machine instruction

TSL → stands for "Test & set Lock"

TSL RX, LOCK

* LOCK is a memory location

indivisible action: $\begin{cases} RX \leftarrow (LOCK) \\ (LOCK) \leftarrow 1 \end{cases}$

even on a multiprocessor system

Fig 2-22

```
enter_region:  TSL   Ri, LOCK
               CMP   Ri, #0
               JNE   enter_region
               RET

leave_region:  MOVE  LOCK, #0
               RET
```

each process contending for a shared resource must:

(a) call enter_region at point of entry to its CR

(b) call leave_region at point of exit from its CR

But note: busy waiting has not yet been removed from the solution!

P.2 The "Producer-Consumer problem"

- two processes share a fixed size buffer (which can store N items)

Consider Fig 2-23

```
#define N 100
int count = 0;

void producer(void)
{
    int item;
    while (TRUE) {
      item = produce();
      if (count == N) sleep();
      insert_item(item);
      count++;
      if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
      :
      if (count == 0) sleep();
      item = remove_item();
      count--;
      if (count == N-1) wakeup(producer)
      consume_item(item)
      :
```

loop

loop

(p.3) But there is still a race condition -

consumer reads 'count' when it is 0

* — process switch — *

producer inserts item, sets count to 1
sends "wakeup" to consumer

* — process switch — *

consumer goes to sleep
(as per code)

Problem - the wakeup call sent by the producer is "lost"!

2.3.5 Semaphores E.W. Dijkstra (1965)

→ solve the problem of "lost wakeups"

→ integer valued variables with two operations defined up()
down()

→ no busy waiting

→ if "no stored wakeups", process is blocked [as part of down()]

→ down() & up() ⇒ ATOMIC

P.4 Solution of Producer-Consumer problem using semaphores –

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

producer loop

```
item = produce();
down(&empty);
down(&mutex);
insert_item(item);
up(&mutex);
up(&full);
```

consumer loop

```
down(&full);
down(&mutex);
item = remove_item();
up(&mutex);
up(&empty);
consume(item);
```